Ejercicios de clasificacion y regresióo

Julio A. Fresneda

May 18, 2018

Clasificación de números a mano alzada

En esta práctica, tenemos una base de datos que contienen información sobre números de 0 al 9 escritos a mano alzada. Concretamente, cada número de la base de datos tiene asignado 64 valores que van entre 0 y 16 y corresponden a la intensidad de cada cuadrícula de una matriz 8x8, de forma que dibujando estos valores sobre la matriz obtenemos el número dibujado a mano alzada.

Objetivo de la práctica

Nuestro objetivo es estimar una función que, a partir de 64 valores de intensidad, estime adecuadamente a qué número corresponden.

Método a utilizar

Para obtener esta función, vamos a usar regresión logística.

El método de regresión logística que vamos a usar clasifica de forma binaria, es decir, entre dos etiquetas. Esto es un problema ya que tenemos 10 números que clasificar, no 2. Para ello, usaremos Softmax.

Para poder estimar un modelo que clasifique correctamente cada número, vamos a seguir los siguientes pasos:

- Clasificaremos cada número contra el resto. Es decir, para cada número, estimaremos un modelo que nos de la probabilidad de si el input corresponde con ese número o no.
- Usaremos Softmax para obtener un modelo de clasificación. Para cada input, estimamos su probabilidad para cada uno de los modelos obtenidos antes, y lo etiquetamos con el número cuyo modelo tenga más probabilidad.

Para clasificar cada número contra el resto, como ya hemos dicho, usaremos regresión logística.

De esta forma obtendremos 10 hipótesis de clasificación. Una nos devolverá la probabilidad de que el input sea 0, otra nos devolverá la probabilidad de que sea 1, etc.

Para etiquetar, usaremos softmax, el cual se explicará más profundamente más adelante.

La hipótesis que usa regresión logística es la siguiente.

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^T x^{(i)}}}$$

De esta forma el valor que obtenemos siempre está entre 0 y 1.

En código, la función es la siguiente

```
hx <- function(x,theta)
{
  1 / (1+exp(- t(theta)%*%x) )
}
```

Lo que nos interesa es encontrar los valores de θ que nos ayuden a predecir correctamente. Una forma de conseguirlo, es minimizar el error de clasificación. El error de clasificación, en regresión logística, sin regularizar, se puede expresar de la siguiente manera:

$$J(\theta) = -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} log\ h_\theta(x^{(i)}) + (1 - y^{(i)}) log\ (1 - h_\theta(x^{(i)}))]$$

Una forma de obtener los valores de θ que buscamos, es minimizar este error.

Para minimizar esta función, vamos a usar el gradiente descendiente.

El gradiente descendiente es un método que aproxima los pesos θ actualizándolos iterativamente de la siguiente forma:

$$Para\ cada\ \theta_j \in \theta,\ simultáneamente : \theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j}$$

Donde α es el learning rate (usaremos $\alpha = 0.1$) y $\frac{\partial}{\partial \theta_j}$ es la derivada parcial de la función de error respecto a un θ_j concreto.

Para usar el gradiente descendiente, vamos a necesitar calcular la derivada de la función de coste. Si usamos la derivada y no la propia función, es por que la derivada es más fácil de converger.

La derivada, sin regularizar, de la función de error de regresión logística es la siguiente.

$$\frac{\partial}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^i) x_j^{(i)}$$

En código, el gradiente descendiente se implementa de la siguiente manera:

```
GD <- function(x,y,theta,lambda,lr=0.1,num_iteraciones=100)
{
  temp = theta

  for( i in 1:100) ## Número de iteraciones, 200 en nuestro caso
  {
    for( j in 1:length(theta) )
    {
      temp[j] = temp[j] - lr*dJ(x,y,theta,j,lambda) # Aplica el gradiente para cada peso
    }
    theta = temp

  }

  theta
}
```

Siendo x los input, y los output, theta los pesos iniciales, lambda el hyperparámetro de regularización (lo explicaré más adelante) y lr el learning rate.

Preprocesamiento de datos

Vamos a cargar nuestras bases de datos.

```
digitosTrain <- read.csv("datos/optdigits_tra.csv", header=F, sep=",") # 1. lectura zip del train
X <- as.matrix(digitosTrain[-65])
#X = X/16                                  # Normalización
X <- cbind(rep(1,nrow(X)),X)              # Añadimos una columna de unos

Y <- as.matrix(digitosTrain[,65])
Yorigins <- Y                            # Lo usaremos más adelante
```

Hemos cargado los dígitos de entrenamiento, más tarde cargaremos los dígitos de test.

Pensé en dividir cada número de la matriz entre 16. Esto es para normalizar los valores de intensidad entre 0 y 1, de forma que 0 sea nada de intensidad, y 1 sea máxima intensidad. Pero las operaciones con decimales parece que tardan más en realizarse, por lo que he preferido no normalizar para que la ejecución sea más rápida.

También hemos añadido una columna de unos. Esto nos sirve para que la función que estimamos tenga término independiente.

Ahora, vamos a crear una matriz de etiquetas.

Cada columna de la matriz representará a un número del 0 al 9, excepto la última columna, que será la columna de etiquetas Y. En cada fila de la matriz, solo habrá ceros, excepto un 1 en la columna cuyo número coincida con la etiqueta de la última columna.

Por ejemplo, si tenemos una fila cuyo último número es un 4, la fila será la siguiente: 0,0,0,1,0,0,0,0,0,4

Esta forma de ordenar las etiquetas nos ahorrará código redundante en las operaciones siguientes.

El código es el siguiente.

```
  Ymatrix = NaN
  for( i in 0:9 )
  {
    for( j in 1:length(Y))
    {
      if( i == 0 && j == 1 )
      {
        Ymatrix = 0
        if( Y[j] == i ) Ymatrix = 1
      }

      else
      {
        if( Y[j] == i ) Ymatrix = c(Ymatrix,1)
        else Ymatrix = c(Ymatrix,0)
      }
    }
  }

  Ymatrix = matrix(Ymatrix,ncol=10,byrow=F)
  Ymatrix = cbind(Ymatrix,Yorigins)
```

Regularización

Un problema que se suele tener cuando estimamos un modelo a partir de unos datos de entrenamiento, es el overfitting. El overfitting se produce cuando el modelo se ajusta demasiado bien a los datos de entrenamiento,

ajustando también el ruido. Esto hace que, aunque tengamos un error muy bajo en el train, en el test el error se dispare, ya que el modelo ajustado no corresponde con la realidad.

Una posible solución al overfitting es la regularización.

La regularización consiste en aplicarle unas condiciones a los valores de θ para evitar el sobreajuste.

Vamos a regularizar imponiendo la siguiente condición a θ:

$$\sum_{j=0}^{n} \theta_j^2 \leq C$$

Esta condición incita a que algunos valores de θ se aproximen a 0, simplificando la función $h_\theta(x)$, por lo que se corregiría el sobreajuste.

Si queremos aplicar esta regularización, necesitamos modificar nuestra función de error y su derivada.

La función de error regularizada es la siguiente:

$$J(\theta) = [-\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} log\ h_\theta(x^{(i)}) + (1-y^{(i)}) log\ (1-h_\theta(x^{(i)}))]] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

Como vemos, si minimizamos esta función con la penalización que le hemos puesto, le obligamos a que la suma de los cuadrados de cada θ_j sea pequeña. Es decir, buscamos pesos pequeños. La importancia que tenga esta condición respecto al resto de la función lo decidiremos con el valor de lambda.

Su derivada, regularizada, es la siguiente función:

$$\frac{\partial}{\partial\theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^i)x_j^{(i)} - \frac{\lambda}{m}\theta_j$$

En el caso del término independiente, θ_1, por convención no se aplica la regularización, por lo que sólo en ese caso se aplicaría la derivada original.

La función de error y su derivada, en código, con la regularización impuesta, son las siguientes.

```
J <- function(x,y,theta,lambda)
{
  cost = 0
  for( i in 1:length(y))
  {
    cost = cost + y[i]*log(hx(x[i,],theta)) + (1-y[i])*log(1-hx(x[i,],theta))
  }
  cost = -cost/length(y)

  sumsquares = 0
  for( i in 1:length(theta) )
  {
    sumsquares = sumsquares + theta[i]*theta[i]
  }
  cost = cost + (lambda/(2*length(y)))*sumsquares
}



dJ <- function(x,y,theta,j,lambda)
{
```

```
  cost = 0
  for( i in 1:length(y) )
  {
    cost = cost + (    1 / (1+exp(-t(theta)%*%x[i,]))    -y[i])*x[i,j]
  }

  cost = cost/length(y)
  if( j != 1 ) cost = cost - (lambda*theta[j])/length(y) # No aplicamos regularización al término indep

  cost
}
```

¿Qué lambda usar? Cross-Validation

Ya tenemos casi todo listo para empezar a entrenar, pero aún nos falta escoger el lambda adecuado. Si escogemos un lambda demasiado pequeño, se regularizaría poco, por lo que seguiríamos teniendo overfitting (si lo hubiera desde un principio), y aunque el Ein no subiría, el Eout no bajaría. En cambio si escogemos un lambda demasiado grande, regularizaríamos demasiado, por lo que la función estimada se simplificaría mucho y tendríamos un problema de underfitting, lo cual dispararía el Ein y el Eout.

Para elegir el hyperparámetro correcto, usaremos Cross-Validation.

Cross-Validation consiste en dividir nuestro set de datos de entrenamiento en K sets. Una vez dividido, para cada K_i, $K_i \in K$, entrenamos nuestro modelo con el resto de sets, calculando el error medio.

Una vez hecho esto, podemos volver a hacer Cross-Validation con otro modelo o el mismo modelo con otros hyperparámetros, como en este caso, y comparar la media de errores, para estimar qué modelo es mejor.

Para elegir un lambda adecuado, se debe realizar Cross-Validation con un lambda muy bajo y realizar Cross-Validation más veces con el valor de lambda cada vez más alto hasta que empeoren los resultados. He realizado esto, y el valor con los que mejores resultados he obtenido es de $\lambda = 0.1$. Para ahorrar tiempo de ejecución, voy a replicar Cross-Validation sólo con $\lambda = 0.1$.

Vamos a dividir los datos de entrenamiento en cinco partes, cada cual será un 20% del total:

```
x1 = X[1:764,]
x2 = X[765:1529,]
x3 = X[1530:2293,]
x4 = X[2294:3058,]
x5 = X[3058:3823,]

y1 = Ymatrix[1:764,]
y2 = Ymatrix[765:1529,]
y3 = Ymatrix[1530:2293,]
y4 = Ymatrix[2294:3058,]
y5 = Ymatrix[3058:3823,]
```

Antes de empezar con el proceso de Cross-Validation, hay que explicar detalladamente cómo usaremos Softmax.

Softmax se puede describir con la regla:

$$Asignar\ x\ a\ C_j\ si\ P(C_j|x) = max_k P(C_k|x),\ k = 1, 2, ...K$$

Explicándolo con lenguaje natural es muy sencillo. Simplemente para cada input aplicamos la función clasificadora estimada para cada número, y como nos devuelve una probabilidad, asignamos a ese imput la

etiqueta cuya función nos haya devuelto mayor probabilidad.

Por ejemplo, si queremos clasificar un input, y obtenemos un 0.7 de probabilidad con la hipótesis que clasifica el 3, un 0.9 de probabilidad con la hipótesis que clasifica el 8, y 0.1 de probabilidad con el resto, etiquetaremos el número como un 8, ya que la probabilidad de que acertemos es la mayor posible.

Estas diez hipótesis las almacenaremos en una matriz.

El proceso de Cross-Validation es el siguiente

```
lambda = 0.1
fallos = 0

for( l in 1:5 ) # Se entrenará 5 veces
{
  if( l == 1 ) # En el primer entrenamiento, el primer subset será de test
  {
    xtest = x1
    ytest = y1
    x = rbind(x2,x3,x4,x5)
    y = rbind(y2,y3,y4,y5)
  }
  if( l == 2 ) # En el segundo entrenamiento, el segundo subset será de test
  {
    xtest = x2
    ytest = y2
    x = rbind(x1,x3,x4,x5)
    y = rbind(y1,y3,y4,y5)
  }
  if( l == 3 ) # En el tercer entrenamiento, el segundo subset será de test
  {
    xtest = x3
    ytest = y3
    x = rbind(x2,x1,x4,x5)
    y = rbind(y2,y1,y4,y5)
  }
  if( l == 4 ) # En el cuarto entrenamiento, el cuarto subset será de test
  {
    xtest = x4
    ytest = y4
    x = rbind(x2,x3,x1,x5)
    y = rbind(y2,y3,y1,y5)
  }
  if( l == 5 ) # En el quinto entrenamiento, el quinto subset será de test
  {
    xtest = x5
    ytest = y5
    x = rbind(x2,x3,x4,x1)
    y = rbind(y2,y3,y4,y1)
  }


  # Matriz de valores de theta. La usaremos para estimar con softmax.
  thetaMatrix = matrix(rep(0,650),ncol=(ncol(X)))
```

```
  # Para cada número del 0 al 9, obtenemos el theta que clasifique estos números adecuadamente
  for( i in 1:10 )
  {
    #Inicializamos valores a 0
    theta = rep(0,ncol(X))

    # Aplicamos el Gradiente descendiente. Aquí es donde usamos la matriz de etiquetas que programamos
    theta = GD(x,y[,i],theta,lambda)

    # Añadimos los valores a la matriz de hipótesis
    thetaMatrix[i,] = theta
  }


  # Softmax
  Ypredicted = rep(0,length(ytest[,1]))
  Yprobs = rep(0,length(ytest[,1]))

  for( i in 1:10 )
  {
    for( j in 1:length(ytest[,i]) )
    {
      if( hx(xtest[j,],thetaMatrix[i,]) > Yprobs[j]  )
      {
        Yprobs[j] = hx(xtest[j,],thetaMatrix[i,])
        Ypredicted[j] = i-1
      }
    }

  }


  # Obtenemos los fallos
  for( i in 1:length(Ypredicted) )
  {
    if( Ypredicted[i] != ytest[i,11] ) fallos = fallos + 1
  }

}

# Fallos obtenidos
fallos = fallos/5

# Error medio obtenido
Eval = fallos/length(ytest[,1])
```

Veamos el Eval obtenido:

```
Eval
```

```
## [1] 0.05013055
```

Como vemos, el error medio obtenido parece bastante bueno.

Entrenamiento final. Obtenemos el modelo de clasificación

Ya sabemos como obtener un modelo de clasificación, y ya tenemos los valores de los hyperparámetros óptimos para obtenerlo. Ahora solo falta entrenar el modelo con toda la base de datos de entrenamiento, y ponerlo a prueba en el test.

```
lambda = 0.1

# Matriz de valores de theta. La usaremos para estimar con softmax.
thetaMatrix = matrix(rep(0,650),ncol=(ncol(X)))


# Para cada número del 0 al 9, obtenemos el theta que clasifique estos números adecuadamente
for( i in 1:10 )
{
  #Inicializamos valores a 0
  theta = rep(0,ncol(X))

  # Aplicamos el Gradiente descendiente. Le pasamos como parámetro toda la base de datos
  theta = GD(X,Y,theta,lambda)

  # Añadimos los valores a la matriz de hipótesis
  thetaMatrix[i,] = theta
}
```

Las hipótesis están listas. Vamos a ponerla a prueba en el test.

Primero, cargamos los datos del test

```
digitosTest <- read.csv("datos/optdigits_tes.csv", header=F, sep=",") # 1. lectura zip del Test

Xtest <- as.matrix(digitosTest[-65])
Xtest = Xtest/16
Xtest <- cbind(rep(1,nrow(Xtest)),Xtest)

Ytest <- as.matrix(digitosTest[,65])
Yorigins = Ytest
```

Ahora vamos a clasificarlos usando softmax:

```
# Softmax
Ypredicted = rep(0,length(Ytest))
Yprobs = rep(0,length(Ytest))

for( i in 1:10 )
{
  for( j in 1:length(Ytest) )
  {
    if( hx(Xtest[j,],thetaMatrix[i,]) > Yprobs[j]  )
    {
      Yprobs[j] = hx(Xtest[j,],thetaMatrix[i,])
      Ypredicted[j] = i-1
    }
  }

}
```

Ya hemos clasificado. Vamos a ver el error de salida

```
# Obtenemos los fallos
fallos = 0
for( i in 1:length(Ypredicted) )
{
  if( Ypredicted[i] != Y[i] ) fallos = fallos + 1
}
Eout = fallos / length(Ytest)
```

Veamos el Eout obtenido:

```
Eout
```

```
## [1] 0.0595025
```

Conclusión

Hemos obtenido unas hipótesis de clasificación, que combinadas con softmax, clasifican bastante bien, teniendo en cuenta que he usado solamente un modelo de regresión logística y gradiente descendiente

He tenido en cuenta que hay alternativas mucho más sofisticadas, como librerías que realizan todo este proceso en segundos, pero he preferido optar por programarlo todo yo mismo ya que aunque es menos óptimo, entiendo bien todo lo que he hecho y me es más fácil de explicar.

Regresión para airfoils

En aerodinámica, un airfoil es el perfil que tiene una superficie cuando se le somete a un esfuerzo, por ejemplo, cuando el viento impacta contra el ala de un avión. Dependiendo de cómo el viento impacte con el ala, ésta tendrá comportamientos diferentes.

En nuestra base de datos, tenemos unos perfiles de pruebas realizadas sobre un ala de avión en un túnel de viento, con diferentes tamaños a diferentes velocidades y distintos ángulos de ataque. El lapso del perfil aerodinámico y la posición del observador fueron los mismos en todos los experimentos.

Todas las columnas, excepto la última, de la base de datos, corresponden a distintos parámetros que se han usado para las mediciones. La última columna corresponde al ruido que se produce en la prueba, en decibelios, que será nuestro output a estimar.

Objetivo de la práctica

Nuestro objetivo es estimar un modelo lineal el cual pueda predecir, a partir de unos parámetros, el ruido que hace un ala de avión en el túnel de viento.

Método a utilizar

Para estimar este modelo lineal, vamos a usar regresión lineal con gradiente descendiente.

Para ello, vamos a suponer que los ejemplos de la base de datos son independientes e idénticamente distribuidos.

Teniendo en cuenta que la base de datos tiene cinco parámetros, nuestra clase de funciones H va a tener el siguiente tipo de funciones $h_\theta(x)$:

$$h_\theta(x) = \theta_1 + \theta_2 x_2 + \theta_3 x_3 + \theta_4 x_4 + \theta_5 x_5 + \theta_6 x_6$$

Donde $x_1 = 1$

En código, la función es así:

```
hx <- function(x,theta)
{
  h = t(theta)%*%x
  h
}
```

Lo que nos interesa es encontrar los valores adecuados de θ para que el error de la estimación sea mínimo. Es decir, debemos minimizar la función de error.

La función de error en regresión lineal (sin regularizar todavía) es la siguiente:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^i)^2$$

Como vemos, la función equivale a la media de la suma de los cuadrados de la diferencia del output real respeto al output estimado.

Para minimizar esta función, usaremos el gradiente descendiente. Para ello necesitamos la derivada parcial de la función de error.

Sin regularizar aún,

$$\frac{\partial}{\partial\theta_j} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^i)x_j^{(i)}$$

Preprocesamiento de datos

Vamos a cargar nuestra base de datos.

```
datos <- read.csv("datos/airfoil_self_noise.csv", header=F, sep=",") # 1. lectura zip del train

X <- as.matrix(datos[-6]) # Inputs, quitamos la columna de outputs
X <- cbind(rep(1,nrow(X)),X) # Añadimos una columna de unos para que podamos tener término independient

Y <- as.matrix(datos[,6]) # Outputs
```

Si echamos un vistazo a los valores de los datos de entrenamiento, vemos que están en escalas demasiado distintas, por lo que el gradiente descendiente tardaría demasiado en converger. Vamos a normalizar estos datos, para que se muevan entre -1 y 1, aproximadamente.

```
for( i in 2:ncol(X) ) # Normalización
{
  X[,i] = (X[,i] - sum(X[,i])/nrow(X))/max(X[,i])
}
```

Vamos a separar estos datos en dos subconjuntos, uno para entrenar, y otro para test. El subconjunto de entrenamiento será el 80% del total, mientras que el subconjunto de test será del 20%.

```
xtrain = X[1:1040,]
xtest = X[1041:1303,]

ytrain = Y[1:1040]
ytest = Y[1041:1303]
```

Regularización

Igual que en el problema anterior, podemos probar a regularizar, a ver si obtenemos mejores resultados.

Vamos a regularizar imponiendo la siguiente condición a θ:

$$\sum_{j=0}^{n} \theta_j^2 \leq C$$

Esta condición incita a que algunos valores de θ se aproximen a 0, simplificando la función $h_\theta(x)$, por lo que se corregiría el sobreajuste.

Si queremos aplicar esta regularización, necesitamos modificar nuestra función de error y su derivada.

La función de error regularizada es la siguiente:

$$J(\theta) = \frac{1}{2m}[\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^i)^2 + \lambda \sum_{j=1}^{n} \theta_j^2]$$

Su derivada, regularizada, es la siguiente función:

$$\frac{\partial}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^i)x_j^{(i)} - \frac{\lambda}{n}\theta_j$$

En el caso del término independiente, θ_1, por convención no se aplica la regularización, por lo que sólo en ese caso se aplicaría la derivada original.

La función de error y su derivada, en código, con la regularización impuesta, son las siguientes.

```
J <- function(x,y,theta,lambda)
{
  cost = 0
  for( i in 1:length(y))
  {
    cost = cost + (hx(x[i,],theta)-y[i])^2
  }

  sumsquares = 0
  for( i in 1:length(theta) )
  {
    sumsquares = sumsquares + theta[i]*theta[i]
  }

  cost = cost + lambda*sumsquares
  cost = cost / (2*length(y))

  cost
}
```

```
dJ <- function(x,y,theta,j,lambda)
{

  cost = 0
  for( i in 1:length(y) )
  {
```

```
    cost = cost + ( hx(x[i,],theta) - y[i] )*x[i,j]
  }
  cost = cost/length(y)


  if( j != 1 ) cost = cost - (lambda*theta[j])/length(theta)



  cost
}
```

¿Qué lambda usar? Cross-Validation

En nuestro caso, haremos Cross-Validation cinco veces, para los siguientes valores de lambda: 0, 0.001, 0.01, 0.1 y 1.

El set de entrenamiento lo dividiremos en cinco partes. Por lo tanto, nuestro set inicial quedaría distribuido de la siguiente forma:

Train CV	Train CV	Train CV	Train CV	Test CV	TEST
16%	16%	16%	16%	16%	20%

```
x1 = xtrain[1:208,]
x2 = xtrain[209:416,]
x3 = xtrain[417:624,]
x4 = xtrain[625:832,]
x5 = xtrain[833:1040,]

y1 = ytrain[1:208]
y2 = ytrain[209:416]
y3 = ytrain[417:624]
y4 = ytrain[625:832]
y5 = ytrain[833:1040]
```

El código para realizar Cross-Validation es el siguiente

```
vector_errores = rep(0,5) # Almacenaremos la media de error obtenida con distintos lambda

for( k in 1:5 ) # 5 Veces Cross-Validation
{
  if( k == 1 ) lambda = 0
  if( k == 2 ) lambda = 0.001
  if( k == 3 ) lambda = 0.01
  if( k == 4 ) lambda = 0.1
  if( k == 5 ) lambda = 1

  error = 0
  for( l in 1:5 ) # Con cada lambda, 5 entrenamientos distintos
  {
    if( l == 1 ) # En el primer entrenamiento, el primer subset será de test
```

```
    {
      xtest = x1
      ytest = y1
      x = rbind(x2,x3,x4,x5)
      y = rbind(y2,y3,y4,y5)
    }
    if( l == 2 ) # En el segundo entrenamiento, el segundo subset será de test
    {
      xtest = x2
      ytest = y2
      x = rbind(x1,x3,x4,x5)
      y = rbind(y1,y3,y4,y5)
    }
    if( l == 3 ) # En el tercer entrenamiento, el tercer subset será de test
    {
      xtest = x3
      ytest = y3
      x = rbind(x2,x1,x4,x5)
      y = rbind(y2,y1,y4,y5)
    }
    if( l == 4 ) # En el cuarto entrenamiento, el cuarto subset será de test
    {
      xtest = x4
      ytest = y4
      x = rbind(x2,x3,x1,x5)
      y = rbind(y2,y3,y1,y5)
    }
    if( l == 5 ) # En el quinto entrenamiento, el quinto subset será de test
    {
      xtest = x5
      ytest = y5
      x = rbind(x2,x3,x4,x1)
      y = rbind(y2,y3,y4,y1)
    }



    theta = rep(0,ncol(xtrain)) # Inicializamos los valores de theta a 0

    # Ejecutamos el gradiente descendiente para encontrar los valores de theta que minimicen la función
    theta = GD(x,y,theta,lambda=0.01,num_iteraciones = 1000)

    # Sumamos el error para hacer la media al acabar el Cross-Validation
    error = error + J(xtest,ytest,theta,lambda)

  }

  # Hacemos la media y la añadimos al vector de errores
  error = error/5
  vector_errores[k] = error

}
```

Veamos el resultado:

```
vector_errores
```

```
## [1] 37.29527 37.33194 37.66197 40.96225 73.96505
```

Como vemos, cuanto más pequeña es lambda, menos error obtenemos. De hecho, el error mínimo lo obtenemos con $\lambda = 0$. Esto significa que no hay nada de overfitting, y si regularizamos obtenemos underfitting.

Aunque hemos conseguido una hipótesis que estima, el error es demasiado grande. Había probado con hipótesis más complejas, como por ejemplo $h_\theta(x) = \theta_0 + \theta_1 x_1 + ... + \theta_5 x_5 + \theta_6 x_1 \check{s} + ... + \theta_{11} x_5 \check{s}$

Sin embargo, aunque mejoraban un poco el error, el tiempo que necesitaban para conseguir los valores era demasiado alto.

Entrenamiento final. Obtenemos el modelo de clasificación

Hemos visto que en este caso, esta regularización en concreto es contraproducente. Por lo tanto, aproximaremos una hipótesis con el hyperparámetro $\lambda = 0$.

```
lambda = 0

theta = rep(0,ncol(xtrain)) # Inicializamos valores a 0

theta = GD(xtrain,ytrain,theta,lambda,num_iteraciones = 5000 ) # Al ser única ejecución podemos subirle
```

Ya tenemos los valores de la hipótesis. Vamos a ponerla a prueba en el test.

```
error = J(xtest,ytest,theta,lambda)
error
```

```
##          [,1]
## [1,] 21.21234
```

Como vemos, el error está acorde a lo esperado. De hecho, incluso ha bajado un poco, seguramente por que hemos entrenado con más iteraciones.

Conclusión

Hemos obtenido un modelo de regresión, que si bien tiene un error un poco alto, se podría bajar con una clase de hipótesis más compleja. El problema de elegir una clase de hipótesis más compleja, es que el tiempo que tardaría en estimar los valores de esa hipótesis sería mucho más alto, lo que hace esta opción inviable. Quizás si pudiese optimizar las operaciones que realiza el gradiente descendiente, esto sería viable.