PRÁCTICA 1 - REGRESIÓN LINEAL

Alumnos: Adrián Ogáyar Sánchez y Arturo Barbero Pérez

PARTE 1: Regresión lineal con una variable

En esta primera parte vamos a aplicar el método de regresión lineal sobre los datos que se encuentran en el fichero *exdata1.csv*, el cual contiene una lista de datos con la relación entre los beneficios de una compañía de distribución de alimentos en una ciudad y la población de dicha ciudad, para así obtener la función de una recta que nos permita aproximar los posibles beneficios de una población dada.

Para hallar esa función suponemos una función hipotética de una recta: $h_0 = \theta_0 + \theta_1 * x$
Así pues nuestro objetivo será hallar θ_0 y θ_1, y para ello deberemos crear un algoritmo que minimice la función de coste $J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (h_0(x_i) - y_i)^2$ mediante el método de descenso de gradiente:

```
def regresionLineal(datos, t0 = 0, t1 = 0, precision = 0.001, a = 0.01):

    #Variables del resultado temporal de t0 y t1
    temp0 = 0
    temp1 = 0

    #Número de vueltas que realiza el bucle
    vueltas = 0

    #Funciones matemáticas propias del algoritmo
    h0 = lambda x : t0 + t1*x
    h1 = lambda x : temp0 + temp1*x
    fun = lambda x : 1/(2*x.size) * sum((h0(x[:,0]) - x[:,1])**2)
    JT = lambda x : 1/(2*x.size) * sum((h1(x[:,0]) - x[:,1])**2)

    #Realizamos un primer descenso de gradiente
    temp0, temp1 = descensoGradiente(a, datos, h0, h1, t0, t1)

    #Comparamos la diferencia entre el coste anterior y el actual
    while abs(fun(datos) - JT(datos)) > precision:
        t0 = temp0
        t1 = temp1

        temp0, temp1 = descensoGradiente(a, datos, h0, h1, t0, t1)
        vueltas += 1

    #Imprimimos por pantalla los resultados
    print(chr(952) + "0: " + str(t0))
    print(chr(952) + "1: " + str(t1))
    print("Vueltas: " + str(vueltas))

    #Realiza la gráfica correspondiente pasándole
    #como argumento los datos y la hipótesis
    graficarDatos(datos, h0)

    return
```

La función *regresionLineal* recibe una matriz de Mx2 que contiene los datos de prueba, además de una serie de parámetros opcionales que son los siguientes:
t0 y **t1**: Los valores iniciales de $\theta_0 y \theta_1$, por defecto son 0, resulta útil si conocemos el valor aproximado de la recta.
precisión: La distancia mínima que tiene que haber entre el coste de función actual y anterior para considerar que ha encontrado el mínimo. Valores muy pequeños harán que la función tarde más en encontrar la recta, pero asegurarán que esta sea más precisa.
a: El valor α que utiliza la función de descenso de gradiente para realizar los saltos al disminuir la función de coste. Un valor muy grande hará que la búsqueda de la recta sea imprecisa (e incluso resulte imposible encontrar el mínimo con la precisión dada), pero uno muy pequeño implicará una búsqueda más lenta.

Esta función ejecuta un bucle ayudándose de la función *descensoGradiente* a través del cual va ajustando el valor de $\theta_0 y \theta_1$, almacenándolas en variables temporales para comprobar si la diferencia entre el coste anterior al bucle y el posterior es lo bastante pequeña como para considerar la convergencia del coste y, por tanto, haber encontrado el mínimo.

La función *descensoGradiente* se encarga de ajustar θ_0 y θ_1 de la siguiente forma:

```
def descensoGradiente(a, datos, h0, h1, t0, t1):
    tmp0 = t0 - a/datos.size * np.sum(h0(datos[:,0]) - datos[:,1])
    tmp1 = t1 - a/datos.size * np.sum((h0(datos[:,0]) - datos[:,1]) * datos[:,0])

    return (tmp0, tmp1)
```

Restando a θ_0 y θ_1 la derivada de $J(\theta)$ multiplicada por α nos aseguramos de obtener un valor de ambas θ que dé lugar a una $J(\theta)$ menor que la anterior (siempre que α sea lo bastante pequeña). *descensoGradiente* devuelve una tupla con los valores temporales de las nuevas θ, que *regresionLineal* recoge para comprobar si los costes han convergido.

Hay que mencionar que esta función asegura encontrar un mínimo, pero no tiene por qué ser un mínimo global. Para asegurarnos de obtener el mínimo global sería necesario realizar el proceso repetidas veces con θ iniciales distintas. Dado que la función de regresión lineal solo dispone de un mínimo global en este caso no existirá ese problema.

Una vez obtenemos θ_0 y θ_1 las mostramos en pantalla junto al número de vueltas necesarias para obtener el resultado:

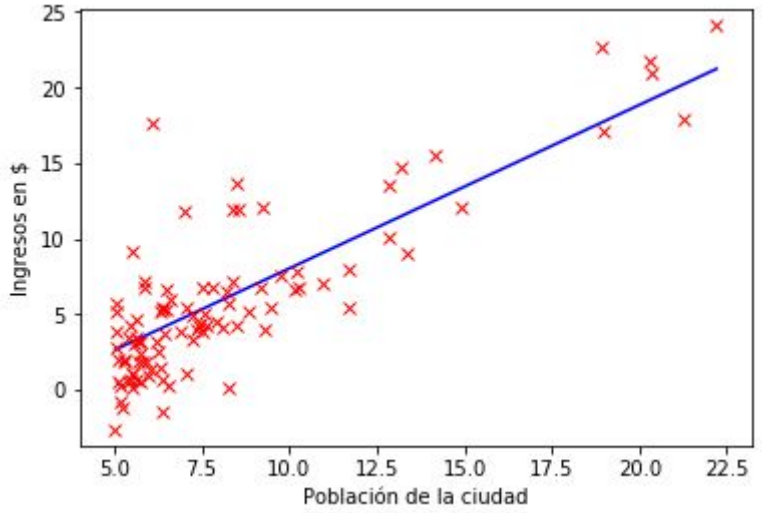
```
θ0: -2.7919004691589815
θ1: 1.082136883499759
Vueltas: 1421
```

Y mostramos las gráficas mediante la función *graficarDatos*:

```
def graficarDatos(datos, h0):

    #Establece unos maximos y minimos para los ejes
    ax = plt.gca()
    ax.axis([datos[:,0].min() - 1, datos[:,0].max() + 1, datos[:,1].min() - 1, datos[:,1].max() + 1])

    #Grafica la recta
    plt.plot([datos[:,0].min(), datos[:,0].max()], [h0(datos[:,0].min()), h0(datos[:,0].max())], '-b')
    '''Logica del plot:
    -El primer array indica lo que debe ocupar el eje X, es decir, del minimo de X (en datos) hasta el maximo
    -El segundo array indica la pendiente de la recta, o dicho de otra forma, donde empieza y acaba la recta
    en el eje Y
    Para hallar esto nos serviremos de la función hip, que es la que nos da la ecuación de la recta.
    Le pasamos como argumento los theta obtenidos y el maximo y minimo X que se puede obtener de los datos y
    creamos un array con esos datos'''

    #Grafica los datos
    plt.plot(datos[:,0], datos[:,1], 'rx')
    plt.xlabel('Población de la ciudad')
    plt.ylabel('Ingresos en $')

    #Grafica del coste con mapa de alturas
    fig = plt.figure()
    ax = fig.gca(projection='3d')

    X = np.arange(-10, 10, 0.01)
    Y = np.arange(-3, 4, 0.01)
    X,Y = np.meshgrid(X,Y)
    Z = coste(datos, X, Y)

    surf = ax.plot_surface(X, Y, Z, cmap = plt.cm.coolwarm, linewidth = 0, antialiased = False)
    plt.xlabel(chr(952)+'0')
    plt.ylabel(chr(952)+'1')

    fig.colorbar(surf, shrink = 0.5)


    #Grafica del coste con mapa de contorno
    plt.figure()
    ax = plt.gca()
    plt.contour(X, Y, Z, colors = 'black')
    ax.contour(X, Y, Z, np.logspace(-2, 3, 20))
    plt.xlabel(chr(952)+'0')
    plt.ylabel(chr(952)+'1')

    plt.show()

    return
```

Esta función recibe los datos y la función de hipótesis (con las θ obtenidas) para así realizar 3 gráficas.

La primera muestra los puntos de los datos de prueba y la recta de regresión lineal.



Como se puede apreciar en la gráfica los ingresos esperados aumentan según la población de la ciudad (en miles) siguiendo la forma de los datos de prueba ofrecidos.

A partir de esta gráfica podríamos predecir que, por ejemplo, en una población de 20.000 habitantes obtendríamos unos beneficios aproximados de 17.500$.

Las segunda y tercera gráficas consisten en una representación del coste de $J(\theta)$ respecto a una serie de valores θ_0 y θ_1 en un intervalo.

Para esto es necesario calcular la función $J(\theta)$ para cada uno de los valores θ_0 y θ_1, obteniendo así una matriz de 2 dimensiones:

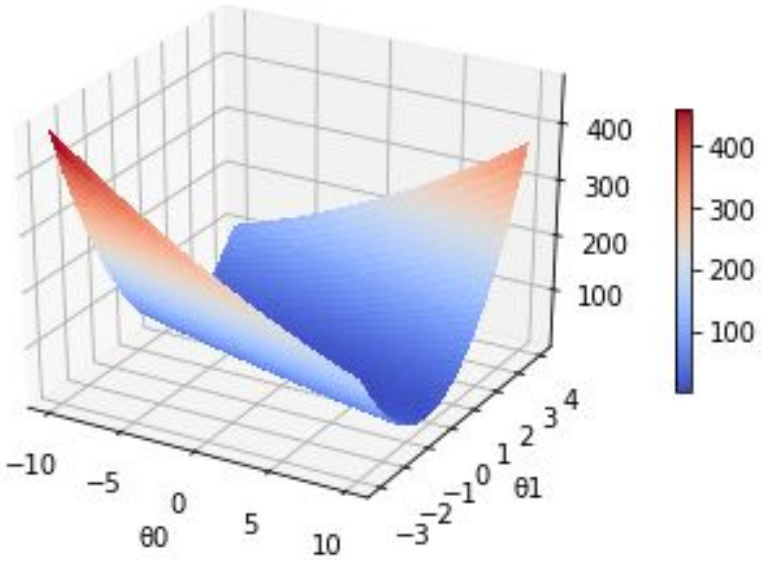
```
def coste(datos, t0, t1):
    #Giramos los vectores de datos para que que se coloquen
    #en una tercera dimension
    x = datos[:, 0, np.newaxis, np.newaxis]
    y = datos[:, 1, np.newaxis, np.newaxis]

    h0 = lambda x, t0, t1 : t0 + t1*x

    R = lambda t0, t1 : 1/(2*datos.size) * np.sum((h0(x, t0, t1) - y)**2, 0)
    v = R(t0,t1)

    return v
```

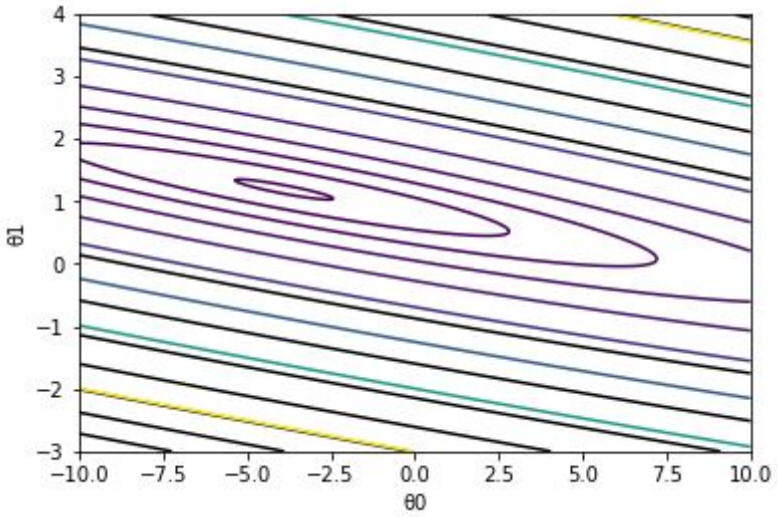
La función *coste* se encarga de realizar el cálculo de esta matriz recibiendo la matriz de datos, otra matriz de θ_0 y otra con los θ_1, estas 2 últimas siendo las matrices obtenidas a partir de la función np.meshgrid($\overline{\theta}_{0,}\ \overline{\theta}_1$)

Antes de realizar el cálculo es necesario hacer que los vectores de datos se encuentren de forma perpendicular a la matriz θ_1 de modo que al realizar $h_0(x, \theta_0, \theta_1)$ obtengamos una matriz cúbica que contenga los resultados de la hipótesis para cada una de las combinaciones de esas 3 variables, con una “base” formada por θ_0 y θ_1 y una “altura” con los valores de x.

A esta matriz se le resta el vector y, también perpendicular a θ y se eleva al cuadrado, obteniendo así una matriz con $(h_0(x, \theta_0, \theta_1) - y)^2$, de modo que solo falta realizar un sumatoria “en la altura”, que serían los resultados del vector de datos, recibiendo así una matriz de 2 dimensiones que representa el valor de $J(\theta)$ para cada θ_0 y θ_1.

La gráficas obtenidas con $\theta_0 \varepsilon[-10, 10]\, y\, \theta_1 \varepsilon[-3, 4]$ son las siguientes:





Como se puede ver en la primera gráfica los valores mínimos se encuentran en $\theta_1 \varepsilon[-1, 1]$, pero resulta difícil conocer el valor de θ_0 pues en ese intervalo de θ_1 los costes son muy similares.

Para eso tenemos la segunda gráfica, la cual nos muestra las altura en un plano, en el cual se puede apreciar que en $\theta_0 \varepsilon[-5, -2.5]$ y $\theta_1 \varepsilon[1, 2]$ tenemos el conjunto de valores mínimos de la función.

Esto puede resultarnos útil para comprobar si θ ha sido calculada correctamente así como conocer las zonas que contienen los puntos mínimos, pudiendo colocar valores iniciales de θ en ellos para así encontrar el mínimo global.

Finalmente, para leer el archivo con los datos de prueba utilizamos la función *lee_csv* y ejecutamos la función *regresionLineal* con las siguientes constantes:

```
def lee_csv(file_name):
    valores = read_csv(file_name, header=None).values

    return valores.astype(float)


#-----------------------------------------------------------------------------#
#-------------------------      CONSTANTES      ------------------------------#
#-----------------------------------------------------------------------------#
theta0 = 0
theta1 = 0
alfa = 0.01
precision = 0.0001


#-----------------------------------------------------------------------------#
#------------------------- LLAMADA A LA FUNCIÓN ------------------------------#
#-----------------------------------------------------------------------------#
#Obtenemos los datos del fichero indicado por parametro
datos = lee_csv("ex1data1.csv")

#Calculamos el resultado.
regresionLineal(datos, theta0, theta1, precision, alfa)
```

PARTE 2: Regresión lineal con varias variables

El objetivo es aplicar el método de regresión lineal a los datos de un archivo dado (*ex1data2.csv*) que contiene datos sobre el precio de casas vendidas en Portland, Oregon. Estos datos incluyen: el tamaño en pies cuadrados, el número de habitaciones y el precio final de la casa en cuestión.
En esta parte se han realizado dos métodos distintos para obtener el parámetro θ gracias al cuál podremos predecir nuestros resultados. Estos métodos son el *descenso de gradiente* y la *ecuación normal.*

Para empezar declaramos una serie de constantes para que la estructura del código sea más clara y limpia.

```
#-------------------------------------------------------------------------#
#-------------------------      CONSTANTES      --------------------------#
#-------------------------------------------------------------------------#

# Obtenemos los datos del fichero indicado por parametro
datos = lee_csv("ex1data2.csv")

# Entradas Xi de los datos, sin contar el precio
x = datos[:, 0:-1]

# Vector de precios
y = datos[:, -1:]

# Normalizamos las x
x_norm, mu, sigma = normalizar(x)

# Inicializamos las thetas a 0
theta = np.zeros((x_norm[0].size + 1, 1))

# Valor usado para calcular el descenso
alfa = 0.01

# Distancia mínima entre los costes calculados
precision = 0.01
```

Mediante la función *lee_csv* obtenemos los datos del fichero especificado por parámetro. En su implementación, los valores los obtenemos en forma de diccionario. Por tanto, los devolvemos en forma de vector para manejarlos de mejor manera en el futuro. Para poder leer estos datos es necesario que importemos *read_csv* procedente de *pandas.io.parsers.*

```
#Lee los datos del fichero csv
def lee_csv(file_name):
    valores = read_csv(file_name, header=None).values

    return valores.astype(float)
```

Acto seguido, guardamos en unas variables *x* e *y* los datos de entrada y los precios de cada casa respectivamente. Llamamos a la función *normalizar* la cual nos va a permitir normalizar los datos de entrada para la versión en la que usamos el descenso de gradiente. Además, obtenemos μ (un vector con la media de cada atributo) y S (un vector con la desviación estándar de cada atributo). Estos vectores son necesarios para hacer predicciones sobre otros ejemplos nuevos de entrada.

```
#Función que normaliza los datos de entrenamiento
def normalizar(x):

    mu = x.mean(axis=0)
    s = x.std(axis=0)

    x_norm = (x - mu) / s

    return (x_norm, mu, s)
```

Una vez hemos obtenido estos vectores, pasamos a crear el vector de θ inicializado con 0's, a declarar la variable α y a designar una *precisión*, la cual se va a tratar de la variación que tiene que haber entre el resultado de la función de coste con θ actual y el resultado de la función de coste con θ anterior, según la cual daremos por terminado el bucle que hay en el descenso de gradiente.

Con estas constantes creadas, podemos pasar a llamar la función de regresión lineal.

```
#-------------------------------------------------------------------------#
#--------------------      LLAMADA A LA FUNCIÓN      --------------------#
#-------------------------------------------------------------------------#


#Opcion de ejecución:
opcion = 0
nuevoDato1 = 1650
nuevoDato2 = 3


if(opcion == 0):
    #Calculamos el resultado mediante descenso gradiente
    print("\nVersion con descenso gradiente")
    print("-------------------------------")
    theta = regresionLineal(alfa, x_norm, y, theta, precision, opcion)
    print("El precio esperado es: " + nuevaPrediccion(nuevoDato1, nuevoDato2, opcion) + "\n")
else:
    #Calculamos el resultado mediante ecuacion normal
    print("\nVersion con ecuacion normal")
    print("-------------------------------")
    theta = regresionLineal(alfa, x, y, theta, precision, opcion)
    print("El precio esperado es: " + nuevaPrediccion(nuevoDato1, nuevoDato2, opcion) + "\n")
```

Mediante la variable *opción* podemos cambiar entre qué versión vamos a utilizar. Si *opción* contiene un 0, se ejecutará la versión que implementa el descenso de gradiente, mientras que si tiene un 1, se ejecutará la que implementa la ecuación normal.

Como vemos en el código, dependiendo de si es una versión u otra, los datos de entrada que recibe la función *regresionLineal* varían entre estar normalizados y no estarlos, ya que en la ecuación normal se necesita que estos datos no estén normalizados.

```
#Algoritmo de aprendizaje
def regresionLineal(alfa, x, y, theta, precision, opcion):

    #Vector de 1's
    p = np.ones( [len(x) , 1], int)

    #Matriz de x_norm con el 1 añadido en el elemento x0
    x = np.hstack([p, x])

    #Variables del resultado temporal de theta
    temp0 = np.zeros((x[0].size, 1))

    if(opcion == 0):
        theta = descensoGradiente(alfa, x, y, theta, temp0)
    else:
        theta = ecuacionNormal(x, y)


    return theta
```

Estén los datos normalizados o no, tenemos que crear un vector de 1's y añadirlo a la matriz de datos como si fuera el elemento x_0 . De esta manera nos aseguramos que la función de la hipótesis la podremos realizar como el producto de dos matrices: X y θ.
Cada versión devuelve un vector θ, el cual nos va a permitir calcular más tarde nuestra predicción, junto con los vectores $\boldsymbol{\mu}$, S, y nuestra función hipótesis.

DESCENSO DE GRADIENTE

Centrándonos en el método de descenso de gradiente, podemos ver que se declaran las funciones lambda necesarias para el cálculo de este.
Fun es la función de coste para el parámetro θ actual, mientras que *JT* es la función de coste para el parámetro θ temporal. Estas dos funciones implementan el siguiente cálculo entre vectores:

$$J(\theta) = \frac{1}{2m}(X\theta - y)^T \cdot (X\theta - y)$$

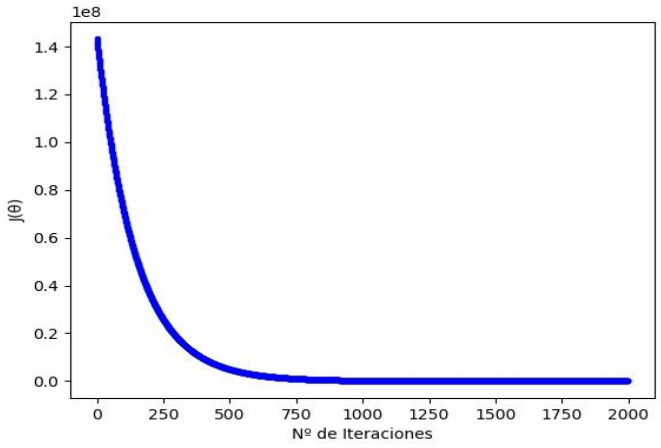
Donde *y* es el vector que contiene los precios, y *Xθ* es el equivalente a nuestra función hipótesis dada de la siguiente manera:

$$h_\theta(x) = X\theta$$

```
def descensoGradiente(a, x, y, theta, temp0):

    #Funciones matemáticas
    h0 = lambda x : x.dot(theta)
    h1 = lambda x : x.dot(temp0)
    #Funciones de coste
    fun = lambda x : 1/(2*x.size) *  ( (h0(x) - y).T).dot( h0(x) - y )
    JT = lambda x : 1/(2*x.size) *  ( (h1(x) - y).T).dot( h1(x) - y )

    #Realizamos un primer descenso de gradiente
    temp0 = calculaDescenso(alfa, x, y, theta, h0)

    vueltas = 0

    #Comparamos la diferencia entre el coste anterior y el actual
    r = abs(fun(x) - JT(x))
    while (r.all() > precision):
        theta = temp0
        temp0 = calculaDescenso(alfa, x, y, theta, h0)
        r = abs(fun(x) - JT(x))

        plt.plot(vueltas, r, 'b.')

        vueltas += 1


    #Imprimimos por pantalla los resultados
    plt.xlabel("Nº de Iteraciones")
    plt.ylabel("J("+chr(952)+")")
    plt.show()

    return theta
```

En este método se va calculando el descenso, mediante un bucle, almacenando los valores de θ en una variable temporal y comprobando en cada iteración si el coste va disminuyendo. Además se va realizando la gráfica, a modo de depuración, para comprobar que esto es cierto.

Ejemplo para un número máximo de 2000 iteraciones



En el método *calculaDescenso* nos encontramos con la expresión que permite ir ajustando los valores de θ para encontrar un mínimo. Matemáticamente la expresión que calcula esto la podemos expresar de la siguiente manera: $\theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}{}_j) \cdot x^{(i)}{}_j$

```
#Función que aplica el descenso de gradiente
def calculaDescenso(a, x, y, theta, h0):
    return (theta - a/(x.size) * ((h0(x) - y).T).dot(x).T)
```

Ante la complejidad de poder representar esto en forma de código, se ha optado por implementar una versión equivalente en forma de matrices, que nos permita hacerlo más eficiente y de una manera más clara:

$$\theta_j := \theta_j - \frac{\alpha}{m} ((X\theta - y)^T \cdot X)^T$$

ECUACIÓN NORMAL

Si nos centramos en el método de la ecuación normal, podemos decir es un método más sencillo de aplicar, ya que mediante unos cálculos matriciales devuelve el valor óptimo para θ.

$$\theta = (X^T X)^{-1} \cdot X^T y$$

Gracias a este método, el cual es más eficiente, no necesitamos elegir un α, y además no se necesita iterar en comparación con el método del *descenso de gradiente,* sin embargo, el coste con este algoritmo puede llegar a ser $O(n^3)$, cuando n es un número muy grande, siendo n el número de ejemplos de entrenamiento. En cambio, el método del descenso de gradiente tiene el mismo coste para todo n

```
#Devuelve en un solo paso el valor óptimo para theta
def ecuacionNormal(x, y):
    return ( (inv((x.T).dot(x))).dot((x.T).dot(y)) )
```

Una vez se ha ejecutado uno de estos dos métodos, si volvemos a la parte del código donde se llama a *regresionLineal*, veremos que se hace una llamada a *nuevaPrediccion.* Esta función se encarga de predecir en este caso un precio, en base a las características de la casa que recibe como parámetros.

```
#Predice un nuevo resultado
def nuevaPrediccion(dato0, dato1, opcion):

    #Función hipotesis
    h0 = lambda x : x.dot(theta)

    x_nuevo = np.array([[dato0, dato1]])

    if(opcion == 0):
        #Normalizamos los datos
        x_nuevo = (x_nuevo - mu) / sigma

    p = np.ones( [len(x_nuevo) , 1], int)
    x_nuevo = np.hstack([p, x_nuevo])


    return str(h0(x_nuevo)[0,0])
```

Esta función además recibe la opción que queremos ejecutar, y con este dato decide si normalizar X o no, ya que en el método de la ecuación normal si se normalizan los datos, el resultado no será correcto.
La predicción se realiza mediante la hipótesis que hemos obtenido, esta se encarga de dar una salida en base a la entrada X y al parámetro θ previamente calculado.

Por último, si ejecutamos las dos versiones aplicándolo al ejemplo de una casa de 1650 pies cuadrados y 3 habitaciones nos damos cuenta de que el precio predecido es el mismo.

```
Version con descenso gradiente
------------------------------
El precio esperado es: 293081.47020516533

Terminated

Version con ecuacion normal
------------------------------
El precio esperado es: 293081.4643348954

Terminated
```