1 - Regresion Lineal

January 27, 2019

1 Parte 1 : Regresión lineal con una variable

Autores: Alberto Pastor Moreno e Iván Fernández Mena

1.1 Carga de datos

En la práctica uno se va a trabajar la regresion lineal con una variable a partir de los datos almacenados en un archivo csv que se nos proporciona. Los datos extraidos estan separado en dos columnas y representan los beneficios de una compañia de distribución de comida en distintas ciudades en base a su población.

A continuación se importaran todas las librerias necesarias para hacer esta practica, tanto para leer el csv como numpy para el soporte de vectores y matrices.

```
In [1]: from pandas.io.parsers import read_csv
        import numpy as np
```

Definimos un método que lee y carga los datos de un csv y retorna esta información recopilada. Se dicta que en el archivo no hay información de cabecera (header = None) y que los valores que son devueltos son de tipo flotantes.

```
In [2]: def load_csv(filename):
            values = read_csv(filename, header=None).values
            return values.astype(float)

In [3]: dataset = load_csv('./datasets/ex1data1.csv')
```

Utilizamos el conjunto de datos obtenido del csv con la función especificada previamente y separamos cada columna en vectores diferentes para poder gestionarlos de manera independiente, de este modo de puede usar cada columna para su estudio.

```
In [4]: independent_data = dataset[:,0]
        dependent_data = dataset[:,1]
```

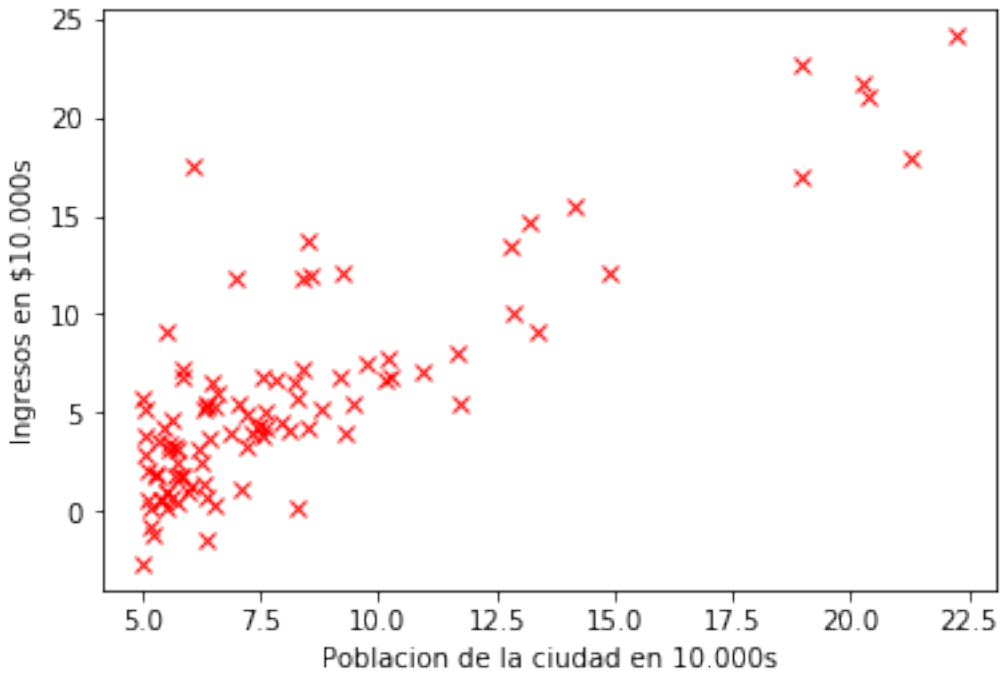
1.2 Visualización simple de dataset

A continuación se generará un plotter con los datos que previamente tratados. Se creará un plotter de cruces enfrentando los valores de beneficios de la empresa estudiada y la población que se nos ha proporcionado en el dataset.

```
In [5]: import matplotlib.pyplot as plt
```

```
In [6]: plt.figure()
        plt.plot(independent_data, dependent_data, 'rx')
        plt.ylabel('Ingresos en $10.000s')
        plt.xlabel('Poblacion de la ciudad en 10.000s')
```

```
Out[6]: Text(0.5,0,'Poblacion de la ciudad en 10.000s')
```



Como se observa en el plotter generado, ya es posible hacer un estudio previo sobre los datos, llegando a la conclusión inicial en que en la mayoría de los datos, según aumente el número de ingresos también aumenta la población de la ciudad. Se puede hacer la hipótesis asimilando que el modelo válido sea la regresión lineal.

1.3 Método de descenso gradiente

Para encontrar la regresión lineal que más se adapte a nuestro dataset, es necesario realizar operaciones básicas de modo que en cada una de las iteraciones se obtenga un resultado más cercano al óptimo. Se necesita definir la función de coste y la función de gradiente de descenso que nos permitira obtener este resultado.

Siempre se tiene que tener en cuenta la función que se usará como hipótesis, en nuestro caso se trata de una unica variable y de una recta.

1.3.1 Implementación de la función de hipótesis de la regresión lineal

```
In [7]: def hyphotesis_function(th0, th1, x):
            return th0 + th1*x
```

1.3.2 Implementacion de la función de coste de la regresion lineal J(θ_0,θ_1)

```
In [8]: def cost_function(fun, th0, th1, m, x, y):
            sum_cost = 0
            for i in range(0, m):
                sum_cost += (fun(th0, th1, x[i]) - y[i])**2
            cost = sum_cost / (2*m)
            return cost
```

1.3.3 Implementación de función *gradient descent*

```
In [10]: def gradient_descent(fun, th0, th1, m, x, y, lr=0.01, epochs=1500):
             cost = []
             vc_th0 = []
             vc_th1 = []
             curr_th0 = th0
             curr_th1 = th1
             for i in range(0, epochs):
                 new_th0 = curr_th0 - (lr/m)*np.sum([fun(curr_th0, curr_th1, x[j]) - y[j]
                                                     for j in range(0, m)])
                 new_th1 = curr_th1 - (lr/m)*np.sum([((fun(curr_th0, curr_th1, x[j]) - y[j])*x
                                                     for j in range(0, m)])
                 curr_th0 = new_th0
                 curr_th1 = new_th1
                 epoch_cost = cost_function(fun, curr_th0, curr_th1, m, x, y)
                 cost += [epoch_cost]
                 vc_th0 += [curr_th0]
                 vc_th1 += [curr_th1]
             return curr_th0, curr_th1, cost, vc_th0, vc_th1
```

A continución se muestran las variables que se han usado para la gestión de la práctica, de este modo se pueden cambiar paramentros de las pruebas de forma comoda.

```
In [11]: th0 = 0
         th1 = 0
         lr = 0.01
         m = len(dataset)
```

1.4 Resultados obtenidos del estudio

Se aplican las funciones definidas previamente y se muestran los resultados para poder tomar unas conclusiones concretas. Se ejecuta la función de descenso del gradiente en nuestra función de hipótesis, además de toda la información necesaria gestionada y obtenida de nuestro conjunto de datos.

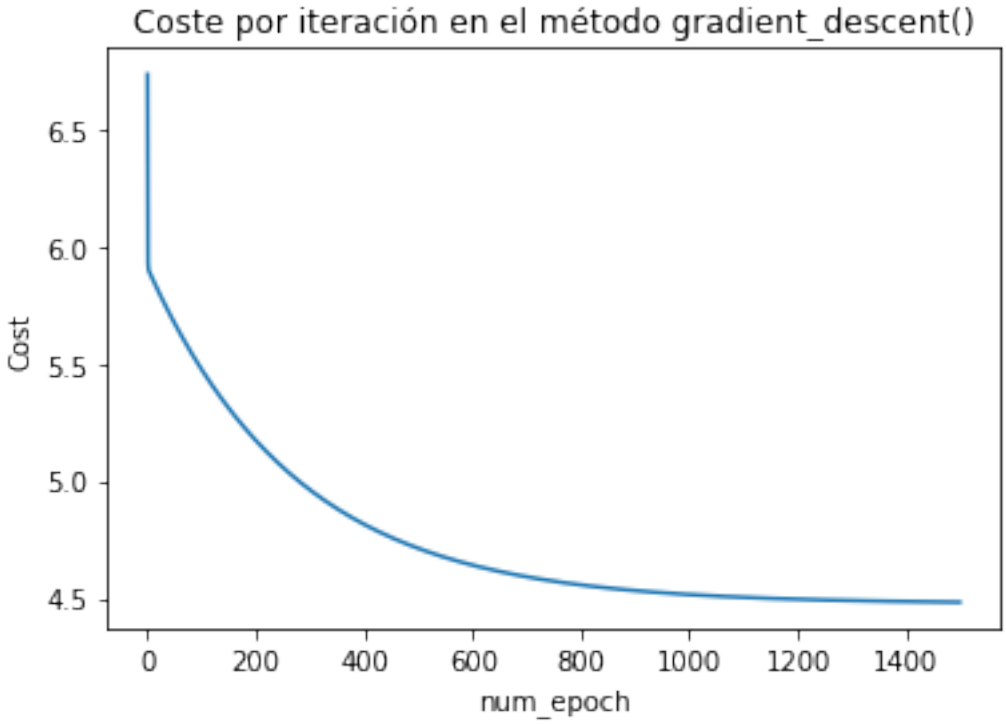
```
In [12]: gd_th0, gd_th1, gd_cost, vc_gd_th0, vc_gd_th1 = gradient_descent(hyphotesis_function,
                                                                           th0, th1, m, indepen
                                                                           dependent_data, lr)
         print ('th0:{}, th1:{}, cost:{}'.format(gd_th0, gd_th1, gd_cost[-1]))
```

```
th0:-3.6302914394043606, th1:1.166362350335582, cost:4.483388256587728
```

Como se estudia en los datos mostrados previamente y en la gráfica que se genera posteriormente, se observa que nuestro valor de coste va disminuyendo según se aplica el descenso de gradiente. Es importante que este coste baje de manera constante, una cantidad mayor al principio y después se estabilice. Si esta acción no se esta realizando es síntoma de que alguna acción esta mal.

```
In [13]: plt.plot(gd_cost)
         plt.title('Coste por iteración en el método gradient_descent()')
         plt.ylabel('Cost')
         plt.xlabel('num_epoch')
```

```
Out[13]: Text(0.5,0,'num_epoch')
```

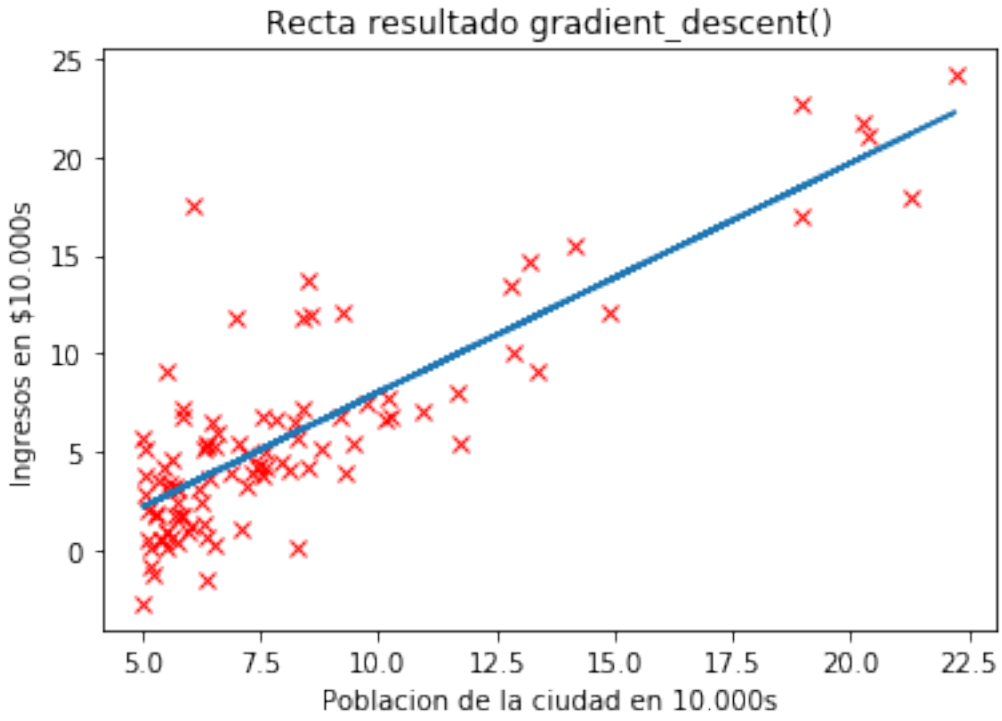


1.4.1 Visualización en plotter normal

Finalmente se muestra el resultado de manera coherente enfrentando los valores del dataset original y los datos obtenidos de ejecutar nuestra función gradiente sobre esos datos.

```
In [14]: plt.plot(independent_data, dependent_data, 'rx',
                  independent_data, hyphotesis_function(gd_th0, gd_th1, independent_data))
         plt.title('Recta resultado gradient_descent()')
         plt.ylabel('Ingresos en $10.000s')
         plt.xlabel('Poblacion de la ciudad en 10.000s')
```

```
Out[14]: Text(0.5,0,'Poblacion de la ciudad en 10.000s')
```



El resultado es una recta que se encuentra ajustada lo máximo posible para nuestro conjunto de datos, esta recta tiene la separación mínima entre los puntos y la resta, obtenido gracias a la función descenso de gradiente.

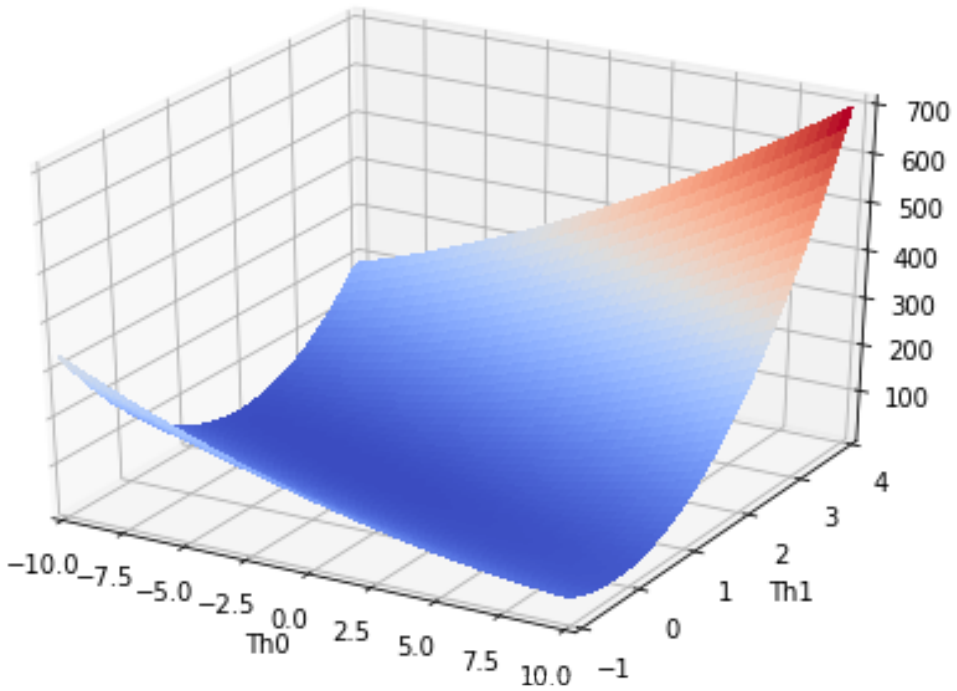
1.4.2 Visualización en plotters 3D

A continuación se monstrará el resultado obtenido en gráficas con mas de dos dimensiones. Estas gráficas son importantes para el análisis del descenso del gradiente de forma visual y para el estudio de aprendizajes mas complejos.

```
In [19]: from mpl_toolkits.mplot3d import Axes3D
         from matplotlib.pyplot import cm
         from matplotlib.ticker import LinearLocator, FormatStrFormatter

In [20]: fig = plt.figure()
         ax = Axes3D(fig)

         X, Y = np.meshgrid(np.arange(-10, 10, 0.1), np.arange(-1, 4, 0.1))
         Z = cost_function(hyphotesis_function, X, Y, m, independent_data, dependent_data)

         ax.plot_surface(X, Y, Z, cmap=cm.coolwarm, linewidth=0, antialiased=False)
         ax.set_xlim(-10, 10)
         ax.set_ylim(-1, 4)
```

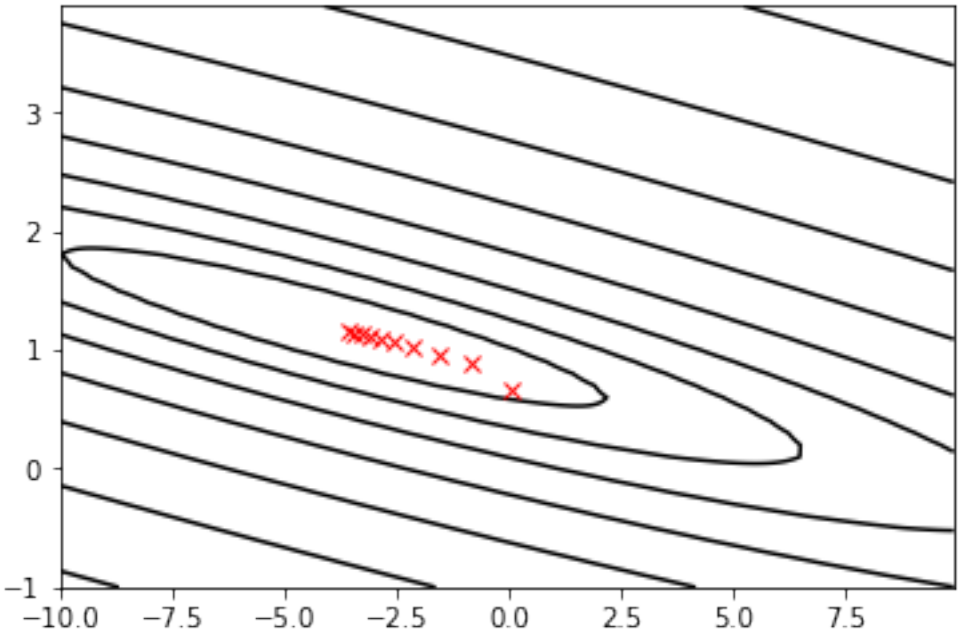
```
        plt.xlabel('Th0')
        plt.ylabel('Th1')
```

```
Out[20]: Text(0.5,0,'Th1')
```




```
In [22]: plt.contour(X, Y, Z ,np.logspace(-2, 3, 20), colors='black')
         plt.plot(vc_gd_th0[::150], vc_gd_th1[::150], 'rx')
```

```
Out[22]: [<matplotlib.lines.Line2D at 0x7fe6760d84a8>]
```



2 Parte 2 : Regresión lineal con dos variables

2.1 Carga y normalización datos

En esta sección, realizaremos regresión lineal con múltiples variables utilizando un dataset diferente al utilizado en la sección anterior. Por ello, realizamos una nueva carga.

```
In [23]: dataset = load_csv('./datasets/ex1data2.csv')
         nfeatures = 2
         dataset[:5]

Out[23]: array([[2.104e+03, 3.000e+00, 3.999e+05],
                [1.600e+03, 3.000e+00, 3.299e+05],
                [2.400e+03, 3.000e+00, 3.690e+05],
                [1.416e+03, 2.000e+00, 2.320e+05],
                [3.000e+03, 4.000e+00, 5.399e+05]])

In [24]: features = dataset[:,:nfeatures] # (47,2)
         target = dataset[:,nfeatures] # (47,)
         features.shape

Out[24]: (47, 2)
```

Normalizamos los datos los datos de las variables características ya que las unidades utilizadas en el dataset utilizado varían en función al atributo en cuestión. Esto podría generar problemas por lo que sustituimos cada valor de X por su división entre su diferencia con la media de su

columna y la desviación estándar de su columna, teniendo en cuenta que cada columna se corresponde con una variable característica en cuestión.

Esto queda expresado por:

$$x_i \leftarrow \frac{x_i - \mu}{\sigma_i}$$

```
In [25]: def normalice(x):
             mu = [np.mean(x[:,i]) for i in range(0,len(x[0]))]
             sigma = [np.std(x[:,i]) for i in range(0,len(x[0]))]
             xnorm = (x - mu)/sigma
             return xnorm, mu, sigma
```

```
In [26]: features_norm, mu, sigma = normalice(features)
         ones = np.ones([len(features),len(features[0])+1])
         ones[:,1:] = features_norm
         features_norm = ones
         features_norm[:3]
```

```
Out[26]: array([[ 1.        ,  0.13141542, -0.22609337],
                [ 1.        , -0.5096407 , -0.22609337],
                [ 1.        ,  0.5079087 , -0.22609337]])
```

```
In [27]: th = [1,1,1]
         lr = 0.01
         m = len(dataset)
```

2.2 Función hipótesis

Debido a que tenemos más de una variable característica, en este caso utilizaremos una función definida de la siguiente forma:

$$h_\theta(x) = \theta^T x$$

```
In [28]: hyphotesis_function = lambda th,x : np.dot(x, th)
```

```
In [29]: hyphotesis_function(th, features_norm[0,:])
```

```
Out[29]: 0.9053220544433592
```

2.3 Función coste

La función de coste (debajo) queda definida por la siguiente expresión:

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

```
In [30]: def cost_function(fun, th, m, x, y):
             cost = (x.dot(th) - y).T.dot(x.dot(th) - y) / (2*m)
             return cost
```

```
In [31]: cost_function(hyphotesis_function, th, m, features_norm, target)
```

```
Out[31]: 65591047222.902596
```

2.4 *Gradient descent*

La función *gradient descent* (debajo) actualiza los valores de θ_j *epochs* veces siguiendo la siguiente expresión:

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

```
In [32]: def gradient_descent(fun, th, m, x, y, lr=0.01, epochs=1500):
             cost = []
             curr_th = th
             for e in range(0, epochs):
                 curr_th = curr_th - (1/m)*lr*(np.dot(x.T, fun(curr_th, x) - y))
                 epoch_cost = cost_function(fun, curr_th, m, x, y)
                 cost += [epoch_cost]
                 #print('It: {}, Cost: {}'.format(e + 1, epoch_cost))
             return curr_th, cost
```

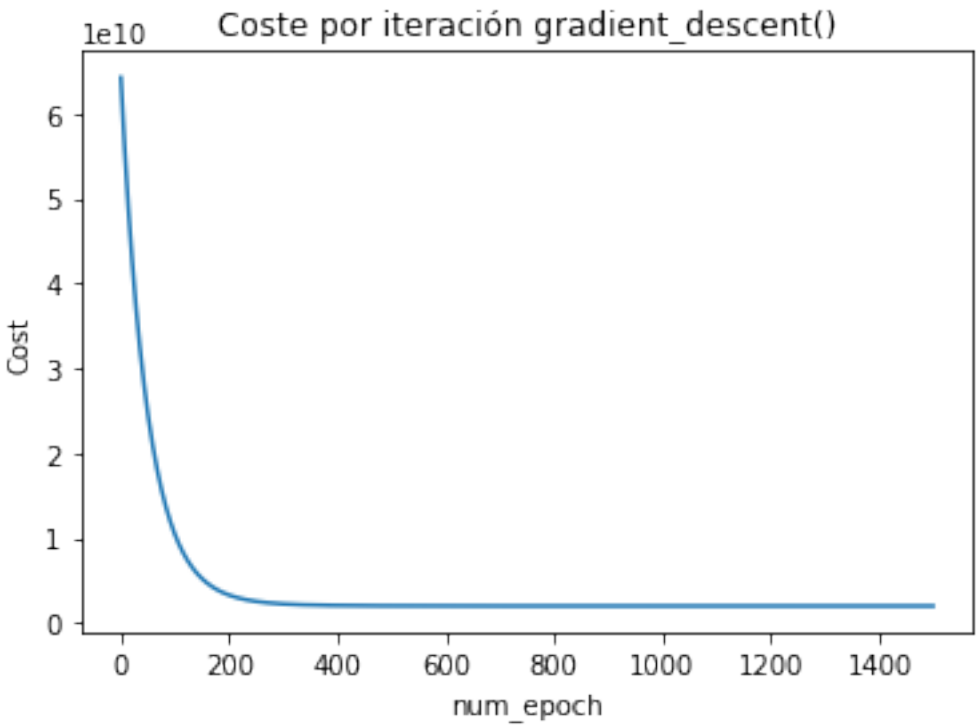
```
In [33]: gd_th, gd_cost = gradient_descent(hyphotesis_function, th,
                                           m, features_norm, target, lr=0.01, epochs=1500)
         gd_th
```

```
Out[33]: array([340412.56301468, 109370.05670466,  -6500.61509507])
```

Mostramos la grafica donde se ve la disminucion de costes:

```
In [34]: plt.figure()
         plt.plot(gd_cost)
         plt.title('Coste por iteración gradient_descent()')
         plt.ylabel('Cost')
         plt.xlabel('num_epoch')
```

```
Out[34]: Text(0.5,0,'num_epoch')
```



2.5 *Normal Equation*

De forma alternativa a *gradient descent*, planteamos la expresión de la ecuación normal que resuelve los valores de θ siguiendo la siguiente expresión:

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

```
In [35]: ones = np.ones([len(features),len(features[0])+1])
         ones[:,1:] = features
         features_wones = ones
         features_t = np.transpose(features_wones)
         th_norm_eq = np.dot(np.dot(np.linalg.pinv(features_t.dot(features_wones)), features_t
         print (th_norm_eq)
```

```
[89597.90954361   139.21067402 -8738.01911255]
```

2.5.1 Resultados

Para comprobar que el modelo implementado utilizando *gradient descent* es correcto, realizamos una predicción con los valores obtenidos de θ con nuestra implementación y los valores obtenidos con la ecuación normal

```
In [37]: x_test = [1, 1650, 3]
         x_test[1:] = (np.array(x_test[1:]) - mu ) / sigma
         print('Predicción gradient descent:{}, predicción ecuación normal:{}'.
               format(hyphotesis_function(gd_th, x_test),
                      hyphotesis_function(th_norm_eq, [1, 1650, 3])))
```

```
Predicción gradient descent:293098.4666760489, predicción ecuación normal:293081.4643349892
```