Regresión Lineal con Python

José Luis Higuera Caraveo

Febrero 25 2022

1 Regresión Lineal

1.1 Regresión Lineal Simple

1.1.1 Hipótesis de la Regresión Lineal

Los problemas de regresión son aquellos que, dado una variable de comportamiento (x) podemos predecir el resultado de una segunda (y).

La regresión Lineal simple se traduce en un sistema lineal donde la hipótesis es:

$$Hiptesis : h_0(x) = \theta_0 + \theta_1 \cdot x$$

donde:

- $h_0(x)$: Es la predicción, es decir la variable y.
- θ_0 y θ_1 : Son los parámetros que el algoritmo trata de encontrar tales que el error de la predicción sea lo mínimo posible.

1.1.2 Función de coste

El proceso del algoritmo es, se inicializan los parámetros θ_0 y θ_1 de manera aleatoria. Por ejemplo:

$$Parmetros = [0, 0]$$

El algoritmo procederá a hacer la primera predicción. Para cada valor de la variable x, se predice una variable y, para diferenciar, llamaremos a esta variable y como y_{pred}.

Una vez se tienen todas las predicciones se procede en calcular el costo. La fórmula para calcularlo es la siguiente:

$$J(\theta_0,\ \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_0(x^i) - y^i)^2$$

donde: * m: Es el número de muestras que se están analizando, o el número de ejemplos. * $h_0(x^i)$: Es el resultado de la predicción calculada anteriormente, es decir, el valor de la variable y_{pred}.

El objetivo final del algoritmo es minimizar $J(\theta_0,\ \theta_1)$ es decir, encontrar los θ_0 y θ_1 óptimos que minimicen el costo o también llamado pérdida.

Hasta este momento podemos resumir que tenemos:

- Hipótesis: $h_0(x) = \theta_0 + \theta_1 \cdot x$.
- Parámetros: $\theta_0,\ \theta_1$
- Función de costo: $J(\theta_0,\ \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_0(x^i) - y^i)^2$
- Objetivo: $min\ J(\theta_0,\ \theta_1)$

1.1.3 Descenso del Gradiente

El descenso del gradiente es un algoritmo de optimización que nos ayuda a encontrar el costo mínimo de un problema de regresión.

Para encontrar el mínimo en una ecuación se usa el cálculo multivariable, se aplica las derivadas parciales a la función de coste, respecto a cada parámetro $\theta_0\ y\ \theta_1$. A su vez, se establece una tasa de aprendizaje que ayuda a estar re calculando los parámetros hasta que este converja a un mínimo. En este sentido el algoritmo se expresa de la siguiente manera.

Repetir hasta converger {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j}\ J(\theta_0,\ \theta_1)$$

$$(Para\ j = 1\ y\ j = 0)$$

}

donde: * α es la tasa de aprendizaje.

Derivadas parciales para $\theta_0\ y\ \theta_1$

$$j = 0\ :\ \frac{\partial}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^{m} (h_0(x^i) - y^i)$$

$$j = 1\ :\ \frac{\partial}{\partial \theta_1} = \frac{1}{m} \sum_{i=1}^{m} (h_0(x^i) - y^i) \cdot x^i$$

Dadas estas derivadas, el algoritmo queda de la siguiente manera:

Repetir hasta converger {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_0(x^i) - y^i)$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_0(x^i) - y^i) \cdot x^i$$

- Actualizar θ_0
- Actualizar θ_1

}

1.1.4 Implementación en código

[1]:
```
import time
from IPython.display import clear_output

import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

matplotlib.style.use('ggplot')
```

Uno de los dataset por excelencia para comprender el algoritmo de Regresión Lineal es el del precio de las casas de Boston.

La data está disponible en el siguiente enlace. Enlace

[2]:
```
data = pd.read_csv('../data/housing.csv', header=None, sep='\s+')
data.head()
```

[2]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	18.7	396.90	5.33	36.2

Vemos que el dataset no tiene columnas, afortunadamente Kaggle nos proporciona el nombre de las mismas para poder añadírselas.

- CRIM: tasa de criminalidad en la zona.
- ZN: proporción de suelo residencial zonificado para lotes de más de 25,000 pies cuadrados.
- INDUS: proporción de acres comerciales no minoristas por ciudad.
- CHAS: variable ficticia del río Charles (= 1 si el tramo limita con el río; 0 en caso contrario).
- NOX: concentración de óxidos nítricos (partes por 10 millones).
- RM: promedio de cuartos por vivienda.
- EDAD: proporción de unidades ocupadas por sus propietarios construidas antes de 1940.
- DIS: distancias ponderadas a cinco centros de empleo de Boston.
- RAD: índice de accesibilidad a las carreteras radiales.
- TAX: tasa de impuesto a la propiedad de valor total por $ 10,000. PTRATIO: proporción de alumnos por maestro por ciudad.
- B: 1000 * (Bk−0.63)2 donde Bk es la proporción de negros por ciudad.

- LSTAT: % de la población con status inferior.
- MEDV: valor medio de las viviendas ocupadas por sus propietarios en miles de dólares.

[3]:
```
columns = ['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',␣
↪'TAX', 'PTRATIO',
           'B', 'LSTAT', 'MEDV']

data.columns = columns
data.head()
```

[3]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	\
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	

	PTRATIO	B	LSTAT	MEDV
0	15.3	396.90	4.98	24.0
1	17.8	396.90	9.14	21.6
2	17.8	392.83	4.03	34.7
3	18.7	394.63	2.94	33.4
4	18.7	396.90	5.33	36.2

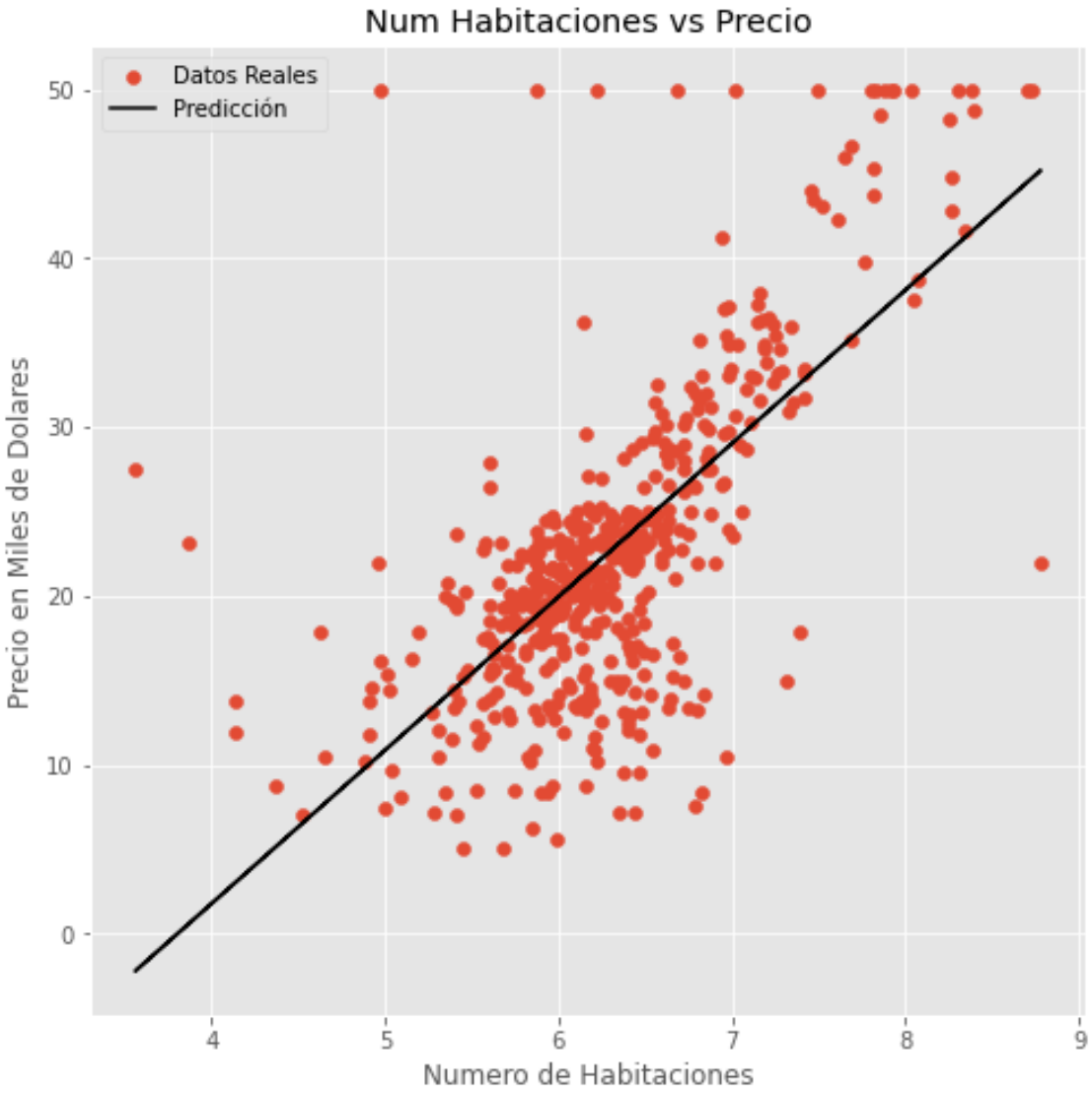
Se procederá a predecir el precio medio de la vivienda (MEDV) respecto al número de habitaciones que la compone (RM).

[9]:
```
# Variables necesarias para el modelo
m = len(data)
x = data['RM']
y = data['MEDV']
```

[8]:
```
# Function to calculate y_pred
def get_y_pred(x, theta):
    return theta[0] + theta[1] * x

# Function to calculate the cost
def get_cost(y_pred, y):
    return (np.sum((y_pred - y).dot((y_pred - y).T))) / (2 * m)

# Gradient descent to find the optimal parameters
def get_predictions(x, y, n_iter, theta, alpha=0.01):
    for i in range(n_iter):
        # Predecimos y respecto a theta en las iteración i-ésima

        y_pred = get_y_pred(x, theta)

        # Calculamos los parámetros theta_0 y theta_1 en la iteración i-ésima
```

```
        theta_0 = theta[0] - (alpha * ((np.sum(y_pred - y)) / m))
        theta_1 = theta[1] - (alpha * ((np.sum((y_pred - y) * x)) / m))

        theta = [theta_0, theta_1]

        # Calculamos el costo respecto a y_pred en las iteración i-ésima
        cost = get_cost(y_pred, y)

        # Hacemos una visualización cada 1000 iteraciones
        if i % 1000 == 0:
            print('Predicción a la iteración {}'.format(i))
            print('Costo a la iteración {}: {}'.format(i, cost))
            print('Theta_0: {}, Theta_1: {}'.format(theta_0, theta_1))
            print('Precio = {} + {} * X'.format(theta_0, theta_1))
            plt.figure(figsize=(8,8))
            plt.scatter(x, y, label='Datos Reales')
            plt.plot(x, y_pred, color='k', label='Predicción')
            plt.title('Num Habitaciones vs Precio')
            plt.xlabel('Numero de Habitaciones')
            plt.ylabel('Precio en Miles de Dolares')
            plt.legend(loc='best')
            plt.show()

            time.sleep(0.1)
            clear_output(wait=True)

    # Retornamos los parámetros theta_0 y theta_1 óptimos encontrados
    return theta

# Corremos el algoritmo.
initial_theta = theta = np.array([0,0])
n_inter = 50001

theta_optim = get_predictions(x, y, n_inter, initial_theta)
```

```
Predicción a la iteración 50000
Costo a la iteración 50000: 21.800321736784475
Theta_0: -34.584358797761816, Theta_1: 9.088548160811078
Precio = -34.584358797761816 + 9.088548160811078 * X
```



Como se puede observar, el costo se va minimizando en cada iteración, y gracias al algoritmo del descenso del gradiente podemos encontrar los valores de Theta óptimos para la predicción.

1.1.5 Algoritmo con la librería SK-Learn

No es escalable estar implementando en código el algoritmo cada vez que se necesite, para ello, podemos ayudarnos de librerías como Scikit Learn que ya lo tiene implementado y es muy eficiente. Es así como podemos, en pocas líneas de código, llegar a una solución óptima.

[10]:
```
from sklearn.linear_model import LinearRegression
```

La diferencia es que Sklearn necesita un array de 2 dimensiones para funcionar, por lo que será necesario re escalar la data.

[11]:
```
x = x.values.reshape(-1,1)
y = y.values.reshape(-1,1)
```

[12]:
```
model = LinearRegression()
model.fit(x, y)

# Opteniendo theta_0 y theta_1
theta_0 = model.intercept_
theta_1 = model.coef_
theta_0, theta_1
```

[12]:
```
LinearRegression()
```

Como se puede observar, hemos obtenido los mismo valores pero con una implementación más sencilla, clara y rápida.

1.2 Regresión Lineal Múltiple

1.2.1 Hipótesis de la Regresión Lineal Múltiple

El algoritmo de regresión lineal simple puede ser muy útil, pero en ocasiones puede quedarse corto, o para problemas un poco más complejos, pueda que no sea la mejor alternativa para realizar predicciones.

El modelo de regresión lineal múltiple sigue el mismo concepto que el simple, la diferencia es que ahora agregamos más de una variable independiente x. Siguiendo este sentido, la hipótesis de este algoritmo queda de la siguiente manera:

$$Hypotesis : h_0(x) = \theta_0 + \theta_1 \cdot x_1 + \theta_2 \cdot x_2 + \cdots + \theta_n \cdot x_n$$

donde:

- $h_0(x)$: Es la predicción, es decir la variable y.
- $\theta_0,\ \theta_2,\ \ldots,\ \theta_n$: Son los parámetros que el algoritmo trata de encontrar tales que el error de la predicción sea lo mínimo posible.

1.2.2 Función de coste

La función de costo para este algoritmo es la misma que un modelo lineal, la diferencia es que ahora se tienen que inicializar más parámetros θ, tales que el número de θ_s es igual a $n + 1$, donde n es el total de variables independientes.

$$J(\theta_s) = \frac{1}{2m} \sum_{i=1}^{m} (h_0(x^i) - y^i)^2$$

El objetivo final del algoritmo es minimizar $J(\theta_s)$ es decir, encontrar los θ_s óptimos que minimicen el costo o también llamado pérdida.

1.2.3 Descenso del Gradiente

Tomamos como referencia el descenso del gradiente para el modelo lineal, la diferencia es que ahora tenemos que encontrar más parámetros, pero el proceso es el mismo.

Repetir hasta converger {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_0(x^i) - y^i)$$

$$\theta_s \; 1 - n := \theta_i - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_0(x^i) - y^i) \cdot x^i$$

- Actualizar θ_0
- Actualizar θ_s del 1 a n

}

1.2.4 Implementación en código

En este caso de estudio, y para ilustrar como se crea un algoritmo de regresión lineal múltiple, tomaremos el mismo set de datos y agregaremos una variable más, esta es la variable LSTAT.

Se procede a seleccionar las variables del conjunto de datos.

[12]:
```
# Variables independientes
x = data[['RM', 'LSTAT']]

# Variable a predecir y
y = data['MEDV']

m = len(data)

# Agregando una columna de 1 a la variable x
x = x.assign(ones=1)

# Re ordenando para que los unos sean la primera columna
x = x[['ones', 'RM', 'LSTAT']].values
```

Nota: Agregamos la primera columna de unos para implementar una solución vectorizada. Theta_0 no se multiplica con ninguna variable X. Para no alterar este parámetro, lo multiplicamos por 1.

[14]:
```
# Predecir y-value
def pred_y_multy(x, thetas):
    return x.dot(thetas.T)

# Función de coste
def get_cost_multy(y_pred, y):
    return (1 / 2*m) * np.sum((y_pred - y) * (y_pred - y))
```

```
# Inplementación del algoritmo de descenso del gradiente
def get_gradient_multy(x, y, n_iter, thetas, alpha):
    for i in range(n_iter):
        y_pred = pred_y_multy(x, thetas)

        thetas = thetas - (alpha * ((1 / m)*((y_pred - y).T.dot(x))))

    return thetas

initial_thetas = np.array([0,0,0])
alpha = 0.001
n_iter = 1000000
thetas = get_gradient_multy(x,y,n_iter, initial_thetas, alpha)

print('Los Theta óptimos son: {}'.format(thetas))
```

```
Los Theta óptimos son: [-1.3528106   5.09403207 -0.64241202]
```

1.2.5 Algoritmo con Scikit-Learn

En la práctica, aplicar un código desde cero resulta poco escalable, afortunadamente, existen librerías que nos ayudan a obtener resultados de forma rápida y precisa. Scikit-Learn es por excelencia una de las mejores en el ámbito del Machine Learning. Podemos implementar un modelo de regresión múltiple con solo unas pocas líneas de código.

[18]:
```
from sklearn.linear_model import LinearRegression

X = data[['RM', 'LSTAT']].values
y = data['MEDV'].values.reshape(-1, 1)

slr = LinearRegression()
slr.fit(X, y)

coef = slr.coef_
intercept = slr.intercept_

print(intercept, coef)
```

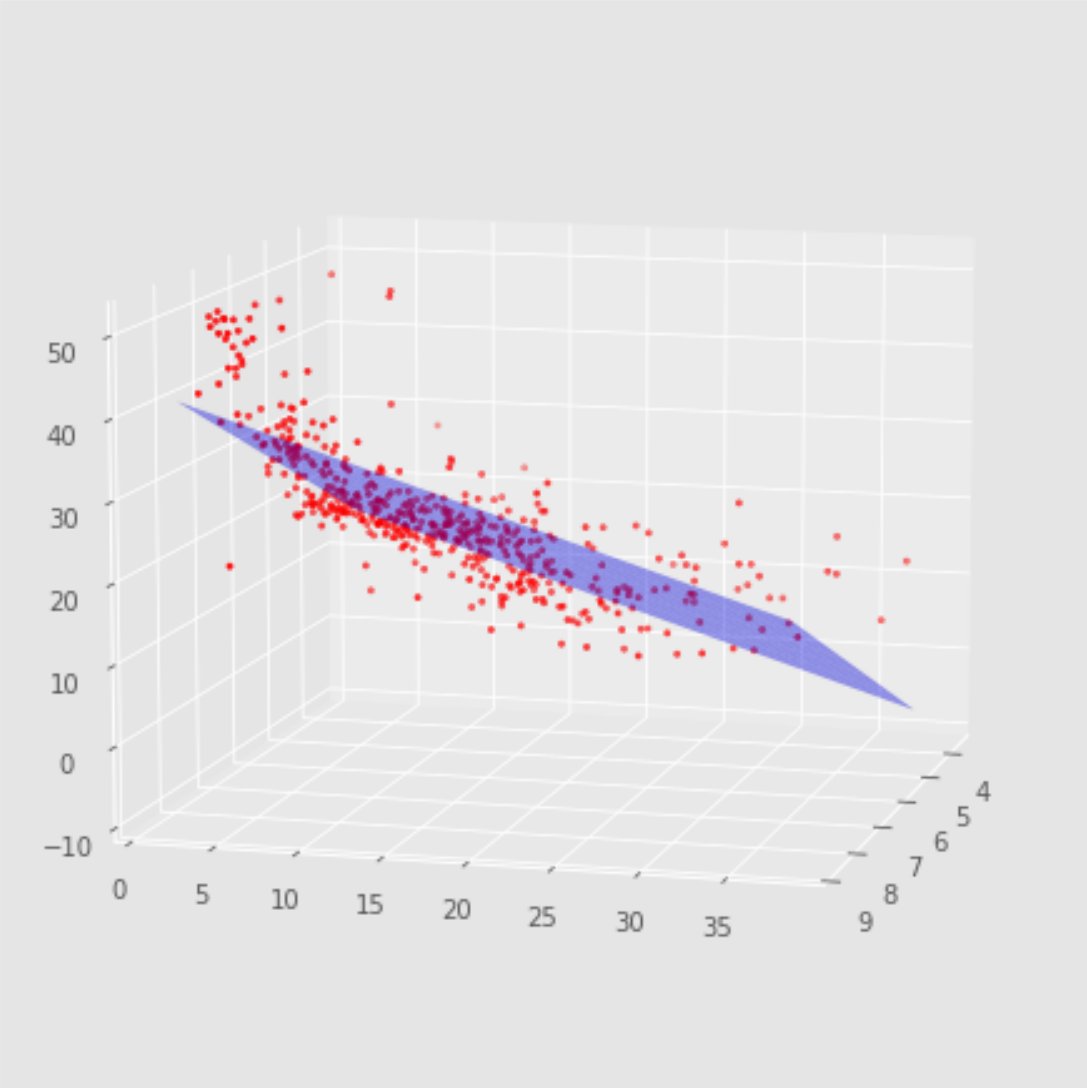
```
[-1.35827281] [[ 5.09478798 -0.64235833]]
```

Los valores que nos da como resultado Scikit-Learn son parecidos a los que se obtuvieron con el algoritmo aplicado en código. Pero hay que destacar que, Scikit-Learn es más rápido y por ende una mejor solución.

Representar visualmente datos con dos o más características analizadas resulta complicado, en el siguiente gráfico se busca resaltar como los datos son proyectados a una solución dada por el algoritmo. Pero no es la mejor manera de hacerlo.

Lo importante es entender los resultados y tomar decisiones en base a los mismos.

[52]:
```
from mpl_toolkits.mplot3d import *

x1_range = np.arange(data['RM'].min(),data['RM'].max())
x2_range = np.arange(data['LSTAT'].min(),data['LSTAT'].max())

X1, X2 = np.meshgrid(x1_range,x2_range)

plano = pd.DataFrame({'RM':X1.ravel(), 'LSTAT':X2.ravel()})
pred = slr.predict(plano).reshape(X1.shape)

fig = plt.figure(figsize=(10,8))
ax = fig.gca(projection='3d')
ax.plot_surface(X1,X2,pred, alpha=0.4, color='b')

ax.scatter3D(data['RM'], data['LSTAT'], data['MEDV'], color='r', marker='.')
ax.view_init(elev=10, azim=15)
plt.show()
```


50
40
30
20
10
0
−10
0
5
10
15
20
25
30
35
4
5
6
7
8
9