Regression/Chapter02 - HandsOn Exercise

In [1]:
```
import pandas as pd
```

In [2]:
```
import numpy as np
```

In [3]:
```
# Read X.txt
# It usually works without the "lineterminator='\n'" option
X=pd.read_csv("./HousingData/X.txt",sep='\t', lineterminator='\n', header=None )
X.head()
```

Out[3]:

	0	1	2	3
0	1	3	4.0	1500
1	1	3	2.0	1700
2	1	3	3.0	1245
3	1	5	1.0	2440
4	1	4	4.0	2962

In [4]:
```
# Read Y.txt
y=pd.read_csv("./HousingData/Y.txt", header=None )
y.head()
```

Out[4]:

	0
0	150000
1	115000
2	159900
3	204999
4	300000

Least Squares: Linear Regression

Performing Linear Regression using Scikit-Learn is simple:

In [5]:
```
# The input X should not have the X0=1 column in it, so, we drop it and create and use X2.
X2 = X.drop([0], axis=1)
X2.head()
```

Out[5]:

	1	2	3
0	3	4.0	1500
1	3	2.0	1700
2	3	3.0	1245
3	5	1.0	2440
4	4	4.0	2962

In [6]:
```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(X2, y)
lin_reg.intercept_, lin_reg.coef_ # Note how the intercept and the rest of the coefficient
s are returned.
```

Out[6]:
```
(array([33867.53322567]),
 array([[-36761.61229634,  10501.035838  ,    132.911633  ]]))
```

The `LinearRegression` class is based on the `scipy.linalg.lstsq()` function (the name `linalg` stands for "**linear algebra**" and `lstsq()` stands for “**least squares**”), which you could call directly.

Now we can predict the house-price for the Query, Q=[5 3 2500], that is a house having 5 bedrooms, 3 bathrooms and 2500 sqft living area as:

In [7]:
```
Q = pd.DataFrame([[5,3,2500]])
Q
```

Out[7]:

	0	1	2
0	5	3	2500

In [8]:
```
lin_reg.predict(Q)
```

Out[8]:
```
array([[213841.66175613]])
```

The Normal Equation

To find the value of **Beta** that minimizes the cost or error function, we can use a *closed-form* solution - in other words, a mathematical equation that gives the result directly. This is called the *Normal Equation*, expresses as $Beta = (X^T X)^{-1} X^T y$.

We will use the `inv()` function from NumPy’s linear algebra module (`np.linalg`) to compute the inverse of a matrix, and the `dot()` method for matrix multiplication

In [9]:
```
Beta = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(y)
Beta
```

Out[9]:
```
array([[ 33867.53322568],
       [-36761.61229634],
       [ 10501.035838  ],
       [   132.911633  ]])
```

If you need to use *pseudoinverse*, you can use `np.linalg.pinv()` as following:

In [10]:
```
np.linalg.pinv(X.T.dot(X)).dot(X.T).dot(y)
```

Out[10]:
```
array([[ 33867.53322798],
       [-36761.61229715],
       [ 10501.03583796],
       [   132.911633  ]])
```

Now we use Beta for prediction.

- For a Query, Q=[1 5 3 2500], that is a house having 5 bedrooms, 3 bathrooms and 2500 sqft living area, for example,
  - we can predict the house price by: Q*Beta;
  - which should return 2.1384e+005 or, 213,840

Note: Q is in row form and Beta is in column form. Thus, we do `Q*Beta` instead of `Q.T*Beta` . And the '1' indicates X_0.

In [11]:
```
Q= np.array([1,5,3, 2500])
Q
```
Out[11]:
```
array([   1,    5,    3, 2500])
```

In [12]:
```
Q.dot(Beta)
```
Out[12]:
```
array([213841.66175613])
```

Additional Information

Now, assume the matrix X did not have *X0*=1 column inserted and we need to insert it:

In [13]:
```
X.shape
```
Out[13]:
```
(50, 4)
```

In [14]:
```
R_cnt=X.shape[0]
R_cnt
```
Out[14]:
```
50
```

`np.c_` is array concatenate

In [15]:
```
X1 = np.c_[np.ones((R_cnt,1)), X]
```

In [16]:
```
X1
```

Out[16]:
```
array([[1.000e+00, 1.000e+00, 3.000e+00, 4.000e+00, 1.500e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.700e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 3.000e+00, 1.245e+03],
       [1.000e+00, 1.000e+00, 5.000e+00, 1.000e+00, 2.440e+03],
       [1.000e+00, 1.000e+00, 4.000e+00, 4.000e+00, 2.962e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.953e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 3.000e+00, 1.750e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.770e+03],
       [1.000e+00, 1.000e+00, 4.000e+00, 2.000e+00, 1.768e+03],
       [1.000e+00, 1.000e+00, 2.000e+00, 1.000e+00, 1.100e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.100e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.300e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.625e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.500e+03],
       [1.000e+00, 1.000e+00, 4.000e+00, 4.000e+00, 3.800e+03],
       [1.000e+00, 1.000e+00, 4.000e+00, 4.000e+00, 4.314e+03],
       [1.000e+00, 1.000e+00, 2.000e+00, 1.000e+00, 1.100e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.500e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 1.000e+00, 1.406e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 1.000e+00, 1.089e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.500e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.500e+03],
       [1.000e+00, 1.000e+00, 4.000e+00, 3.000e+00, 2.480e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.500e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 3.000e+00, 1.685e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.798e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 2.576e+03],
       [1.000e+00, 1.000e+00, 2.000e+00, 2.500e+00, 2.050e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.349e+03],
       [1.000e+00, 1.000e+00, 2.000e+00, 1.000e+00, 1.124e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.258e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 1.000e+00, 1.255e+03],
       [1.000e+00, 1.000e+00, 4.000e+00, 3.000e+00, 2.280e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.300e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.580e+03],
       [1.000e+00, 1.000e+00, 5.000e+00, 4.000e+00, 3.500e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.438e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.500e+00, 2.110e+03],
       [1.000e+00, 1.000e+00, 5.000e+00, 7.500e+00, 5.823e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.702e+03],
       [1.000e+00, 1.000e+00, 4.000e+00, 2.000e+00, 1.404e+03],
       [1.000e+00, 1.000e+00, 2.000e+00, 1.000e+00, 9.800e+02],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.450e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.530e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 1.143e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 3.000e+00, 3.193e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 2.130e+03],
       [1.000e+00, 1.000e+00, 3.000e+00, 2.000e+00, 2.400e+03],
       [1.000e+00, 1.000e+00, 5.000e+00, 3.000e+00, 2.367e+03],
       [1.000e+00, 1.000e+00, 4.000e+00, 3.000e+00, 2.860e+03]])
```