Linear Regression- House Prices

Siyuan Zhang

1 Resources

Student: Siyuan Zhang
Language: Python
Code: https://github.com/Siyuan-gwu/Machine-Learning-Linear-Regression-Housing-prices
Instruction: The instruction to run the code is on github (readme)
Resource:

1. House prices data from Kaggle
2. https://towardsdatascience.com/introduction-to-machine-learning-algorithms-linear-regression-14c4e325882a

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

2 Dataset details

There is one dataset, which contains 20 columns and different factors as well as price.

id
date
price
bedrooms
bathrooms
sqft_living
sqft_lot
floors
waterfront
view
condition
grade
sqft_above
sqft_basement
yr_built
yr_renovated
zipcode
lat

long
sqft_living15

Inspect the Dataset

```
data = pd.read_csv("kc_house_data.csv")
print (data.describe())
```

```
                 id         price  ...  sqft_living15     sqft_lot15
count  2.161300e+04  2.161300e+04  ...   21613.000000   21613.000000
mean   4.580302e+09  5.400881e+05  ...    1986.552492   12768.455652
std    2.876566e+09  3.671272e+05  ...     685.391304   27304.179631
min    1.000102e+06  7.500000e+04  ...     399.000000     651.000000
25%    2.123049e+09  3.219500e+05  ...    1490.000000    5100.000000
50%    3.904930e+09  4.500000e+05  ...    1840.000000    7620.000000
75%    7.308900e+09  6.450000e+05  ...    2360.000000   10083.000000
max    9.900000e+09  7.700000e+06  ...    6210.000000  871200.000000
```

Check whether there is missing data.

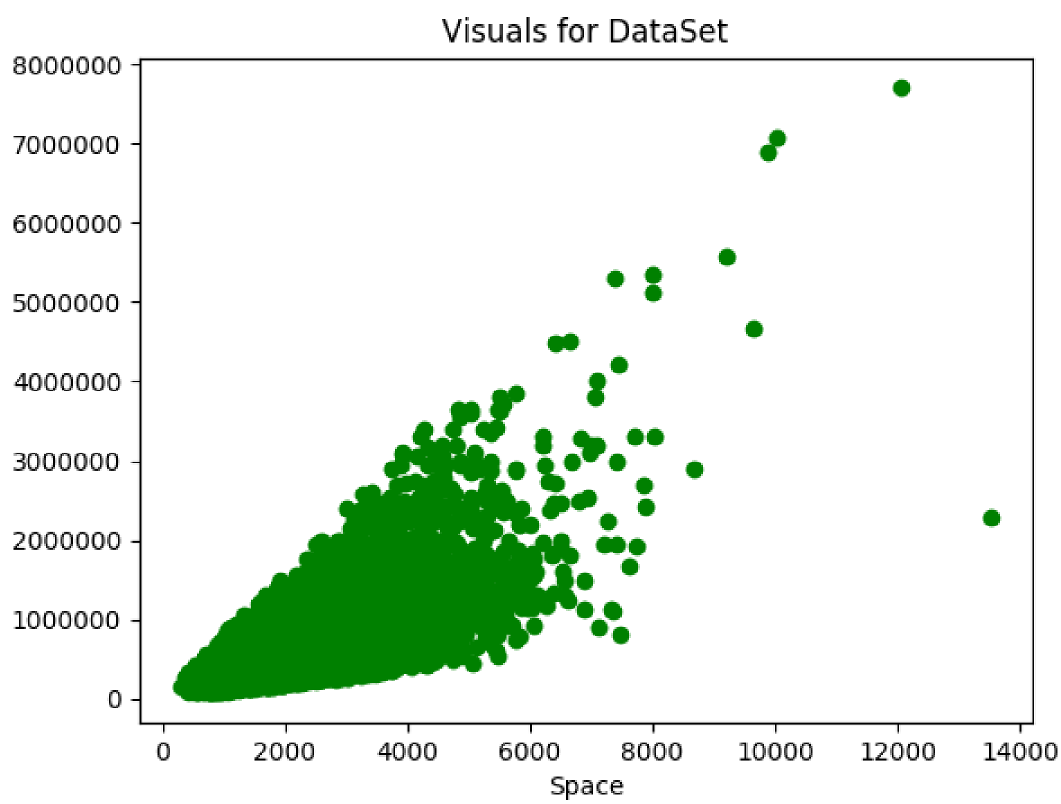
```
missingTotal = data.isnull().sum()
print (missingTotal)
```

```
id               0
date             0
price            0
bedrooms         0
bathrooms        0
sqft_living      0
sqft_lot         0
floors           0
waterfront       0
view             0
condition        0
grade            0
sqft_above       0
sqft_basement    0
yr_built         0
yr_renovated     0
zipcode          0
lat              0
long             0
sqft_living15    0
sqft_lot15       0
dtype: int64
```

I found that there is no missing data, which is perfect that I do not need to process the missing data.

Dataset Visualization

```
space = data['sqft_living']
price = data['price']
plt.scatter(space, price, color='green')
axes = plt.gca()
plt.title("Visuals for DataSet")
plt.xlabel("Space")
plt.ylabel("Price")
plt.show()
```



The x-axis is the space of house.
The y-axis is the prices

Feature Selection

```
print (data['sqft_living'])
```

```
0        1180
1        2570
2         770
3        1960
4        1680
         ...
21608    1530
21609    2310
21610    1020
21611    1600
21612    1020
Name: sqft_living, Length: 21613, dtype: int64
```

In this project, I chose the 'sqrt_living' as the feature to predict the house prices. Since in general, the living area is the most likely factor affecting the prices of a house.

Data Splitting

```
#read the data
from sklearn.model_selection import train_test_split
space_train, space_test, price_train, price_test =
train_test_split(space_arr,price_arr,test_size=0.4,train_size=0.6)
```

Here, I split the dataset into three parts, training data (60%) and testing data (40%). Firstly, I used the training data to optimize the linear regression. Then, I would use the result linear regression model to test the testing data and predict the accuracy.

Data Pre-processing

```
price = data['price']
print (price)
```

```
0        221900.0
1        538000.0
2        180000.0
3        604000.0
4        510000.0
           ...
21608    360000.0
21609    400000.0
21610    402101.0
21611    400000.0
21612    325000.0
Name: price, Length: 21613, dtype: float64
```

I noticed that the prices are huge, which will encounter overflow during the training process. So, I did data normalization on the dataset (min-max normalization).

The min-max normalization method is a linear transformation of the original data. Let minA and maxA be the minimum and maximum values of the attribute A, respectively, and normalize an original value x of A by min-max to the value x' in the interval [0, 1].

Equation: $$x' = \frac{x - x\_min}{x\_\max - x\_min}$$

```
# min-max normalization
def MaxMinNormalization(x):
    """[0,1] normaliaztion"""
    x = (x - np.min(x)) / (np.max(x) - np.min(x))
    return x
```

3 Algorithm Description

1. Linear regression: Simple linear regression is a type of regression analysis where the number of independent variables is one and there is a linear relationship between the independent(x) and dependent(y) variable.

   $$Equation: y = a1 * x + a0$$

2. Cost Function: The cost function helps us to figure out the best possible values for a1 and a0 which would provide the best fit line for the data points. Since we want the best values for a1 and a0, we convert this search problem into a minimization problem where we would like to minimize the error between the predicted value and the actual value.

$$minimize \frac{1}{n} \sum_{i=1}^{n} (pred_i - y_i)^2$$

$$J = \frac{1}{n} \sum_{i=1}^{n} (pred_i - y_i)^2$$

```
# cost function
def compute_error(b, m, space_data, price_data):
    for i in range(len(space_data)):
        x = space_data[i]
        y = price_data[i]
        totalError = (y - m * x - b) ** 2
    totalError = np.sum(totalError, axis=0)
    return totalError/len(space_data)
```

3. Gradient Descent: Gradient descent is a method of updating a0 and a1 to reduce the cost function. The idea is that we start with some values for a0 and a1 and then we change these values iteratively to reduce the cost.

$$J = \frac{1}{n} \sum_{i=1}^{n} (pred_i - y_i)^2$$

$$J = \frac{1}{n} \sum_{i=1}^{n} (a_0 + a_1 \cdot x_i - y_i)^2$$

$$\frac{\partial J}{\partial a_0} = \frac{2}{n} \sum_{i=1}^{n} (a_0 + a_1 \cdot x_i - y_i) \implies \frac{\partial J}{\partial a_0} = \frac{2}{n} \sum_{i=1}^{n} (pred_i - y_i)$$

$$\frac{\partial J}{\partial a_1} = \frac{2}{n} \sum_{i=1}^{n} (a_0 + a_1 \cdot x_i - y_i) \cdot x_i \implies \frac{\partial J}{\partial a_1} = \frac{2}{n} \sum_{i=1}^{n} (pred_i - y_i) \cdot x_i$$

$$a_0 = a_0 - \alpha \cdot \frac{2}{n} \sum_{i=1}^{n} (pred_i - y_i)$$

$$a_1 = a_1 - \alpha \cdot \frac{2}{n} \sum_{i=1}^{n} (pred_i - y_i) \cdot x_i$$

```
# gradient descent function
def compute_gradient(b_cur, m_cur, space_data, price_data, learning_rate):
    b_gradient = 0
    m_gradient = 0

    N = float(len(space_data))

    for i in range(0, len(space_data)):
        x = space_data[i]
        y = price_data[i]
        b_gradient += -(2 / N) * (y - ((m_cur * x) + b_cur))
```

```
        m_gradient += -(2 / N) * x * (y - ((m_cur * x) + b_cur))

    b_next = b_cur + (-learning_rate * b_gradient)
    m_next = m_cur + (-learning_rate * m_gradient)
    return (b_next, m_next)
```

The partial derivates are the gradients and they are used to update the values of a_0 and a_1. Alpha is the learning rate which is a hyperparameter that you must specify. A smaller learning rate could get you closer to the minima but takes more time to reach the minima, a larger learning rate converges sooner but there is a chance that you could overshoot the minima.

4 Algorithm Results

Selection of learning rate

Different selection of learning rate will affect the accuracy significantly.

This method is used to calculate the loss with the increasement of iteration.

```
def optimizer(space_data, price_data, initial_b, initial_m, learning_rate, num_iter):
    b = initial_b
    m = initial_m

    for i in range(num_iter):
        b, m = compute_gradient(b, m, space_data, price_data, learning_rate)

        if i % 10 == 0:
            print ('Iter: %s'%i, 'error: %s'%compute_error(b, m, space_data, price_data))

    return [b, m]
```

1. First, I choose 0.0001 as learning rate.

```
Iter: 0 error: 0.06409958575411255
Iter: 10 error: 0.0035907389634170636
Iter: 20 error: 0.02748301545767841
Iter: 30 error: 0.04063013705042877
Iter: 40 error: 0.04565434209569472
Iter: 50 error: 0.04741900193928002
Iter: 60 error: 0.04802296570927402
Iter: 70 error: 0.048227800588250644
Iter: 80 error: 0.04829692643989075
Iter: 90 error: 0.04832008251938799
```

We can notice that after 10th iteration, the loss becomes larger.

2. Then, I chose 0.01 as the learning rate.

```
Iter: 0 error: 26.94779862337903
Iter: 10 error: 7.923903193734181e+20
Iter: 20 error: 2.2666930828258978e+40
Iter: 30 error: 6.484048840686979e+59
Iter: 40 error: 1.85481173816428e+79
Iter: 50 error: 5.305830768029269 5e+98
Iter: 60 error: 1.5177734515970466e+118
Iter: 70 error: 4.341706984425723e+137
Iter: 80 error: 1.2419784730567115e+157
```

Also, the loss becomes larger and larger from the beginning.

3. Finally, I chose 0.0005 as the learning rate.

```
Iter: 0 error: 0.008314890920516507
Iter: 10 error: 0.00016472428829718386
Iter: 20 error: 0.0001646001918650764
Iter: 30 error: 0.00016582887794771458
Iter: 40 error: 0.00016706103644925142
Iter: 50 error: 0.0001682957390684825
Iter: 60 error: 0.0001695329662478126
Iter: 70 error: 0.00017077269912224864
Iter: 80 error: 0.00017201491890592426
Iter: 90 error: 0.00017325960689141187
```

We can notice that the loss is much smaller than above. Then I use the 0.0005 as the learning rate.

Result Visualization

Firstly, I use the training data to get the model.

```
# training result
b, m = linear_regression(space_train, price_train, 0.0005)
```

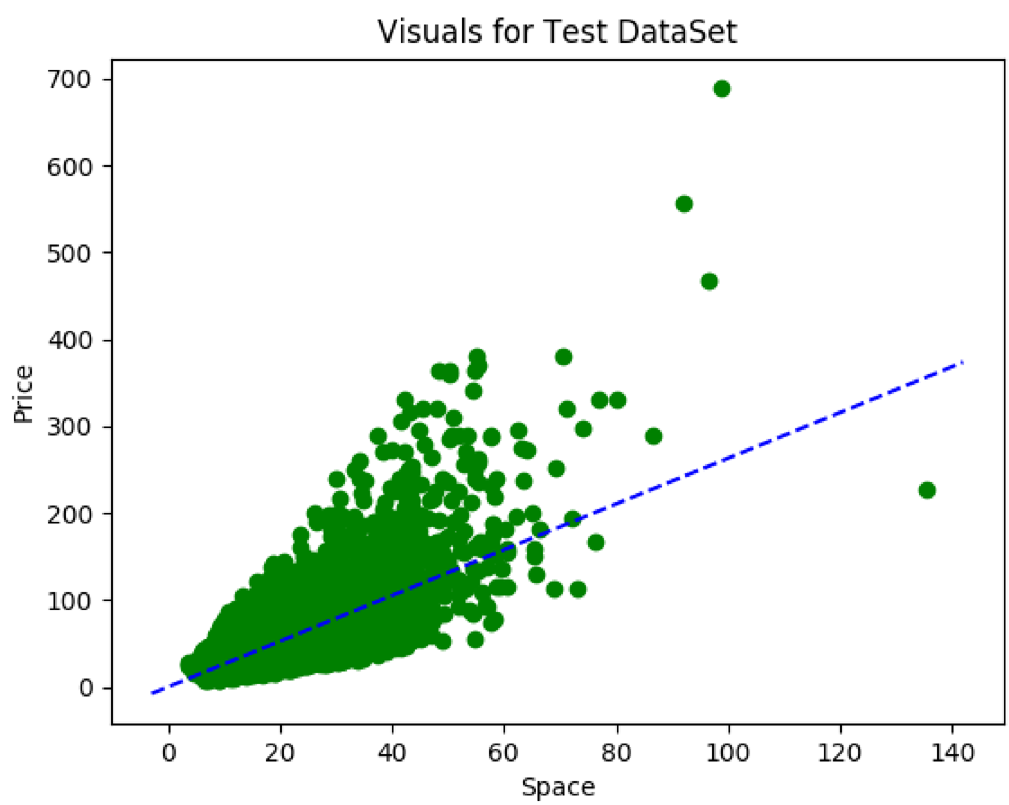
Then, use the testing data to predict the result and plot the picture.

```
# visualize the test results
plt.scatter(space_test, price_test, color='green')
axes = plt.gca()
x_vals = np.array(axes.get_xlim())
y_vals = b + m * x_vals
plt.plot(x_vals, y_vals, '--', color='blue')
plt.title("Visuals for Test DataSet")
plt.xlabel("Space")
plt.ylabel("Price")
plt.show()
```

```
[0.04053222] [2.63134492]
```

The linear result is $y = 0.04053 + 2.63134492 * x$

Visualization



5 Runtime

For a (n x k) matrix, (X’ X) takes $O(n * k^2)$ time and produces a $(k * k)$ matrix.
The matrix inversion of a $(k * k)$ matrix takes $O(k^3)$ time
(X’ Y) takes $O(n * k^2)$ time and produces a $(k * k)$ matrix.
The final matrix multiplication of two $(k * k)$ matrices takes $O(k^3)$ time.

So, the overall Big-O running time is $O(k^2(n * k))$

In this project, I only chose one feature, if the number of iterations is n, the number of data is m, then the time complexity is $O(n * m)$

```
start = time.thread_time()
b, m = linear_regression(space_train, price_train, 0.0005)
end = time.thread_time()
print ('Time used: {}'.format(end - start))
```

Runtime: `Time used: 10.74575913`