Jumping Into The World of Machine Learning:-

Part 1 – Simple Linear Regression:-

- Article Objectives:-
    1) Understanding the algorithm.
    2) Implementation of the algorithm in Python.

Introduction:-

Linear Regression is a supervised learning algorithm used in Machine Learning in which the output of the algorithm has a constant slope and is in the form of continuous values.

Its used to predict values within a continuous range such as sales, price of items etc. For example, we may be given the information about the price of houses according to the number of bedrooms in the house. Our task is to device an ML model that can predict the price of new houses based on the data provided to us, then the algorithm around which we will build our model is Linear Regression.

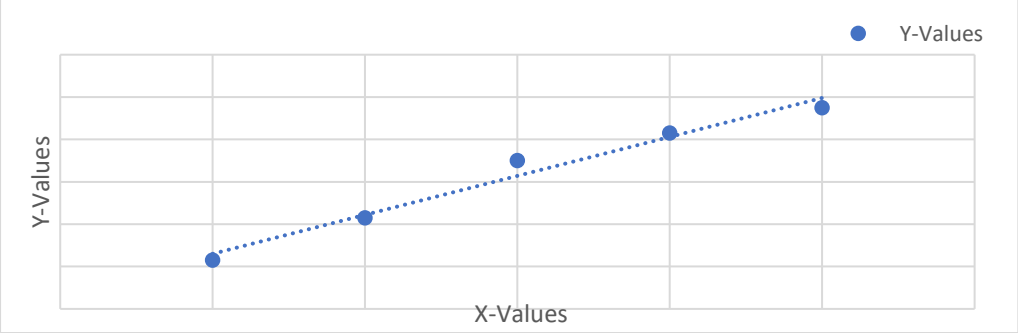
Linear Regression is further divided into two categories:- *Simple Linear Regression* and *Multivariable Linear Regression.*

In this article we are going to cover Simple Linear Regression along with its implementation in Python.

Simple Linear Regression:-

Simple linear regression uses traditional slope-intercept form, where W and B are the variables our algorithm will try to *"learn"* to produce the most accurate predictions. x represents our input data and y represents our prediction.

$$y = Wx + B$$



In lay man terms, the 'W' variable (referred to as the *Weight*) is used to rotate the best fit line and the 'B' variable (referred to as the *Bias*) is used to translate the best fit line on given data to find the predicted values that are the most closest to the actual values.

In practice, the Weight and the bias parameters are randomly initialised to some arbitrary value.

Real World Application of Simple Linear Regression:-

Let's consider the following data which is completely made up by me for the purpose of this article.

House Serial Number	Number of Bedrooms	Price(in 10000 rupees)
1	3	40
2	2	30
3	5	60
4	1	20

Now in this dataset our input values i.e. the *X-Values* is the *Number of Bedrooms* attribute and our output values i.e. the *Y-Values* is the *Price(in 10000 rupees)* attribute.

Therefore our linear regression equation becomes:-

$$\boldsymbol{Price = W * Number\ of\ Bedrooms + B}$$

The task associated with this problem is to predict the values of new data points that are currently not in the dataset i.e the price of new houses based on their number of bedrooms.

Python Code to read the data:-

Let's consider that the given data is stored in the 'csv' file format. Then the code to read this data is as follows:-

```
import pandas as pd
data = pd.read_csv('sample_data.csv',index_col=0)
x_train = data['Number of Bedrooms'].tolist()
y_train = data['Price (in 10000 rupees)'].tolist()

[23] print(x_train)
     print(y_train)

     [3, 2, 5, 1]
     [40, 30, 60, 20]
```

Cost Function:-

In order to evaluate the performance of our linear regression model we must have a metric that tells us how much our predicted value deviates from the actual value. This metric is important as it guides us to tune our model in a way that decreases this deviation and hence makes our model better at its predictions.

One such metric that is used in Simple Linear Regression is called as the *Mean Squared Error(*abbreviated as *MSE).*

MSE measures the average squared difference between an observation's actual and predicted values. The output is a single number representing the cost, or score, associated with our current set of weights. Our goal is to minimize MSE to improve the accuracy of our model.

For a given simple linear equation y = Wx+B, we can calculate MSE as:-

$$\boldsymbol{MSE} = \frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{i}=\mathbf{1}}^{\boldsymbol{n}}(\boldsymbol{y\_i} - (\boldsymbol{W} * \boldsymbol{x\_i} - \boldsymbol{B}))\text{^}\mathbf{2},$$

Where

- N → Number of observations in the dataset. In this case it's the number of houses.
- i → i^{th} observation in the dataset.
- W → Weight value
- B → Bias value
- y_i is the actual value of the i^{th} observation and W*x_i+B is our prediction

Python Code to return the cost values:-

```
[8] def cost_function(number_bedrooms, price, weight, bias):
        observations = len(number_bedrooms)# here number_bedrooms is vector containing the values of number of bedrooms
                           # the length of this vector is equal to the number of observations present
                           # in the dataset
        total_error = 0.0
        for i in range(observations):
          #here price is avector containing our actual output values
            total_error += (price[i] - (weight*number_bedrooms[i] + bias))**2
        return total_error / observations
```

Gradient Descent:-

In the previous section we discussed about the need of a cost function and the fact that minimizing the cost function leads to improving the accuracy of our regression model.

But how do we minimize our cost function?

The answer to the above question is an algorithm called as *'Gradient Descent'*.

Gradient Descent is one of the several optimization algorithms available in the toolbox of a machine learning practitioner. A detailed explanation of the working of this algorithm is behind the scope of this article, but ,in lay man terms, what this algorithm essentially does is that it updates the *Weight* and the *Bias* parameters of our model by subtracting these parameters from the derivative of the cost function times a constant.

i.e. $$new_W = old_W - \frac{d}{dW}(cost\ function) * alpha$$

$$new_B = old_B - \frac{d}{dB}(cost\ function) * alpha$$

, where *alpha* is called the *learning rate*. Alpha determines how fast or slow the cost function should move towards the local minima (the point beyond which the cost function would no longer decrease.)

The formula for the partial derivatives of the cost function w.r.t Weight and Bias respectively are:-

$$\frac{d}{dW}(cost\ function) = -2 * x * (y - (W * x + B))$$

$$\frac{d}{dB}(cost\ function) = -2 * (y - (W * x + B))$$

Python Function to implement Gradient Descent:-

```
def update_weights(number_bedrooms, price, weight, bias, learning_rate):
    weight_derivative = 0 # partial derivative of the cost function w.r.t weight
    bias_derivative = 0 # partial derivative of the cost function w.r.t bias
    observations = len(number_bedrooms) # total number of rows in the dataset

    for i in range(observations):
        # Calculate partial derivatives
        # -2x(y - (Wx + B))
        weight_derivative += -2*number_bedrooms[i] * (price[i] - (weight*number_bedrooms[i] + bias))

        # -2(y - (Wx + B))
        bias_derivative += -2*(price[i] - (weight*number_bedrooms[i] + bias))

    # We subtract because the derivatives point in direction of steepest ascent in Gradient Descent
    weight -= (weight_derivative / observations) * learning_rate
    bias -= (bias_derivative / observations) * learning_rate

    return weight, bias
```

Training:-

Training a model is the process of iteratively improving your prediction equation by looping through the dataset multiple times, each time updating the weight and bias values in the direction indicated by the slope of the cost function (gradient). Training is complete when

we reach an acceptable error threshold, or when subsequent training iterations fail to reduce our cost.

Before training we need to initialize our weights, set learning rate and number of iterations, and prepare to log our progress over each iteration.

Python Function to implement Training Process:-

```
[35] def train(number_bedrooms, price, weight, bias, learning_rate, iters):
        cost_history = []
        weight_history = []
        bias_history = []

        for i in range(iters):
            weight,bias = update_weights(number_bedrooms, price, weight, bias, learning_rate)
            weight_history.append(weight)
            bias_history.append(bias)

            #Calculate cost for debugging purposes
            cost = cost_function(number_bedrooms, price, weight, bias)
            cost_history.append(cost)

        return weight_history, bias_history, cost_history
```

Python code to initialize the parameters:-

```
[68] #initializing parameters
     import numpy as np
     weight=float(np.random.randint(low=0,high=10,size=1))
     bias = 10
     alpha = 0.001
```

Model evaluation:-

The final step of the implementation of our regression model is its evaluation. If our model is learning the parameters W and B efficiently then we should see our cost function decreasing with each training loop as given in the below image.

Python code to train our model:-

```
[69] weight_history,bias_history,cost_history = train(x_train,y_train,weight,bias,alpha,10)
     for i in range(len(a)):
       print("iteration:{} --- weight:{:.2f} --- bias:{:.4f} --- cost:{}".format(i,weight_history[i],bias_history[i],
                                                                                 cost_history[i]))

 iteration:0 --- weight:8.04 --- bias:10.0110 --- cost:37.375310249999984
 iteration:1 --- weight:8.08 --- bias:10.0218 --- cost:35.81840924924779
 iteration:2 --- weight:8.11 --- bias:10.0323 --- cost:34.32646848354798
 iteration:3 --- weight:8.15 --- bias:10.0426 --- cost:32.89677745972807
 iteration:4 --- weight:8.19 --- bias:10.0527 --- cost:31.526738781158322
 iteration:5 --- weight:8.22 --- bias:10.0626 --- cost:30.213863428747615
 iteration:6 --- weight:8.26 --- bias:10.0722 --- cost:28.955766238841946
 iteration:7 --- weight:8.29 --- bias:10.0817 --- cost:27.750161569809844
 iteration:8 --- weight:8.32 --- bias:10.0909 --- cost:26.594859149441334
 iteration:9 --- weight:8.36 --- bias:10.0999 --- cost:25.487760095615954
```

Predicting price of houses for new values of the number of bedrooms:-

Now let's define a function that outputs the price of a 5 and 7 bedroom house respectively.

This function will take the new updated values of the weight and bias parameters. The new values of the parameters are the last entries in the weight_history and bias_history variables defined in the above given piece of code.

Python code to print updated values of weight and bias:-

```
[72] updated_weight = weight_history[len(weight_history)-1]# the last index of our weight_history
     updated_bias = bias_history[len(bias_history)-1]# the last index of our bias_history
     print(updated_weight)
     print(updated_bias)
```

```
8.355057488468875
10.099934580672457
```

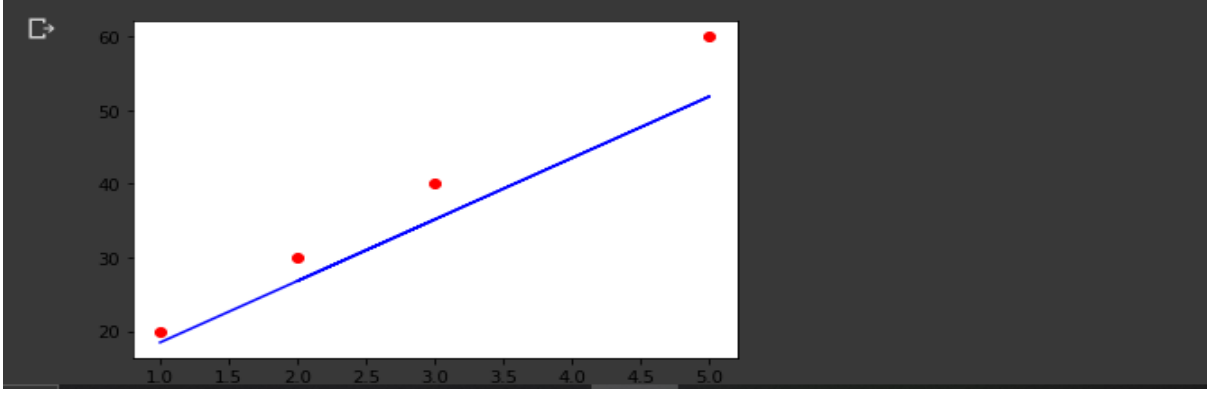
Python code to predict new values:-

```
[73] # predicting new values
     def predict_price(number_bedrooms, weight, bias):
         return weight*number_bedrooms + bias

     print(predict_price(5,updated_weight,updated_bias))
     print(predict_price(7,updated_weight,updated_bias))
```

```
51.87522202301683
68.58533699995458
```

Python code to graph the difference between the actual and predicted values on thtraining set:-

```
import matplotlib.pyplot as plt
y_predicted = []
for i in range(len(x_train)):
  y_predicted.append(predict_price(x_train[i],updated_weight,updated_bias))

plt.scatter(x_train,y_train,color="red")
plt.plot(x_train,y_predicted,color="blue")
plt.show()
```



What can we infer from the above graph and training process?

The first thing that we should notice from the above graph is that we are having significant error in our predicted values, although the error is relatively less.

Some reasons for this type of output is as follows:

1) Our training data is very less. With more data we could have trained the model better.
2) Initialization of parameters affect our training process significantly. Right initialization will yield more accurate results.
3) Our data is not normalized. Normalization of data (i.e. making both the input as well as output data in the same range) is very important to yield good results.

Future improvements that will be brought in this article:-

In the future improvement of this article I will deal with the topics of initialization and normalization of the data to improve the accuracy of our model. I will focus on taking real world large datasets for the purpose of training the model.

NOTE:- *For the complete source code, here is my github repo link:*

https://github.com/Avhijit-codeboy/Simple-Linear-Regression