The AI-ML and Data Science Club of IITH

Linear Regression

Contents

Compiled by : Tejadhith and Divyanshu Bhatt

1. What is Linear Regression?

Linear Regression is one of the **supervised learning**[1] models of Machine Learning which tries to find the **linear** relationship between the input/feature variable X and the target variable y. The model's output is a continuous value hence called **regression** hence the model is called **linear regression**.

Let's look into a single feature/input example to get an idea of it. The model is trained with the noisy data point observed in the V-I characteristics of the resistor, which has a resistance of 3Ω. Model fits the best line as shown in following figure. Hence the trained model can predict the value of current based on the voltage provided by mapping it to the best fit line. It implies that ML models are not 100% accurate as the figure shows slope = 3.04 and intercept = -0.3, where the expected values are 3 and 0, respectively.

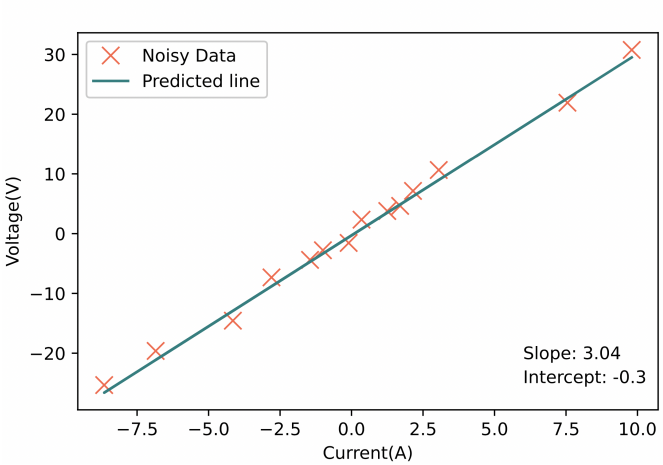


Figure 1. V-I Characteristics of a Resistor

2. How Linear regression works?

The model finds a linear relationship between X and y based on the training data, and the output is given by the trained model based on the input mapping to the best fit line.
The best fit line is defined such that the sum of squares of distance between the line and the target variable(y) is minimum.
We define a linear function called as **Hypothesis**, then a **Cost/Loss function** that gives the error in the Hypothesis while an algorithm is used to reduce the Cost function.

2.1 Hypothesis

X and y are defined as follows (N is the number of features, and M is the number of data points)

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_M \end{bmatrix} \in \mathbb{R}^{M \times N}$$

$$x_i = \begin{bmatrix} x_i^{(1)} & x_i^{(2)} & \cdots & x_i^{(N)} \end{bmatrix} \in \mathbb{R}^{1 \times N}$$

$$y_i \in \mathbb{R}$$

[1] Machines are trained with the data that is **tagged** with the correct output, which is used to predict the output

where x_i and y_i denote the i^{th} data point and $x_i^{(j)}$ denotes the j^{th} feature of the i^{th} data point.

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_N \end{bmatrix}$$

θ is defined as Weights/Parameter/Regression Coefficients
Hypothesis of Linear Regression is defined as,

$$\hat{y}_i = \langle x_i, \theta \rangle + b_i$$

where $\hat{y} \in \mathbb{R}$ is the predicted target variable, $b_i \in \mathbb{R}$ is called bias and $\langle \cdot, \cdot \rangle$ represents dot product between two vectors.
Instead of explicitly adding the bias, we can include it inside our θ parameters, i.e. we can add 1 as the last element in the vector x_i and b as the last element θ. So $x_i \in \mathbb{R}^{1\times(N+1)}$ and $\theta \in \mathbb{R}^{N+1}$ and our equation changes as

$$\hat{y}_i = \langle x_i, \theta \rangle$$

3. Cost/Loss Function

Cost Function returns the **error between the predicted and the actual outcome**.
Mean Squared Error(MSE) is the Cost function $L(\theta)$ in Linear Regression.

$$\begin{aligned} L(\theta) &= \frac{1}{2M}\|\hat{y} - y\|^2 \\ &= \frac{1}{2M}\sum_{i=1}^{M}(\hat{y}_i - y_i)^2 \end{aligned}$$

Where M is the number of samples

$$L(\theta) = \frac{1}{2M}\sum_{i=1}^{M}(x_i\theta - y_i)^2$$

3.1 Why Mean Squared Error as Cost function?

We can consider our desired output and the actual output by the model as two points in the space. So we want our actual output to be as close as possible to the desired output. MSE calculates the distance between the desired output and the actual output. Hence, minimising the loss function (MSE) is the same as making our prediction more closer to the actual value.

4. Algorithms to minimize the Cost Function

One of the algorithms used in Linear Regression to minimize the Cost function is Gradient Descent Algorithm.

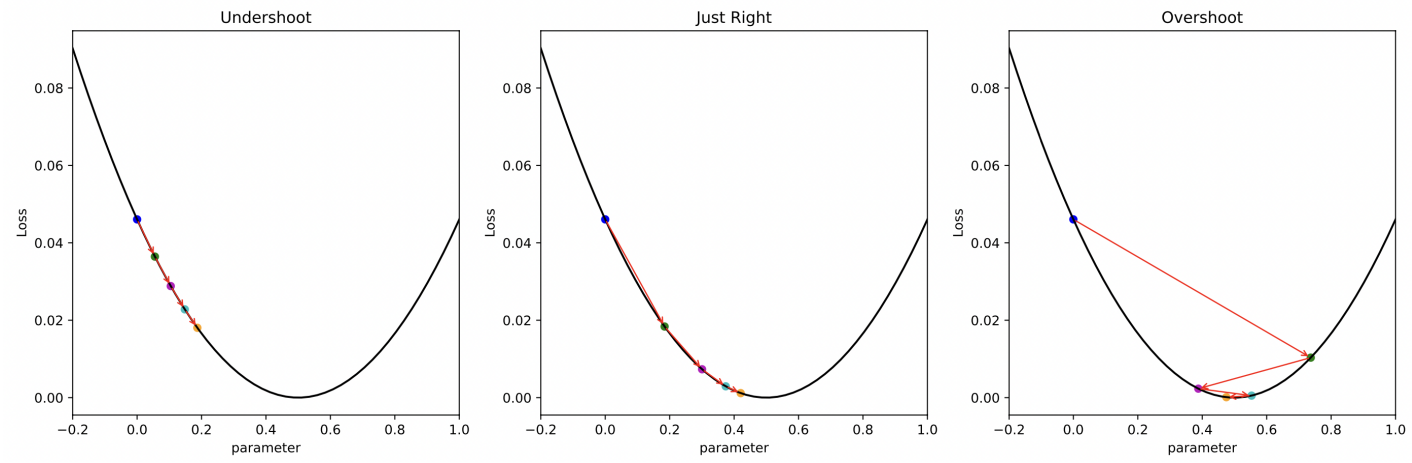
4.1 Gradient Descent

In Gradient Descent, we move in the opposite direction of the slope to attain the global minimum, which is attained by finding the gradient of the Cost Function concerning each parameter and then updating the parameter iteratively.

$$\theta^{(j)} = \theta^{(j)} - \alpha\frac{\partial}{\partial\theta^{(j)}}L(\theta)$$

where α is the Learning rate. The negative gradient of the loss function approximately points toward the minimum of the function. Now, the learning rate decides how big the step should be taken in that direction. Setting the learning rate too high might skip the point of minimum, i.e. our gradient overshoots,

while setting the learning rate low increases the time complexity because we have to train our model for more epochs to reach the minimum.



Differentiating the loss function,

$$\begin{aligned}\frac{\partial}{\partial\theta^{(j)}}L(\theta) &= \frac{\partial}{\partial\theta^{(j)}}\left(\frac{1}{2M}\sum_{i=1}^{M}(x_i\theta - y_i)^2\right)\\ &= \frac{2}{2M}\sum_{i=1}^{M}(x_i\theta - y_i)\ \frac{\partial}{\partial\theta^{(j)}}(x_i\theta)\\ &= \frac{1}{M}\sum_{i=1}^{M}(x_i\theta - y_i)\ (x_i^{(j)})\\ &= \frac{1}{M}\langle X\theta - y, x^{(j)}\rangle\end{aligned}$$

Hence,

$$\implies \theta^{(j)} = \theta^{(j)} - \frac{\alpha}{M}\langle X\theta - y, (x^{(j)})\rangle$$

For multiple samples,

$$\implies \theta^{(j)} = \theta^{(j)} - \frac{\alpha}{M}\sum_{i=1}^{M}(x_i\theta - y_i)\ (x_i^{(j)})$$

4.2 Normal Equation

One of the other algorithms to minimize the Cost function is **Normal Equation**. Instead of using the iterative approach of gradient descent, we can directly make the partial derivative of the cost function concerning θ zero

$$\begin{aligned}\frac{d\mathcal{L}(\theta)}{d\theta} &= \frac{d}{d\theta}\left(\frac{1}{2M}\sum_{i=1}^{M}\|X\theta - y\|^2\right)\\ &= \frac{1}{2M}\frac{d}{d\theta}\Big((y - X\theta)^T(y - X\theta)\Big)\\ &= \frac{1}{2M}\frac{d}{d\theta}\Big(y^Ty - y^TX\theta - \theta^TX^Ty + \theta^TX^TX\theta\Big)\\ &= \frac{1}{2M}(0 - y^TX - y^TX + 2\theta^TX^TX)\\ &= \frac{1}{M}(\theta^TX^TX - y^TX)\end{aligned}$$

Now, making the partial derivative equal to 0

$$\implies \frac{\partial L}{\partial \theta} = 0$$
$$\implies \frac{1}{M}(\theta^T X^T X - y^T X) = 0$$
$$\implies \theta^T X^T X = y^T X$$
$$\implies X^T X \theta = X^T y$$
$$\implies \theta = (X^T X)^{-1} X^T y$$

4.3 Gradient Descent v/s Normal Equation

Gradient descent is an iterative approach, whereas Normal Equation is an analytical approach. For a large data set, one should always use gradient descent because the time complexity of Gradient descent is much low as compared to the Normal Equation. For small data sets (when the no. of features is not too high), there is not much difference between the two, so Normal Equation can be used because there is no need for setting up any hyperparameters like learning rate in it.

$$\text{Gradient Descent Complexity} \longrightarrow O(knm)$$
$$\text{Normal Equation Complexity} \longrightarrow O(n^3 + mn^2)$$

5. Limitations and Assumptions

Assumptions made while making the Linear Regression model :
1) Linearity : The relationship between X and Y is linear.
2) Independence : Input/Features are independent of each other i.e co-relativity = 0
3) Homoscedasticity : The variance of residual is the same for any value of X.
4) Normality: For any fixed value of X, Y is normally distributed.

Limitation of Linear Regression model :
1) Multi Co-Linearity : Multi Co-Linearity should be removed using Dimensionality reduction techniques.
2) Outliers : Data point that has high error and it influence's the best fine the most.
3) Overfitting :Fits exactly against it's training data such that it doesn't perform well in test data.

6. Questions

1) In a simple linear regression model with n features which is trained over m data points. If n > m, what would be the case?
2) Is Linear Regression Generative (or) Discriminative ?
3) Explain the Probabilistic interpretation of Linear Regression ?
4) Why don't we use absolute values of difference as Cost Function ?
5) Why is Normalization required for an data set ?
6) What is Lasso and Ridge Regression? How is it different from Linear Regression?
7) How is r^2 score calculated?
8) In Linear Regression, Is Mean Squared Error function Convex (or) Non-Convex?

7. Linear Regression with python

A Linear Regression model is built from scratch by only using `numpy`,`pandas` and `matplotlib` libraries.

In [1]:

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

In [2]:

```
def loading_data():
    # reading data
    train_set = pd.read_csv('sample_data/california_housing_train.csv')
    test_set = pd.read_csv('sample_data/california_housing_test.csv')

    # data preprocessing
    train_y = train_set["median_house_value"]
    del train_set["median_house_value"]
    del train_set["longitude"]
    del train_set["latitude"]

    test_y = test_set["median_house_value"]
    del test_set["median_house_value"]
    del test_set["longitude"]
    del test_set["latitude"]

    # normalising data
    train_set_normal = (train_set-train_set.mean())/train_set.std()
    test_set_normal = (test_set-test_set.mean())/test_set.std()

    # converting data set to array
    train_set = train_set_normal.to_numpy()
    test_set = test_set_normal.to_numpy()

    train_y = train_y.to_numpy()
    test_y = test_y.to_numpy()

    return (train_set, train_y), (test_set, test_y)
```

In [3]:

```
(train_X, train_y), (test_X, test_y) = loading_data()
```

In [4]:

```
class LinearRegressor(object):
    def __init__(self):
        pass

    def fit_SGD(self, train_X, train_y, epochs=10, lr=0.01):
        self.num_datapoints = train_X.shape[0]
        self.num_features = train_X.shape[1]

        self.weight = np.random.randn(self.num_features, 1)
        self.bias = np.random.randn()

        self.train_loss = []  # contains loss of each epoch

        self.train_loss.append(self.train_test_loss(
            data_X=train_X, data_y=train_y))

        for j in range(epochs):

            nabla_w, nabla_b = self.update_parameters(train_X, train_y)

            self.weight = self.weight - (lr/self.num_datapoints)*(nabla_w)
            self.bias = self.bias - (lr/self.num_datapoints)*(nabla_b)

            self.train_loss.append(self.train_test_loss
                                   (data_X=train_X, data_y=train_y))
```

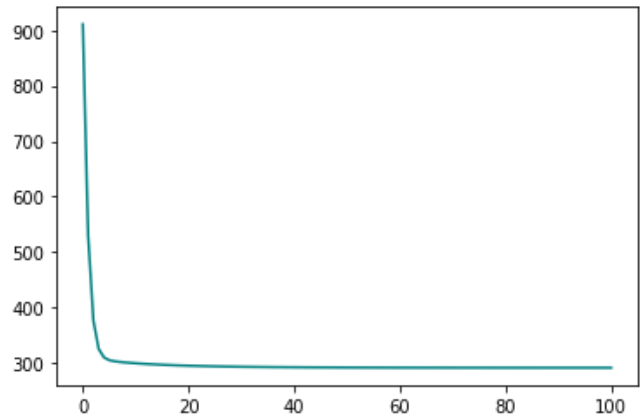
```
        self.plot_loss()

    def fit_NE(self, train_X, train_y):
        self. weight = np.linalg.pinv(
            (train_X.T) @ train_X) @ ((train_X.T) @ train_y)

    def update_parameters(self, X, y):
        delta = np.matmul(X, self.weight) + self.bias
        delta = delta.reshape(-1)  # making shape of y and delta same
        delta = delta - y

        nabla_b = np.sum(delta)  # same as dot product with ones vector
        # reshaping it as a column vector
        nabla_w = np.matmul(delta, X).reshape(-1, 1)

        return nabla_w, nabla_b

    def train_test_loss(self, data_X, data_y):
        predicted_output = self.prediction(data_X)
        return self.loss_function(predicted_output, data_y)

    def prediction(self, data):
        return np.matmul(data, self.weight) + self.bias

    def loss_function(self, predicted_output, acutal_output):
        # Mean Square Error
        predicted_output = predicted_output.reshape(-1)
        return (1.0/(2*self.num_datapoints))*(np.linalg.norm(predicted_output - acutal_o
utput))

    def plot_loss(self):
        x = np.arange(0, len(self.train_loss), 1)
        plt.plot(x, self.train_loss, color="teal")
        plt.show()

    def predict(self, test_X, test_y):
        predicted_output = self.prediction(test_X).reshape(-1)
        self.r2score(predicted_output, test_y)

    def r2score(self, predicted_output, actual_output):
        rss = np.linalg.norm(actual_output - predicted_output)**2
        tss = np.linalg.norm(actual_output - np.mean(actual_output))**2

        print(f"r2score: {(1-(rss/tss))*100}")
```

In [5]:

```
lig = LinearRegressor()
```

In [6]:

```
lig.fit_SGD(train_X, train_y, epochs=100, lr=0.5)
```



In [7]:

```
lig.predict(test_X, test_y)
```

```
r2score: 54.4758007984627
```

In [8]:

```
lig.fit_NE(train_X, train_y)
```

In [9]:

```
lig.predict(test_X, test_y)
```

```
r2score: 54.46683334777153
```

In [10]:

```
from sklearn.linear_model import LinearRegression
```

In [11]:

```
sk_lig = LinearRegression()
sk_lig.fit(train_X, train_y)
print(sk_lig.score(test_X, test_y)*100)
```

```
54.466833347771406
```