MACHINE LEARNING LAB

ASSIGNMENT - 4

T.Siva Teja

AP20110010813

CSE-L

1. What is Gradient descent, write short note on it.

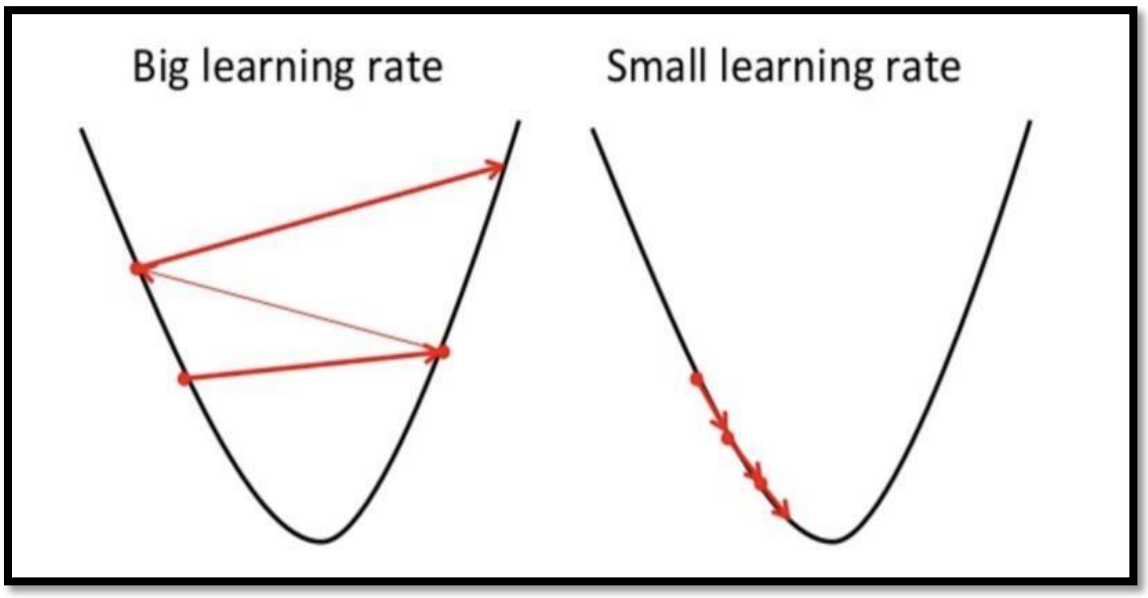
Gradient descent (GD) is an iterative first-order optimisation algorithm used to find a local minimum/maximum of a given function. This method is commonly used in machine learning (ML) and deep learning (DL) to minimise a cost/loss function (e.g., in a linear regression).

A gradient simply measures the change in all weights about the change in error. You can also think of a gradient as the slope of a function. The higher the gradient, the steeper the slope and the faster a model can learn. But if the slope is zero, the model stops learning. In mathematical terms, a gradient is a partial derivative with respect to its inputs.

However, its use is not limited to ML/DL only, it's being widely used also in areas like:

- control engineering (robotics, chemical, etc.)
- computer games
- mechanical engineering

For the gradient descent algorithm to reach the local minimum we must set the learning rate to an appropriate value, which is neither too low nor too high. This is important because if the steps it takes are too big, it may not reach the local minimum because it bounces back and forth between the convex function of gradient descent (see left image below). If we set the learning rate to a very small value, gradient descent will eventually reach the local minimum but that may take a while (see the right image).



There are three popular types of gradient descent that mainly differ in the amount of data they use:

BATCH GRADIENT DESCENT

Batch gradient descent, also called vanilla gradient descent, calculates the error for each example within the training dataset, but only after all training examples have been evaluated does the model get updated.

STOCHASTIC GRADIENT DESCENT

By contrast, stochastic gradient descent (SGD) does this for each training example within the dataset, meaning it updates the parameters for each training example one by one. Depending on the problem, this can make SGD faster than batch gradient descent.

MINI-BATCH GRADIENT DESCENT

Mini-batch gradient descent is the go-to method since it's a combination of the concepts of SGD and batch gradient descent. It simply splits the training dataset into small batches and performs an update for each of those batches.

Implement Linear regression using Gradient descent.

CODE: -

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


class Linear_Regression:
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y
        self.b = [0, 0]


    def update_coeffs(self, learning_rate):
        Y_pred = self.predict()
        Y = self.Y
        m = len(Y)
        self.b[0] = self.b[0] - (learning_rate * ((1/m) *
                            np.sum(Y_pred - Y)))


        self.b[1] = self.b[1] - (learning_rate * ((1/m) *
                            np.sum((Y_pred - Y) * self.X)))


    def predict(self, X=[]):
```

```
    Y_pred = np.array([])
    if not X: X = self.X
    b = self.b
    for x in X:
        Y_pred = np.append(Y_pred, b[0] + (b[1] * x))

    return Y_pred

def get_current_accuracy(self, Y_pred):
    p, e = Y_pred, self.Y
    n = len(Y_pred)
    return 1-sum(
        [
            abs(p[i]-e[i])/e[i]
            for i in range(n)
            if e[i] != 0]
    )/n
#def predict(self, b, yi):

def compute_cost(self, Y_pred):
    m = len(self.Y)
    J = (1 / 2*m) * (np.sum(Y_pred - self.Y)**2)
```

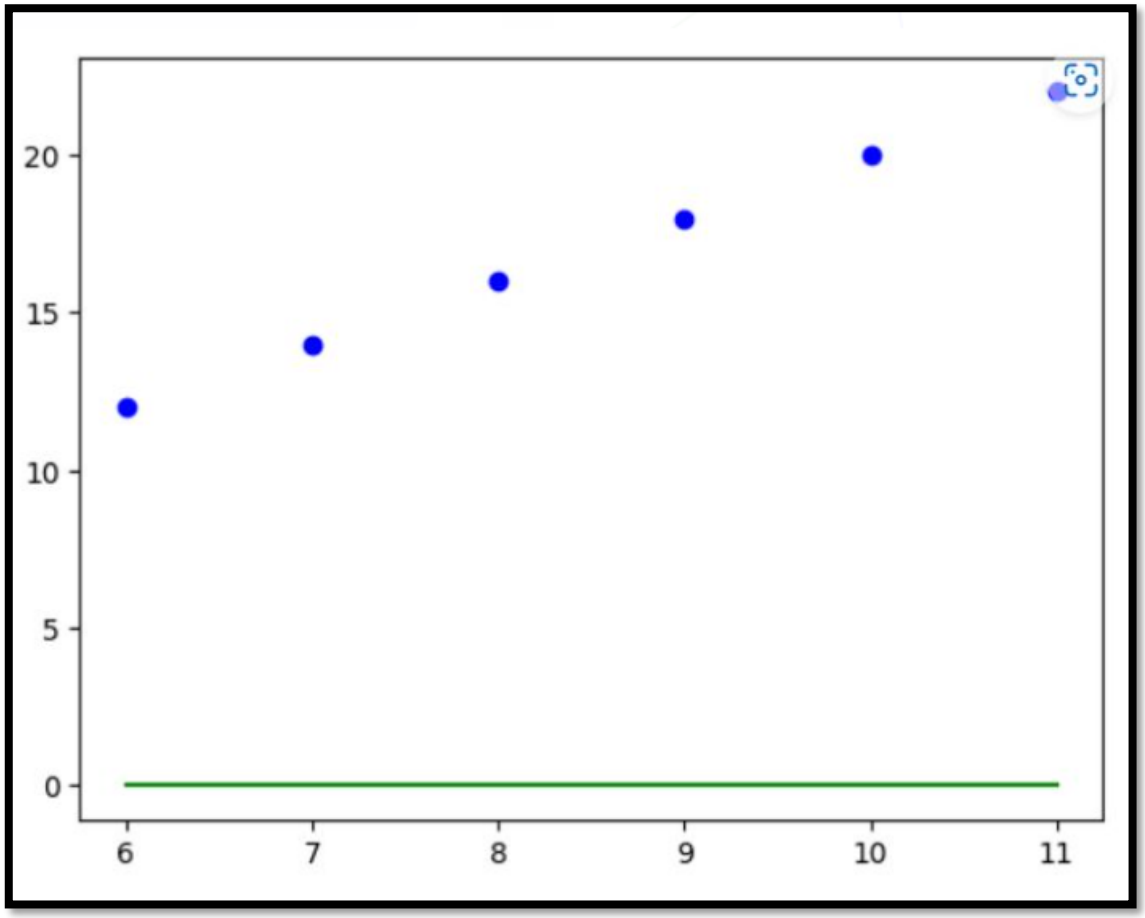
```
        return J

    def plot_best_fit(self, Y_pred, fig):
            f = plt.figure(fig)
            plt.scatter(self.X, self.Y, color='b')
            plt.plot(self.X, Y_pred, color='g')
            f.show()



def main():
    a=list(map(int,input().split()))
    b=list(map(int,input().split()))
    # observations / data
    X = np.array(a)
    Y = np.array(b)


    regressor = Linear_Regression(X, Y)


    iterations = 0
    steps = 100
    learning_rate = 0.01
    costs = []
```

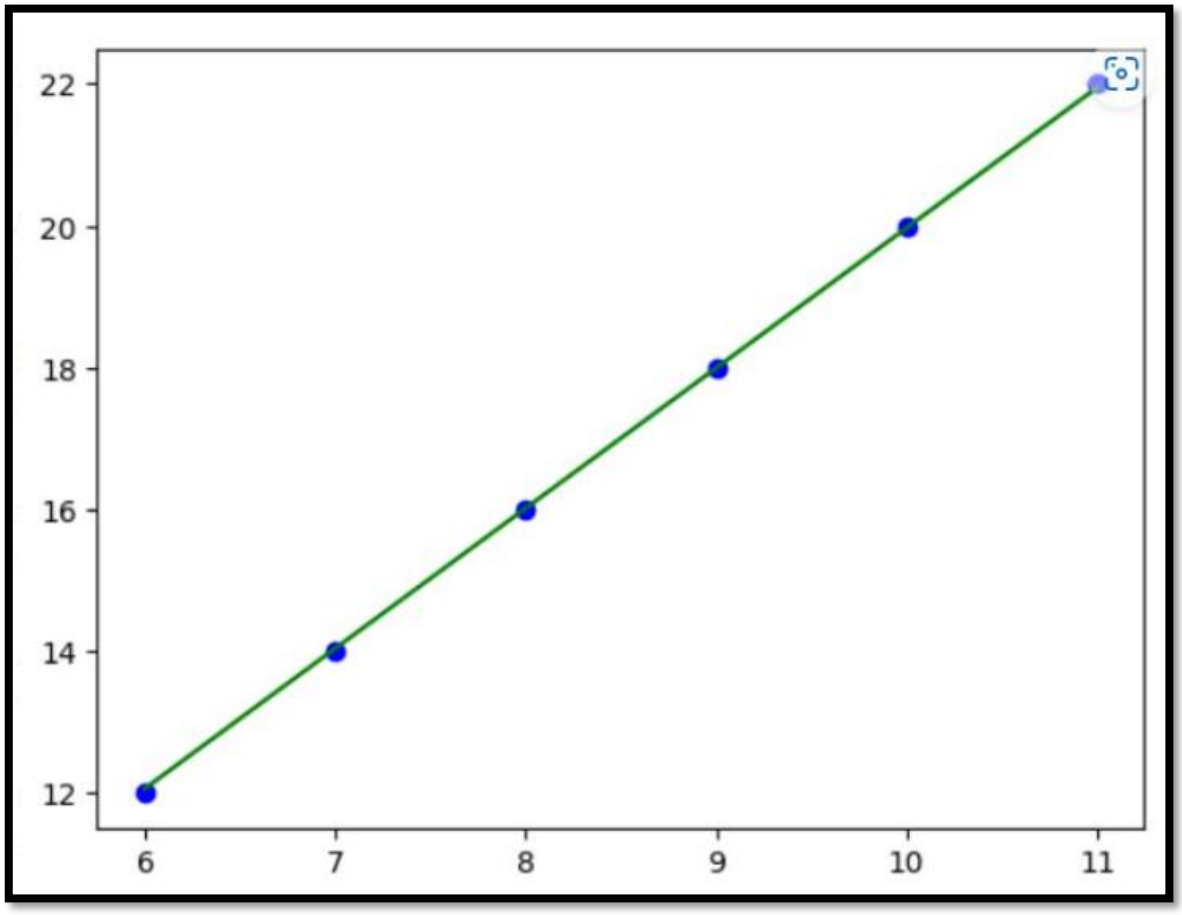
```
#original best-fit  line
Y_pred = regressor.predict()
regressor.plot_best_fit(Y_pred, 'Initial Best Fit Line')



while 1:
    Y_pred = regressor.predict()
    cost = regressor.compute_cost(Y_pred)
    costs.append(cost)
    regressor.update_coeffs(learning_rate)

    iterations += 1
    if iterations % steps == 0:
        print(iterations, "epochs elapsed")
        print("Current accuracy is :",
              regressor.get_current_accuracy(Y_pred))

        stop = input("Do you want to stop (y/*)??")
        if stop == "y":
            break
```

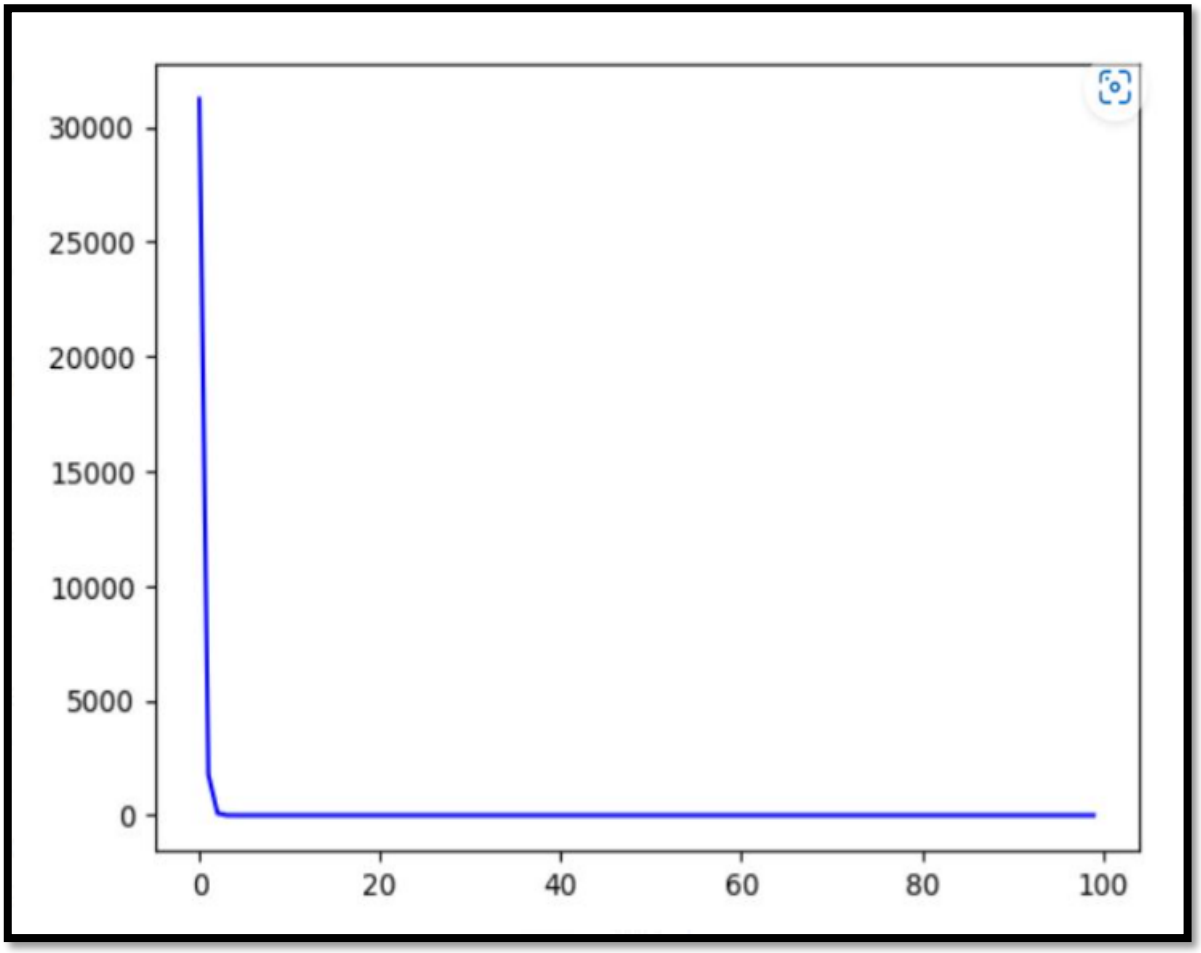
```
    #final best-fit line
    regressor.plot_best_fit(Y_pred, 'Final Best Fit Line')


    #plot to verify cost function decreases
    h = plt.figure('Verification')
    plt.plot(range(iterations), costs, color='b')
    h.show()


    # if user wants to predict using the regressor:
    regressor.predict([i for i in range(10)])


if__name__== '__main__':
    main()
    print("\nT.Siva Teja, AP20110010813")
```

Screenshots of OUTPUT: -

```
6 7 8 9 10 11 12
12 14 16 18 20 22 24
<ipython-input-11-8175cf2463be>:49: UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
show the figure.
  f.show()
100 epochs elapsed
Current accuracy is : 0.9976353032618925
Do you want to stop (y/*)??y

T.Siva Teja, AP20110010813
<ipython-input-11-8175cf2463be>:49: UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
show the figure.
  f.show()
<ipython-input-11-8175cf2463be>:93: UserWarning: Matplotlib is currently using
module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so cannot
show the figure.
  h.show()
```


20
15
10
5
0
6
7
8
9
10
11


22
20
18
16
14
12
6
7
8
9
10
11

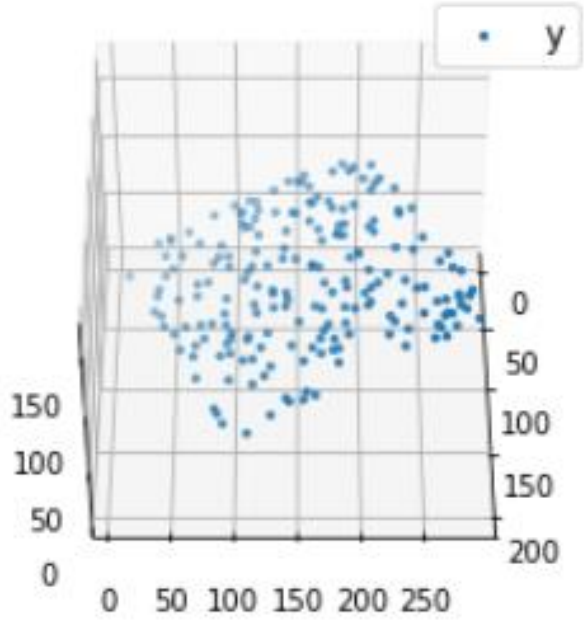

30000
25000
20000
15000
10000
5000
0
0
20
40
60
80
100

2. Write a code in python to implement Multiple Linear regression.

CODE: -

```
import numpy as np
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

def generate_dataset(n):
    x = []
    y = []
    random_x1 = np.random.rand()
    random_x2 = np.random.rand()
    for i in range(n):
        x1 = i
        x2 = i/2 + np.random.rand()*n
        x.append([1, x1, x2])
        y.append(random_x1 * x1 + random_x2 * x2 + 1)
    return np.array(x), np.array(y)

x, y = generate_dataset(200)

mpl.rcParams['legend.fontsize'] = 12

fig = plt.figure()
ax = fig.add_subplot(projection ='3d')

ax.scatter(x[:, 1], x[:, 2], y, label ='y', s = 5)
ax.legend()
ax.view_init(45, 0)

plt.show()
print("\nT.Siva Teja, AP20110010813")
```

Screenshot of OUTPUT: -



T.Siva Teja, AP20110010813

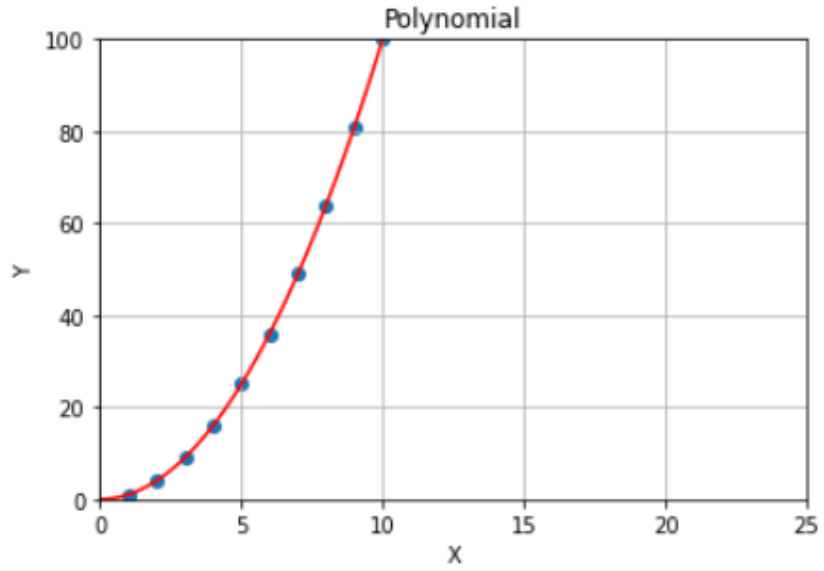
3. Write a code in python to implement Polynomial Regression.

CODE: -

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

a=list(map(int,input().split()))
b=list(map(int,input().split()))
   # observations / data
X = np.array(a)
Y = np.array(b)

# Train Algorithm (Polynomial)
degree = 2
poly_fit = np.poly1d(np.polyfit(X,Y, degree))

# Plot data
xx = np.linspace(0, 26, 100)
plt.plot(xx, poly_fit(xx), c='r',linestyle='-')
plt.title('Polynomial')
plt.xlabel('X')
plt.ylabel('Y')
plt.axis([0, 25, 0, 100])
plt.grid(True)
plt.scatter(X, Y)
plt.show()

# Predict price
print("Enter the price to Predict:- ")
n = int(input())
print(poly_fit(n))
print("\nT.Siva Teja, AP20110010813")
```

Screenshot of OUTPUT: -

```
1 2 3 4 5 6 7 8 9 10
 1 4 9 16 25 36 49 64 81 100
```




```
Enter the price to Predict:-
5
24.999999999999982

T.Siva Teja, AP20110010813
```