LINK - https://colab.research.google.com/drive/1R4ivrDTbwv7FY3_yW-QEinN-hPIPXkxC?usp=sharing

Problem [Python] Gradient descent learning in Python:

a. Create class NeuralNetwork(): that creates a single neuron with a linear activation, train it using gradient descent learning. This class should have the following function:

def **init**(self, learning_r): that initializes a 3x1 weight vector randomly and initializes the learning rate to learning_r. Also, it creates a history variable that saves the weights and the training cost after each epoch (i.e., iteration).

def sigmoid(self, x): that takes an input x, and applies the sigmoid function to return:

def forward_propagation(self, inputs): that performs forward propagation by multiplying the inputs by the neuron weights, uses sigmoid activation function and then generates the output.

def train(self, inputs_train, labels_train, num_train_iterations): that performs the gradient descent learning rule for num_train_iterations times using the inputs and labels.

```
import numpy as np

class NeuralNetwork:
    def __init__(self, learning_r):
        self.weights = np.random.randn(3, 1)
        self.learning_rate = learning_r
        self.history = {'weights': [], 'cost': []}

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def forward_propagation(self, inputs):
        z = np.dot(inputs, self.weights)
        return self.sigmoid(z)

    def train(self, inputs, train_labels, train_num_iterations):
        for iteration in range(train_num_iterations):
            predictions = self.forward_propagation(inputs)

            error = predictions - train_labels

            sigmoid_derivative = predictions * (1 - predictions)
            gradient = np.dot(inputs.T, error * sigmoid_derivative) / inputs.shap

            self.weights -= self.learning_rate * gradient

            cost = np.mean((predictions - train_labels) ** 2)

            self.history['weights'].append(self.weights.copy())
            self.history['cost'].append(cost)
```

b. Use the gradient descent rule to train a single neuron on the datapoints given below:

i) Create an np array of a shape 10x2 that contains the inputs, and another array with a shape 10x1 that contains the labels.

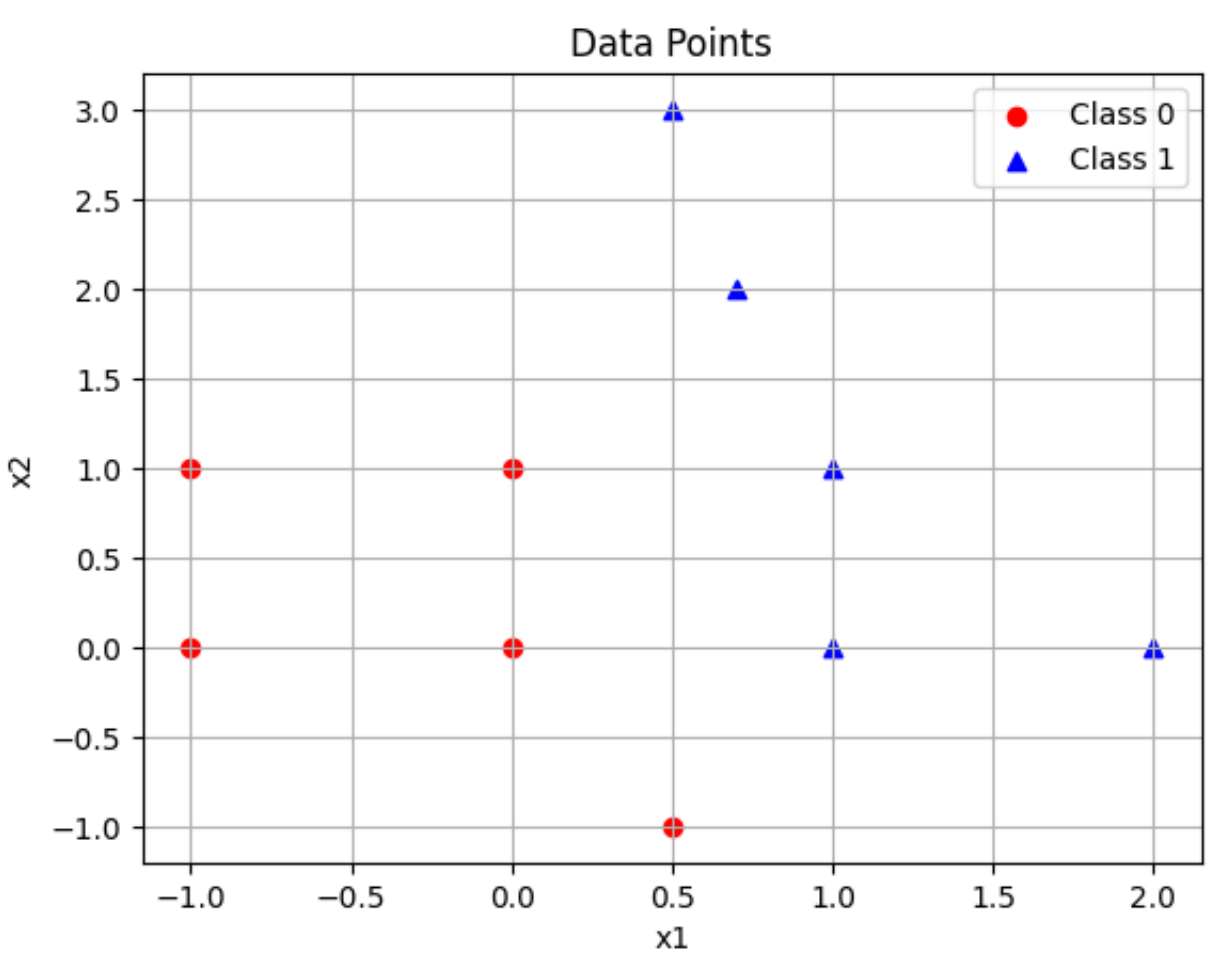
```
inputs = np.array([
    [1, 1],
    [1, 0],
    [0, 1],
    [0.5, -1],
    [0.5, 3],
    [0.7, 2],
    [-1, 0],
    [-1, 1],
    [2, 0],
    [0, 0]
])

labels = np.array([[1], [1], [0], [0], [1], [1], [0], [0], [1], [0]])
```

iii) Add the bias to the inputs array to have a 10x3 shape.

```
bias = np.ones((inputs.shape[0], 1))
inputs_with_bias = np.hstack((inputs, bias))
```

ii) Plot the data points with different markers for each class

```
import matplotlib.pyplot as plt

class_0 = inputs[labels.flatten() == 0]
class_1 = inputs[labels.flatten() == 1]

plt.scatter(class_0[:, 0], class_0[:, 1], marker='o', label='Class 0', color='red
plt.scatter(class_1[:, 0], class_1[:, 1], marker='^', label='Class 1', color='blu
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.title('Data Points')
plt.grid(True)
plt.show()
```

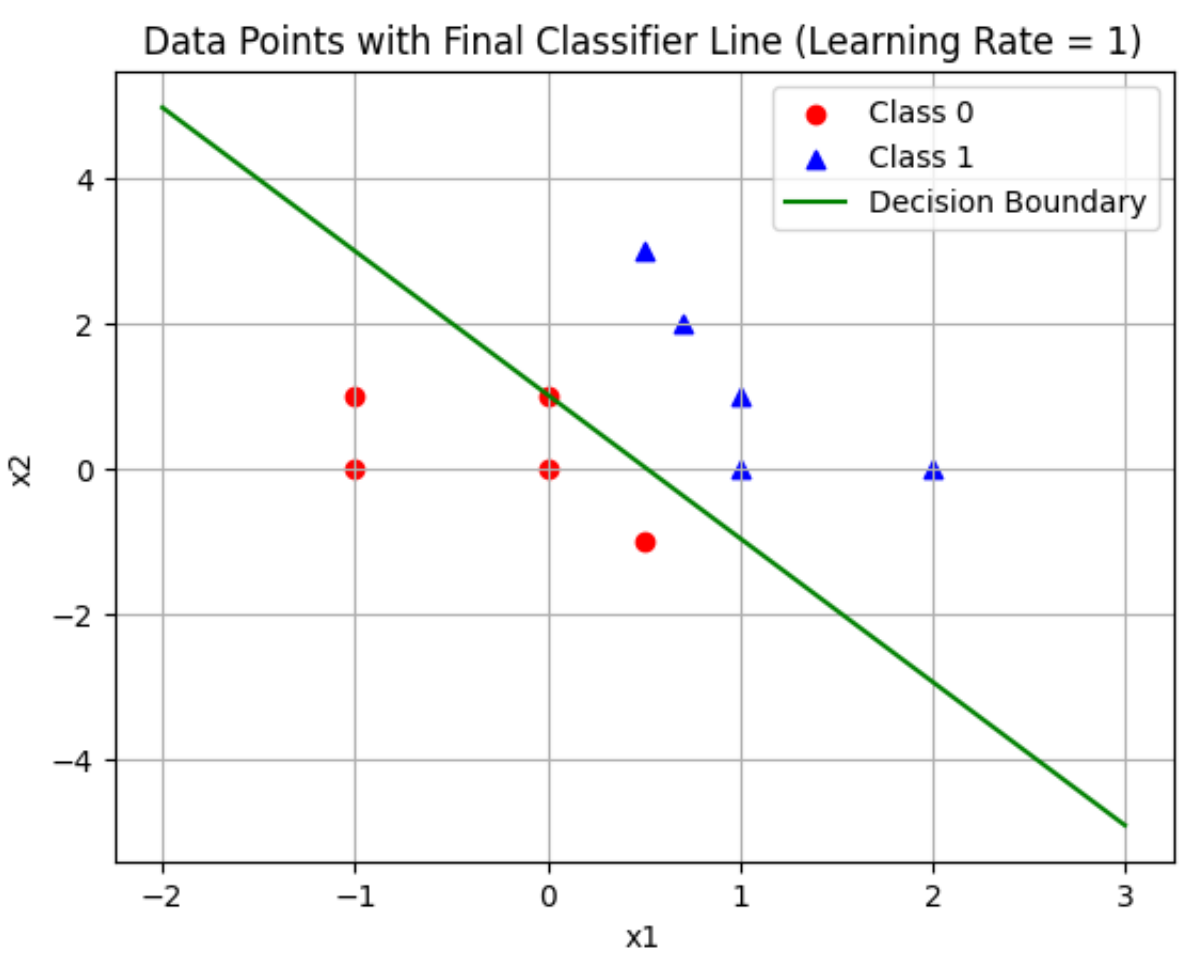


iv) Create the network with one neuron using the class NeuralNetwork() with learning rate of 1 then train it using train (inputs, labels, 50) function.

```
nn = NeuralNetwork(learning_r=1)

nn.train(inputs_with_bias, labels, train_num_iterations=50)
```

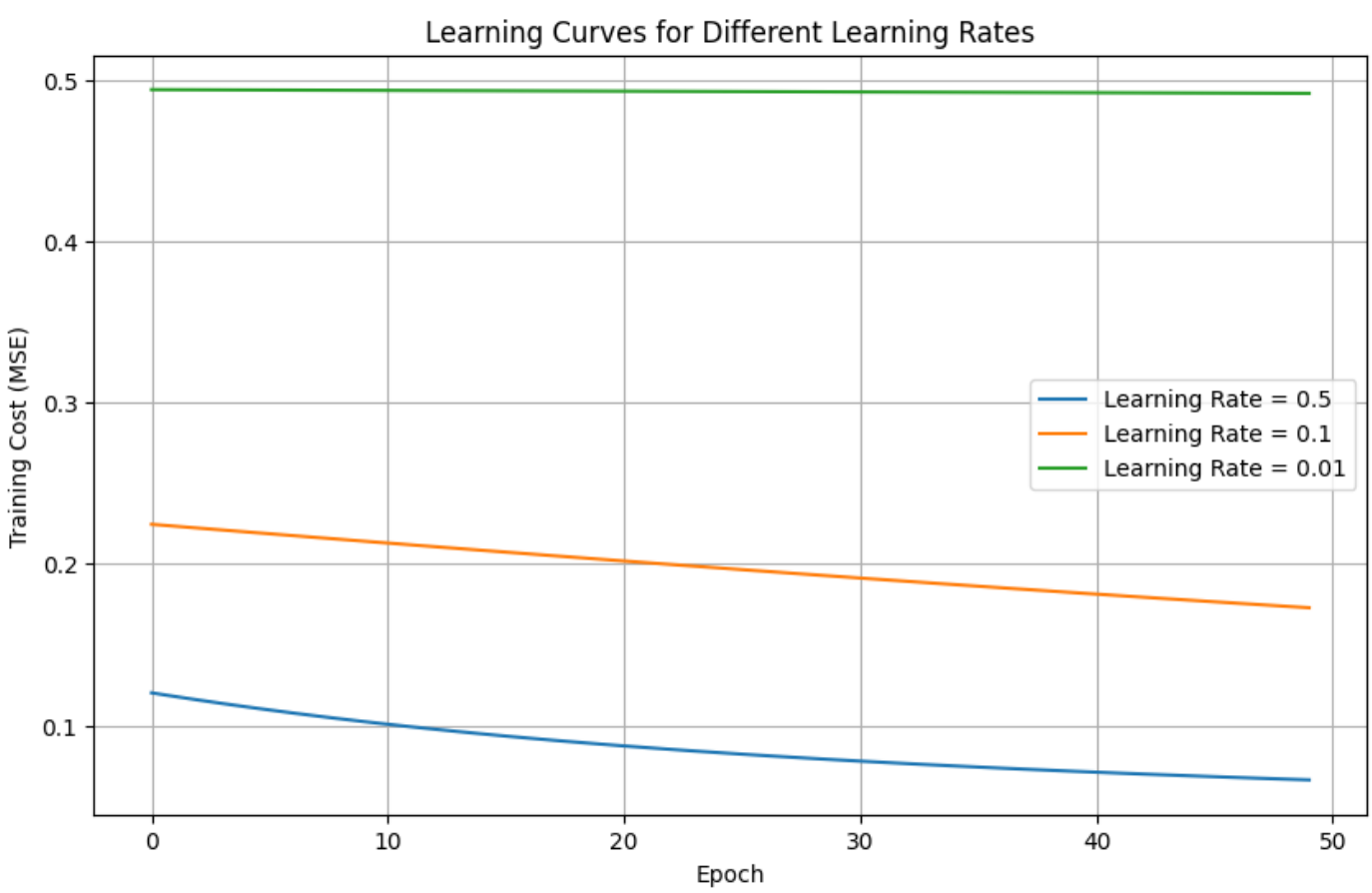
c. Use the trained weights and plot the final classifier line.

```
final_weights = nn.history['weights'][-1]
w1, w2, b = final_weights[0, 0], final_weights[1, 0], final_weights[2, 0]

x1_vals = np.linspace(min(inputs[:, 0]) - 1, max(inputs[:, 0]) + 1, 100)
x2_vals = -(w1 / w2) * x1_vals - (b / w2)

plt.scatter(class_0[:, 0], class_0[:, 1], marker='o', label='Class 0', color='red
plt.scatter(class_1[:, 0], class_1[:, 1], marker='^', label='Class 1', color='blu
plt.plot(x1_vals, x2_vals, label='Decision Boundary', color='green')
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.title('Data Points with Final Classifier Line (Learning Rate = 1)')
plt.grid(True)
plt.show()
```



d. Plot the training cost (i.e., the learning curve) for all the epochs.

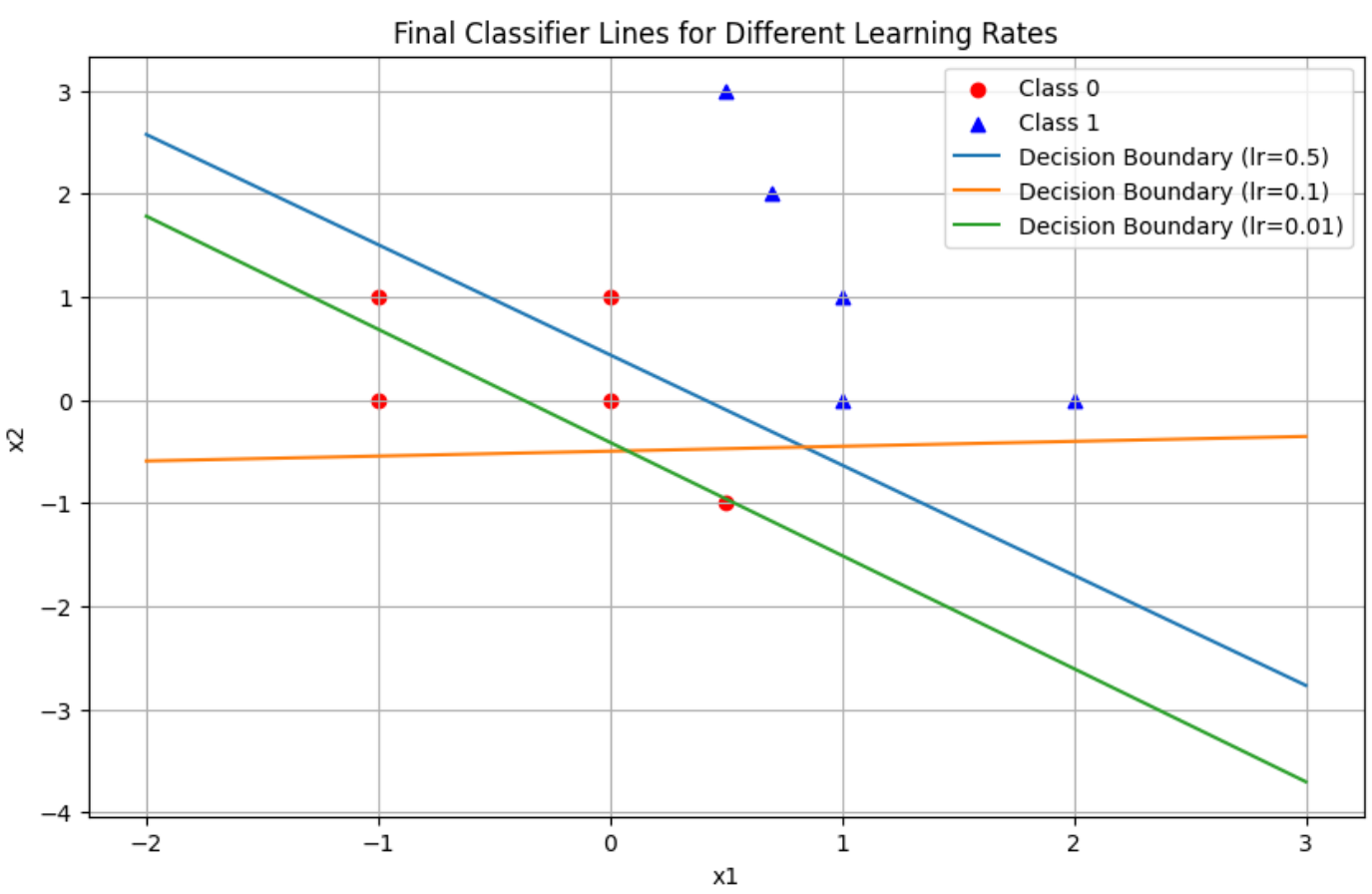
```
learning_rates = [0.5, 0.1, 0.01]
costs = {}

for lr in learning_rates:
    nn = NeuralNetwork(learning_r=lr)
    nn.train(inputs_with_bias, labels, train_num_iterations=50)
    costs[lr] = nn.history['cost']

plt.figure(figsize=(10, 6))
for lr in learning_rates:
    plt.plot(range(50), costs[lr], label=f'Learning Rate = {lr}')
plt.xlabel('Epoch')
plt.ylabel('Training Cost (MSE)')
plt.title('Learning Curves for Different Learning Rates')
plt.legend()
plt.grid(True)
plt.show()
```



e. Repeat step (b.iv) with the learning rates of 0.5, 0.1, and 0.01. Plot the final classifier line and the learning curve for each learning rate.

```
plt.figure(figsize=(10, 6))
plt.scatter(class_0[:, 0], class_0[:, 1], marker='o', label='Class 0', color='red
plt.scatter(class_1[:, 0], class_1[:, 1], marker='^', label='Class 1', color='blu

for lr in learning_rates:
    nn = NeuralNetwork(learning_r=lr)
    nn.train(inputs_with_bias, labels, train_num_iterations=50)
    final_weights = nn.history['weights'][-1]
    w1, w2, b = final_weights[0, 0], final_weights[1, 0], final_weights[2, 0]
    x2_vals = -(w1 / w2) * x1_vals - (b / w2)
```

```
        plt.plot(x1_vals, x2_vals, label=f'Decision Boundary (lr={lr})')

plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.title('Final Classifier Lines for Different Learning Rates')
plt.grid(True)
plt.show()
```



f. What behavior do you observe from the learning curves with the different learning rates? Explain your observations. Which learning rate is more suitable? Explain

Observations from the Learning Curves

Learning Rate = 0.5: The training cost decreases relatively quickly and stabilizes after a few epochs. However, there might be slight oscillations due to the relatively large step size, which can cause the weights to overshoot the optimal values.

Learning Rate = 0.1: The training cost decreases more smoothly and steadily compared to 0.5. It converges to a low value, but the convergence is slower than with 0.5 because the step size is smaller.

Learning Rate = 0.01: The training cost decreases very slowly due to the small step size. Even after 50 epochs, the cost might not have converged to a low value, indicating that the learning rate is too small for efficient training in this case.

Among the three options, the learning rate of **0.01** is likely the most suitable. It strikes a balance between speed and stability. It converges more smoothly than 0.5, avoiding large oscillations. However, it converges slower than 0.01, which is too slow to reach an optimal solution within 50 epochs. A learning rate of 0.5 might converge faster, but it risks overshooting and oscillating around the minimum, potentially leading to a less stable solution. A learning rate of 0.01 is too small, resulting in extremely slow convergence, which is inefficient for this problem.