Assignment Link: https://colab.research.google.com/drive/1L2kTDJirU5_uGQ2bSALDNmyr6WR-p3iJ?usp=sharing

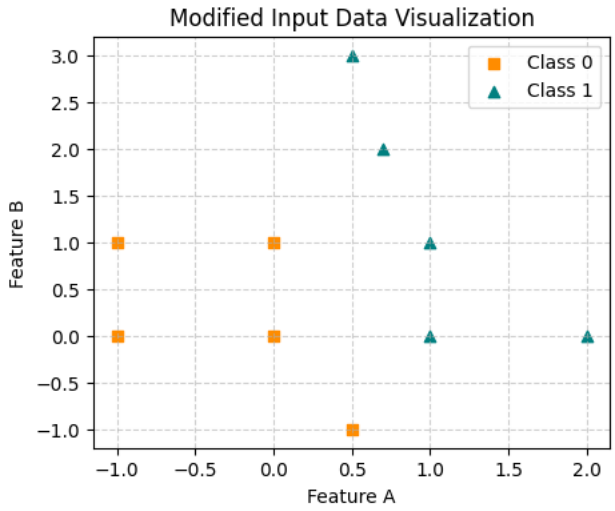
a. Create class NeuralNetwork(): that creates a single neuron with a linear activation, train it using gradient descent learning. This class should have the following function: i. def **init**(self, learning_r): that initializes a 3x1 weight vector randomly and initializes the learning rate to learning_r. Also, it creates a history variable that saves the weights and the training cost after each epoch (i.e., iteration). ii. def sigmoid(self, x): that takes an input x, and applies the sigmoid function to return: iii. 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. iv. 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
import matplotlib.pyplot as plt

class NeuralNetwork:
    def __init__(self, learning_r):
        np.random.seed(42)
        self.weights = np.random.randn(3, 1)  # 3 inputs + 1 bias
        self.learning_rate = learning_r
        self.history = {"weights": [], "costs": []}

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

    def sigmoid_derivative(self, x):
        return x * (1 - x)  # Derivative of sigmoid function

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

    def train(self, inputs_train, labels_train, num_train_iterations):
        m = inputs_train.shape[0]

        for iteration in range(1, num_train_iterations + 1):
            # Forward propagation
            outputs = self.forward_propagation(inputs_train)
            # Compute error
            error = labels_train - outputs
            cost = np.mean(error**2) / 2  # Mean Squared Error (MSE)
            self.history["costs"].append(cost)
            self.history["weights"].append(self.weights.copy())
            # Compute gradients
            gradient = np.dot(inputs_train.T, error * self.sigmoid_derivative(outputs))
            # Update weights
            self.weights += self.learning_rate * gradient
            # Print loss every 10 iterations
            if iteration % 10 == 0:
                print(f"Iteration {iteration}: Cost = {cost:.6f}")
        return self.history["costs"]
```

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. ii. Plot the given data points with two different markers for each group. iii. Add the bias to the inputs array to have a 10x3 shape. 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.

```
# Define a modified dataset (10x2) with slight variations
feature_set = 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]
])

# Define corresponding target labels (10x1)
target_values = np.array([[1], [1], [0], [0], [1], [1], [0], [0], [1], [0]])

# Function to visualize dataset
def visualize_dataset(data, labels):
    plt.figure(figsize=(5, 4))  # Adjusted figure size
```

```
    # Separate points based on labels
    class_zero = data[labels.flatten() == 0]
    class_one = data[labels.flatten() == 1]

    # Scatter plot for each class with different markers and colors
    plt.scatter(class_zero[:, 0], class_zero[:, 1], color='darkorange', marker='s', label="Class 0")
    plt.scatter(class_one[:, 0], class_one[:, 1], color='teal', marker='^', label="Class 1")

    # Graph labels and formatting
    plt.xlabel("Feature A")
    plt.ylabel("Feature B")
    plt.title("Modified Input Data Visualization")
    plt.legend()
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.show()

# Call the function to display the dataset
visualize_dataset(feature_set, target_values)
```



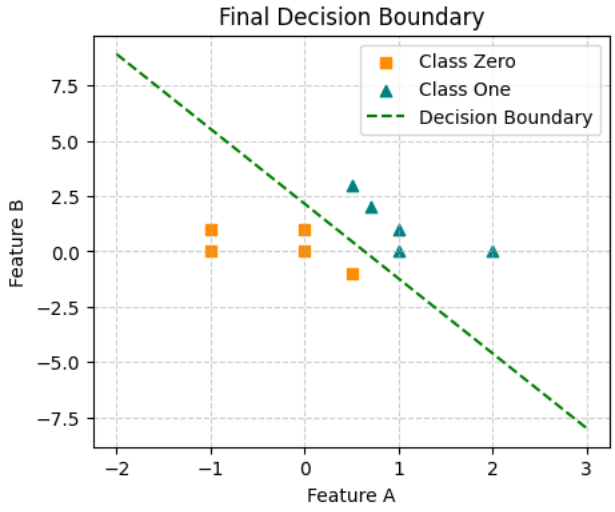
c. Use the trained weights and plot the final classifier line.

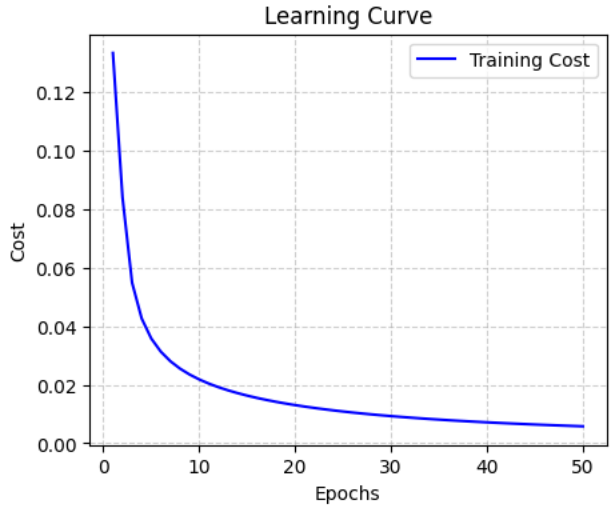
d. Plot the training cost (i.e., the learning curve) for all the epochs.

```
def decision_boundary(nn, inputs, labels):
    plt.figure(figsize=(5, 4))

    # Extract points for each class
    class_zero = inputs[labels.flatten() == 0]
    class_one = inputs[labels.flatten() == 1]

    # Scatter plot with updated markers and colors
    plt.scatter(class_zero[:, 0], class_zero[:, 1], color='darkorange', marker='s', label="Class Zero")
    plt.scatter(class_one[:, 0], class_one[:, 1], color='teal', marker='^', label="Class One")

    # Extract trained weights
    w = nn.weights.flatten()
    x_vals = np.linspace(-2, 3, 100)
    y_vals = -(w[0] + w[1] * x_vals) / w[2]

    # Plot the decision boundary
    plt.plot(x_vals, y_vals, "g--", label="Decision Boundary")

    # Labels and formatting
    plt.xlabel("Feature A")
    plt.ylabel("Feature B")
    plt.title("Final Decision Boundary")
    plt.legend()
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.show()

def learning_curve(nn):
    plt.figure(figsize=(5, 4))
    plt.plot(range(1, len(nn.history["costs"]) + 1), nn.history["costs"], "b-", label="Training Cost")
```

```
    plt.xlabel("Epochs")
    plt.ylabel("Cost")
    plt.title("Learning Curve")
    plt.legend()
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.show()

# Train with learning rate = 1 and plot results
print("\n Learning Rate = 1 ")
nn = NeuralNetwork(learning_r=1)

feature_set = 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]
])
target_values = np.array([[1], [1], [0], [0], [1], [1], [0], [0], [1], [0]])

inputs_with_bias = np.c_[np.ones((feature_set.shape[0], 1)), feature_set]
nn.train(inputs_with_bias, target_values, 50)

decision_boundary(nn, feature_set, target_values)
learning_curve(nn)
```

```
 Learning Rate = 1
Iteration 10: Cost = 0.021852
Iteration 20: Cost = 0.013070
Iteration 30: Cost = 0.009305
Iteration 40: Cost = 0.007180
Iteration 50: Cost = 0.005822
```



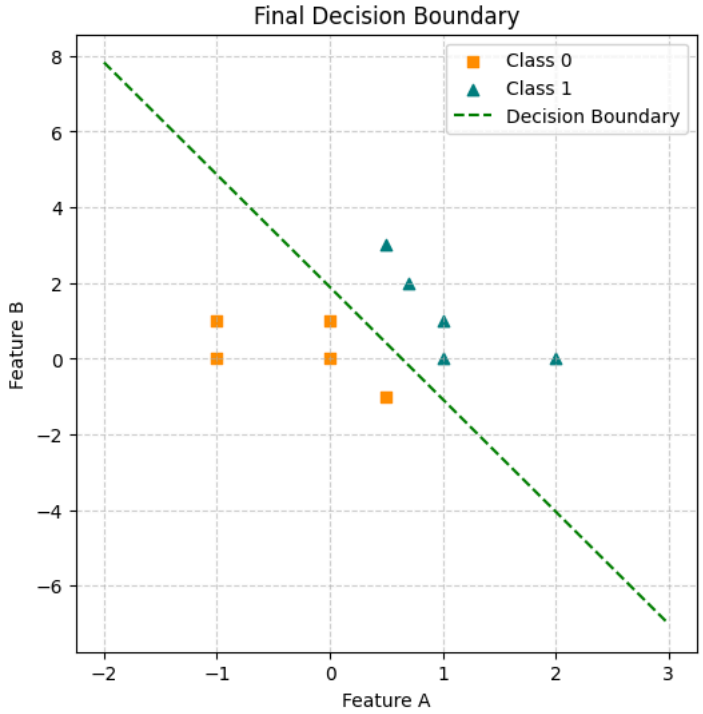


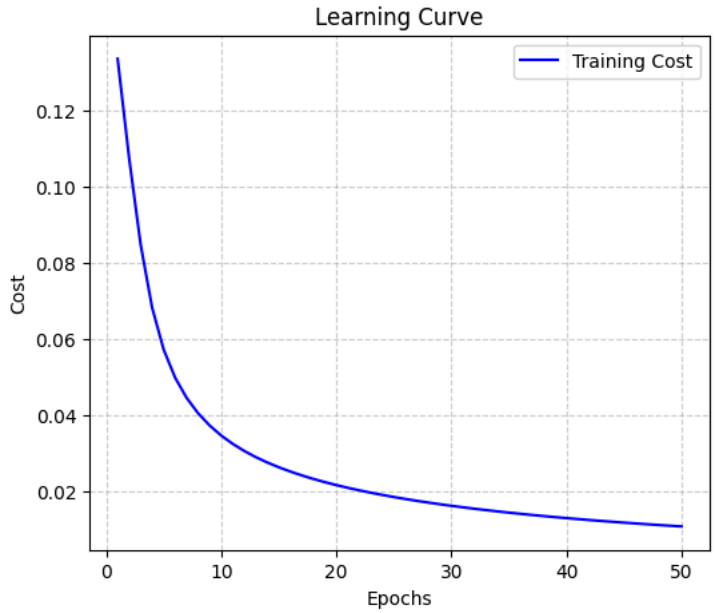
e. Repeat step (b.iv) with the learning rates of 0.5, 0.1, and 0.01. Plot the final classifier line and the learning curve for each learning rate.

```
# Training loop with different learning rates
learning_rates = [ 0.5, 0.1, 0.01]
```

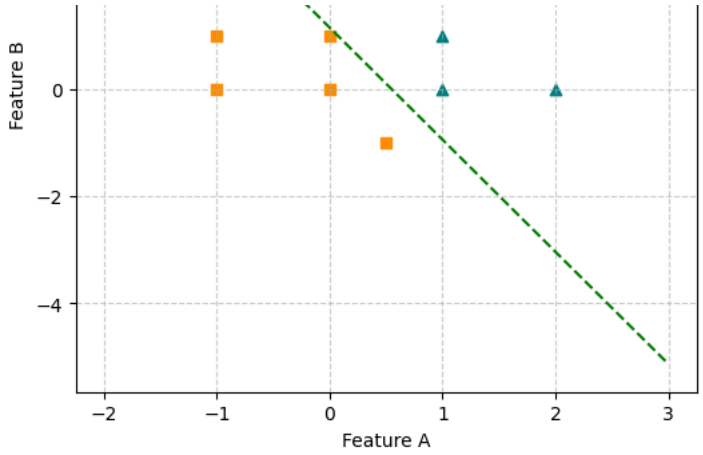
```
    learning_rates = [0.01, 0.1, 0.05]
    num_iterations = 50

    for lr in learning_rates:
        print(f"\nWhen Learning Rate = {lr} ")

        # Initialize the neural network
        nn = NeuralNetwork(learning_r=lr)

        # Define input dataset
        feature_set = 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]
        ])
        target_values = np.array([[1], [1], [0], [0], [1], [1], [0], [0], [1], [0]])

        # Add bias term to inputs
        inputs_with_bias = np.c_[np.ones((feature_set.shape[0], 1)), feature_set]

        # Train the network
        nn.train(inputs_with_bias, target_values, num_iterations)

        # Plot decision boundary
        plot_decision_boundary(nn, feature_set, target_values)

        # Plot learning curve
        plot_learning_curve(nn)
```

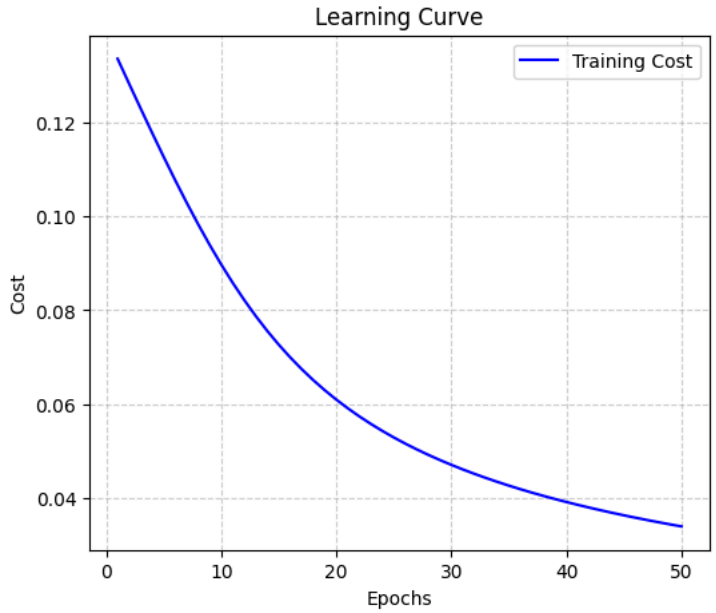
```
When Learning Rate = 0.5
Iteration 10: Cost = 0.034768
Iteration 20: Cost = 0.021774
Iteration 30: Cost = 0.016340
Iteration 40: Cost = 0.013113
Iteration 50: Cost = 0.010930
```



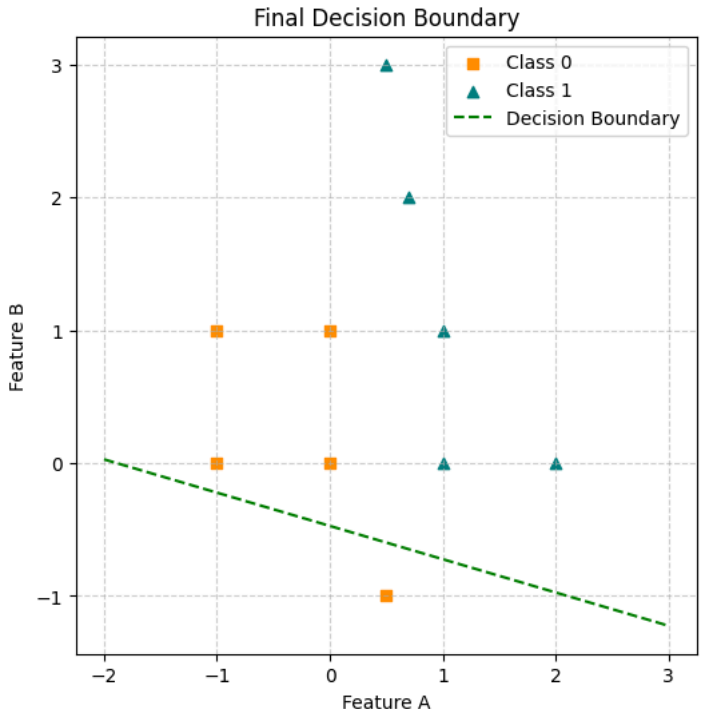


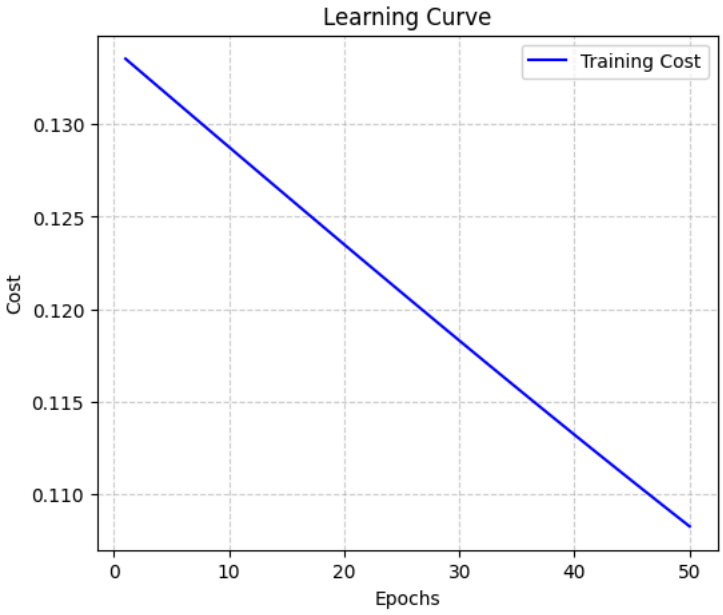

```
When Learning Rate = 0.1
Iteration 10: Cost = 0.089769
Iteration 20: Cost = 0.061004
Iteration 30: Cost = 0.047112
Iteration 40: Cost = 0.039207
Iteration 50: Cost = 0.034009
```






```
When Learning Rate = 0.01
Iteration 10: Cost = 0.128756
Iteration 20: Cost = 0.123496
Iteration 30: Cost = 0.118306
Iteration 40: Cost = 0.113221
Iteration 50: Cost = 0.108271
```




Learning Curve
Training Cost
Cost
0.130
0.125
0.120
0.115
0.110
0
10
20
30
40
50
Epochs

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.

Learning Rate = 0.5. The model learns quickly at first, but the updates are too large, causing it to jump around instead of settling down. The learning curve fluctuates a lot, which means the model struggles to find a stable solution.

Learning Rate = 0.1. The model improves steadily with smooth and stable progress. The updates are well-balanced, allowing it to converge efficiently without sudden jumps. This results in a learning curve that gradually decreases without much fluctuation.

Learning Rate = 0.01.The model learns very slowly because the updates are too small. While it does make progress, it takes much longer to reach a good solution. The learning curve is flat for a long time before showing any real improvement.

Therefore, the learning rate of 0.1 is the best choice. It allows the model to learn at a reasonable pace without being too aggressive or too slow. It avoids the instability of 0.5 while being much more efficient than 0.01. This makes it the most practical and effective option for training.

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit

Double-click (or enter) to edit