National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Electrical Engineering

Faculty Member: Ma'am Neelma Naz

Date: October 23, 2025

Semester: 7th

Group: 02

CS471 Machine Learning

Lab 7: Logistic Regression

		PLO4	**PLO5**	**PLO5**	**PLO8**	**PLO9**
		CLO4	CLO5	CLO5	CLO6	CLO7
Student Name	**Reg. No**	**Viva / Quiz / Demo** 5 Marks	**Analysis of Data in Report** 5 Marks	**Modern Tool Usage** 5 Marks	**Ethics** 5 Marks	**Individual and Teamwork** 5 Marks
Hanzla Sajjad	403214					
Irfa Farooq	412564					

Introduction

This laboratory exercise will focus on the python implementation of logistic regression. Logistic regression is a supervised learning technique that incorporates a sigmoid function activation with the linear regression algorithm to implement classification. Unlike regression, classification involves discrete labels such 0/1, true/false, cat/not a cat, benign/malignant etc. The sigmoid function causes the hypothesis values to take place between 0 or 1. Similar to regression, weight parameters are trained on a dataset so as to fit a model that can make accurate predictions from that dataset. To prevent overfitting, regularization can be used in logistic regression.

Objectives

The following are the main objectives of this lab:

- Extract and prepare the training and validation datasets
- Use feature scaling to ensure uniformity among the feature columns
- Implement cost function to get the overall loss
- Implement gradient descent algorithm to update weight parameters
- Plot the training and validation losses
- Make scatter plots of the classification

Lab Conduct

- Respect faculty and peers through speech and actions
- The lab faculty will be available to assist the students. In case some aspect of the lab experiment is not understood, the students are advised to seek help from the faculty.
- In the tasks, there are commented lines such as #YOUR CODE STARTS HERE# where you have to provide the code. You must put the code/screenshot/plot between the #START and #END parts of these commented lines. Do NOT remove the commented lines.

- Use the tab key to provide the indentation in python.
- When you provide the code in the report, keep the font size at 12

Theory

Logistic Regression is another basic supervised learning technique besides Linear Regression. In logistic regression, the linear regression algorithm is modified by applying a sigmoid function to the predicted value. This causes the prediction to fall between 0 to 1 values. Thus, logistic regression is actually a classification technique built from the linear regression. The sigmoid function is a type of an activation function. Aside from loss and accuracy, logistic regression also at times, requires calculation of precision and recall. This is needed when the dataset is skewed; the two class labels are not equally distributed in the dataset.

A brief summary of the relevant keywords and functions in python is provided below:

print()	output text on console
input()	get input from user on console
range()	create a sequence of numbers
len()	gives the number of characters in a string
if	contains code that executes depending on a logical condition
else	connects with **if** and **elif**, executes when conditions are not met
elif	equivalent to **else if**
while	loops code as long as a condition is true
for	loops code through a sequence of items in an iterable object
break	exit loop immediately
continue	jump to the next iteration of the loop
def	used to define a function

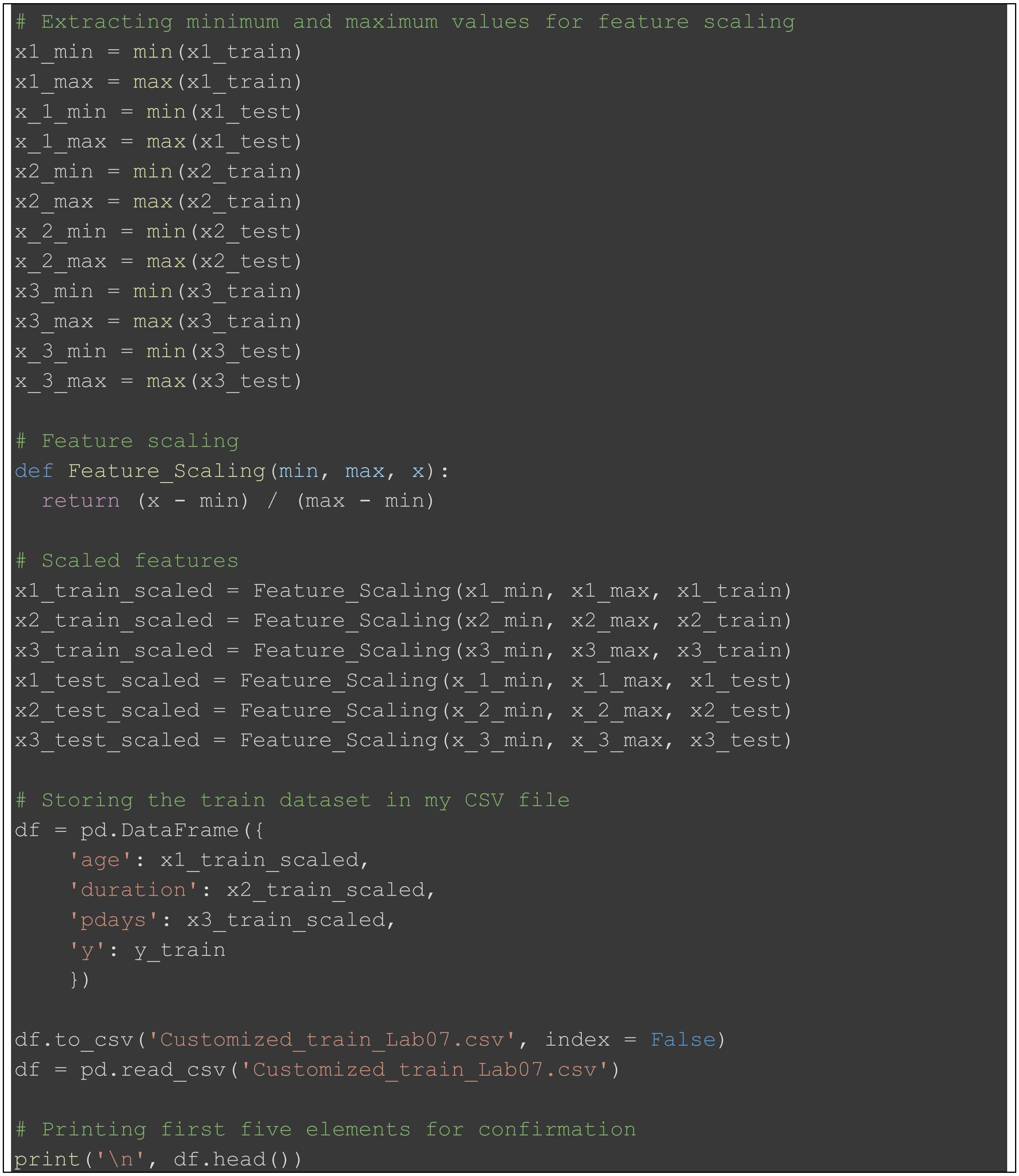
Lab Task 1 - Dataset Preparation, Feature Scaling

You have been provided with a dataset containing several feature columns. You will need to select any 3 of the feature columns to make your own dataset. Also, you will need to choose the label column that your model will predict. Ensure that the label column has binary values. Specify the features and label that you choose.

The dataset examples are to be divided into 2 separate portions: training and validation datasets (choose from 80-20 to 70-30 ratios). Load the dataset into your python program, split them into the portions and store them as NumPy arrays (X_{train} , y_{train}, X_{val}, $y_{val,}$). Next, use feature scaling to rescale the feature columns for both datasets so that their values range from 0 to 1. Finally, print both of the datasets (you need to show any 5 rows of the datasets).

Code

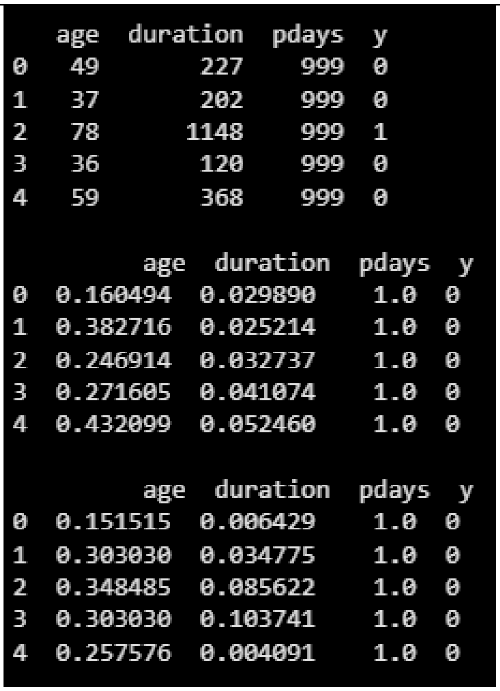
```
# Task 1
df = pd.read_csv('ML_Lab07_new_train.csv')

# Customizing the data set
df = df[['age', 'duration', 'pdays', 'y']]

# Printing first five elements for confirmation
print(df.head())

# Extracting features in separate variables
x1 = np.array(df['age'])
x2 = np.array(df['duration'])
x3 = np.array(df['pdays'])
y = np.array(df['y'])

# Performing train test split on data
x1_train, x1_test, x2_train, x2_test, x3_train, x3_test, y_train, y_test =
train_test_split(x1, x2, x3, y, test_size = 0.2, random_state = 42)
```



```
# Extracting minimum and maximum values for feature scaling
x1_min = min(x1_train)
x1_max = max(x1_train)
x_1_min = min(x1_test)
x_1_max = max(x1_test)
x2_min = min(x2_train)
x2_max = max(x2_train)
x_2_min = min(x2_test)
x_2_max = max(x2_test)
x3_min = min(x3_train)
x3_max = max(x3_train)
x_3_min = min(x3_test)
x_3_max = max(x3_test)

# Feature scaling
def Feature_Scaling(min, max, x):
  return (x - min) / (max - min)

# Scaled features
x1_train_scaled = Feature_Scaling(x1_min, x1_max, x1_train)
x2_train_scaled = Feature_Scaling(x2_min, x2_max, x2_train)
x3_train_scaled = Feature_Scaling(x3_min, x3_max, x3_train)
x1_test_scaled = Feature_Scaling(x_1_min, x_1_max, x1_test)
x2_test_scaled = Feature_Scaling(x_2_min, x_2_max, x2_test)
x3_test_scaled = Feature_Scaling(x_3_min, x_3_max, x3_test)

# Storing the train dataset in my CSV file
df = pd.DataFrame({
    'age': x1_train_scaled,
    'duration': x2_train_scaled,
    'pdays': x3_train_scaled,
    'y': y_train
    })

df.to_csv('Customized_train_Lab07.csv', index = False)
df = pd.read_csv('Customized_train_Lab07.csv')

# Printing first five elements for confirmation
print('\n', df.head())
```

```
# Storing the test dataset in my CSV file
df = pd.DataFrame({
    'age': x1_test_scaled,
    'duration': x2_test_scaled,
    'pdays': x3_test_scaled,
    'y': y_test
    })

df.to_csv('Customized_test_Lab07.csv', index = False)
df = pd.read_csv('Customized_test_Lab07.csv')

# Printing first five elements for confirmation
print('\n', df.head())
```

Output Console



```
   age  duration  pdays  y
0   49       227    999  0
1   37       202    999  0
2   78      1148    999  1
3   36       120    999  0
4   59       368    999  0

        age  duration  pdays  y
0  0.160494  0.029890    1.0  0
1  0.382716  0.025214    1.0  0
2  0.246914  0.032737    1.0  0
3  0.271605  0.041074    1.0  0
4  0.432099  0.052460    1.0  0

        age  duration  pdays  y
0  0.151515  0.006429    1.0  0
1  0.303030  0.034775    1.0  0
2  0.348485  0.085622    1.0  0
3  0.303030  0.103741    1.0  0
4  0.257576  0.004091    1.0  0
```

Lab Task 2 – Sigmoid Activation

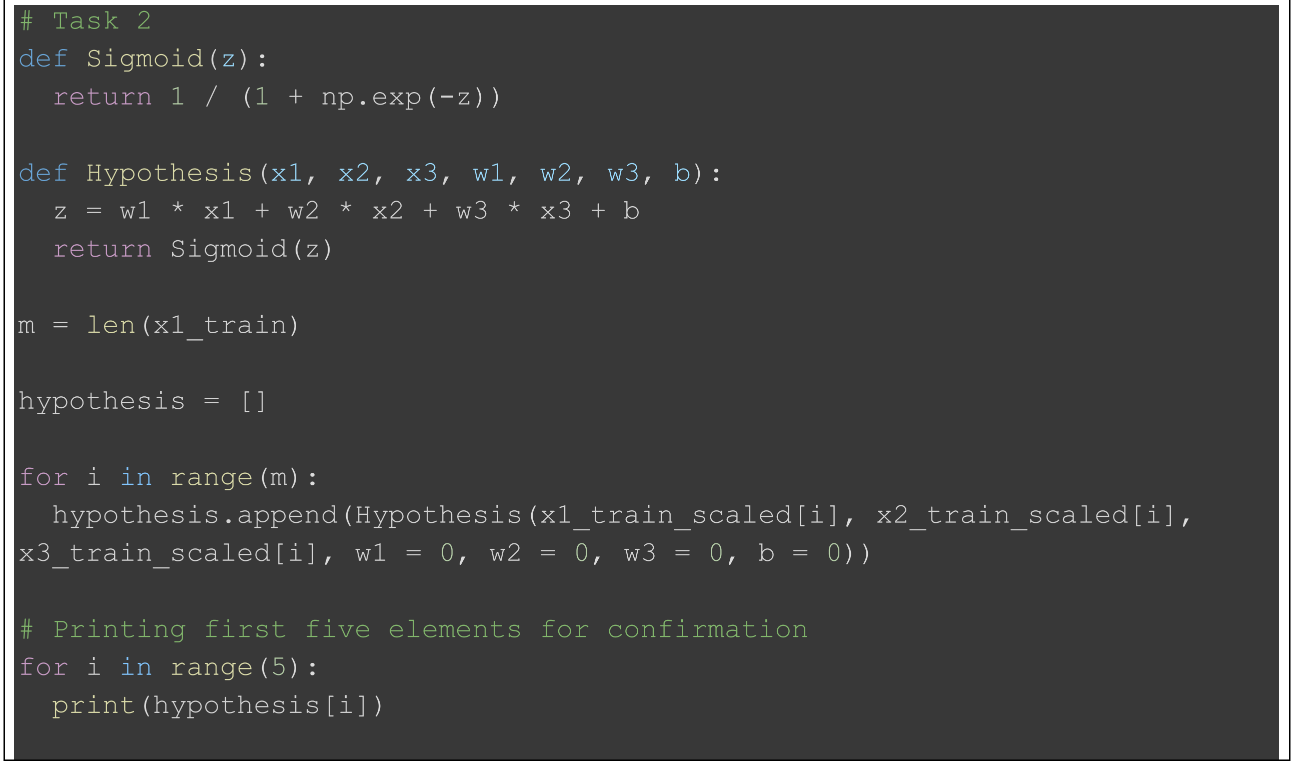
For logistic regression, you will implement the following hypothesis:

$$h(x) = g(b + w_1x_1 + w_2x_2 + w_3x_3 + \ldots)$$

$$g(z) = 1 / (1 + e^{-z})$$

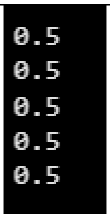
The w represents the weights and the x represent the features. h(x) is to be calculated for each training example and its difference with the label y of that training example will represent the loss. The g(z) function represents the sigmoid activation function. In this task, you will write a function that takes in a value z as argument and outputs the result of the sigmoid activation g(z). Provide the code for this task.

Code



```
# Task 2
def Sigmoid(z):
  return 1 / (1 + np.exp(-z))

def Hypothesis(x1, x2, x3, w1, w2, w3, b):
  z = w1 * x1 + w2 * x2 + w3 * x3 + b
  return Sigmoid(z)

m = len(x1_train)

hypothesis = []

for i in range(m):
  hypothesis.append(Hypothesis(x1_train_scaled[i], x2_train_scaled[i], 
x3_train_scaled[i], w1 = 0, w2 = 0, w3 = 0, b = 0))

# Printing first five elements for confirmation
for i in range(5):
  print(hypothesis[i])
```

Output Console



```
0.5
0.5
0.5
0.5
0.5
```

Lab Task 3 - Cost Function

In this task, you will write a cost function that calculates the overall loss across a set of training examples. This cost function will be useful to calculate the losses in both the training and validation phases of the program.

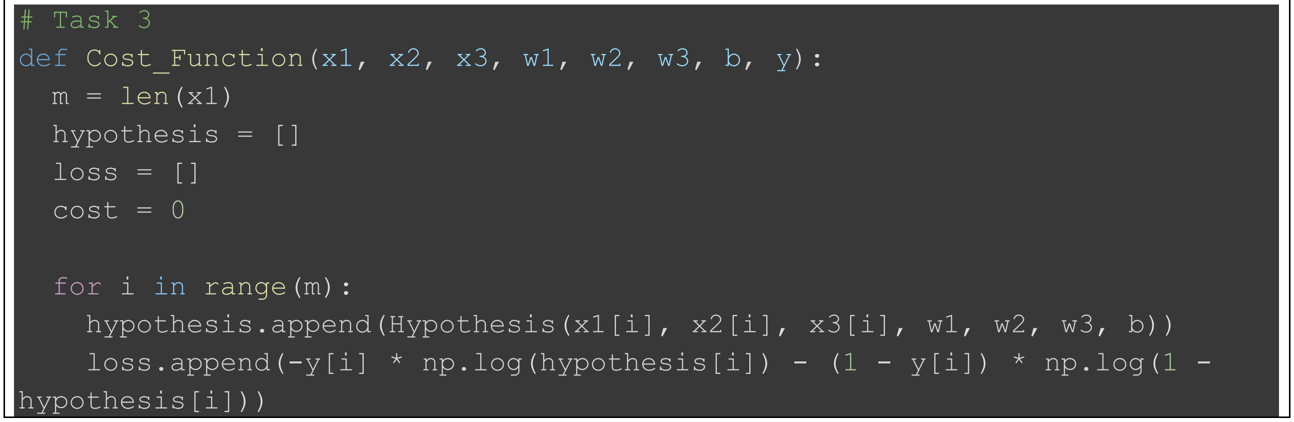
```
cost_function(X, y)
```

The X and y are the features and labels of the training/validation dataset. The function will return the cost value. The cost function is given by:

$$J(w) = \frac{1}{m}\sum_{i=0}^{m-1}\left(-y^{(i)}\log\left(h\left(x^{(i)}\right)\right) - \ \left(1-y^{(i)}\right)\log\left(1-\ h\left(x^{(i)}\right)\right)\right)$$

The m is the number of the training/validation examples in the dataset. Remember that the hypothesis requires sigmoid activation. Write the code for the cost function and implement it to print out the cost. You will need to initialize the weights to some random values in order to calculate the hypothesis. Provide the code and all relevant screenshots showcasing the use of your cost function.

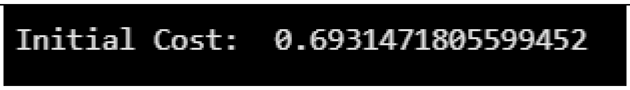
Code



```
# Task 3
def Cost_Function(x1, x2, x3, w1, w2, w3, b, y):
  m = len(x1)
  hypothesis = []
  loss = []
  cost = 0

  for i in range(m):
    hypothesis.append(Hypothesis(x1[i], x2[i], x3[i], w1, w2, w3, b))
    loss.append(-y[i] * np.log(hypothesis[i]) - (1 - y[i]) * np.log(1 -
hypothesis[i]))
```

```
  cost = np.sum(loss) / m
  return cost

# Printing initial cost
print('Initial Cost: ', Cost_Function(x1_train_scaled, x2_train_scaled,
x3_train_scaled, 0, 0, 0, 0, y_train))
```

Output Console



```
Initial Cost:  0.6931471805599452
```

Lab Task 4 – Gradient Descent

In this task, you will write a function that uses gradient descent to update the weight parameters:

```
gradient_descent(X, y, alpha)
```

The X and y are the features and labels of the training dataset, *alpha* is the learning rate which is a tuning hyperparameter. The gradient descent algorithm is given as follows:

$$dw_j = \frac{\partial \mathrm{J}}{\partial w_j} = \frac{1}{m}\sum_{i=1}^{m}(\mathrm{h}(\mathrm{x}^{(i)}) - \mathrm{y}^{(i)})x_j{}^{(i)}$$

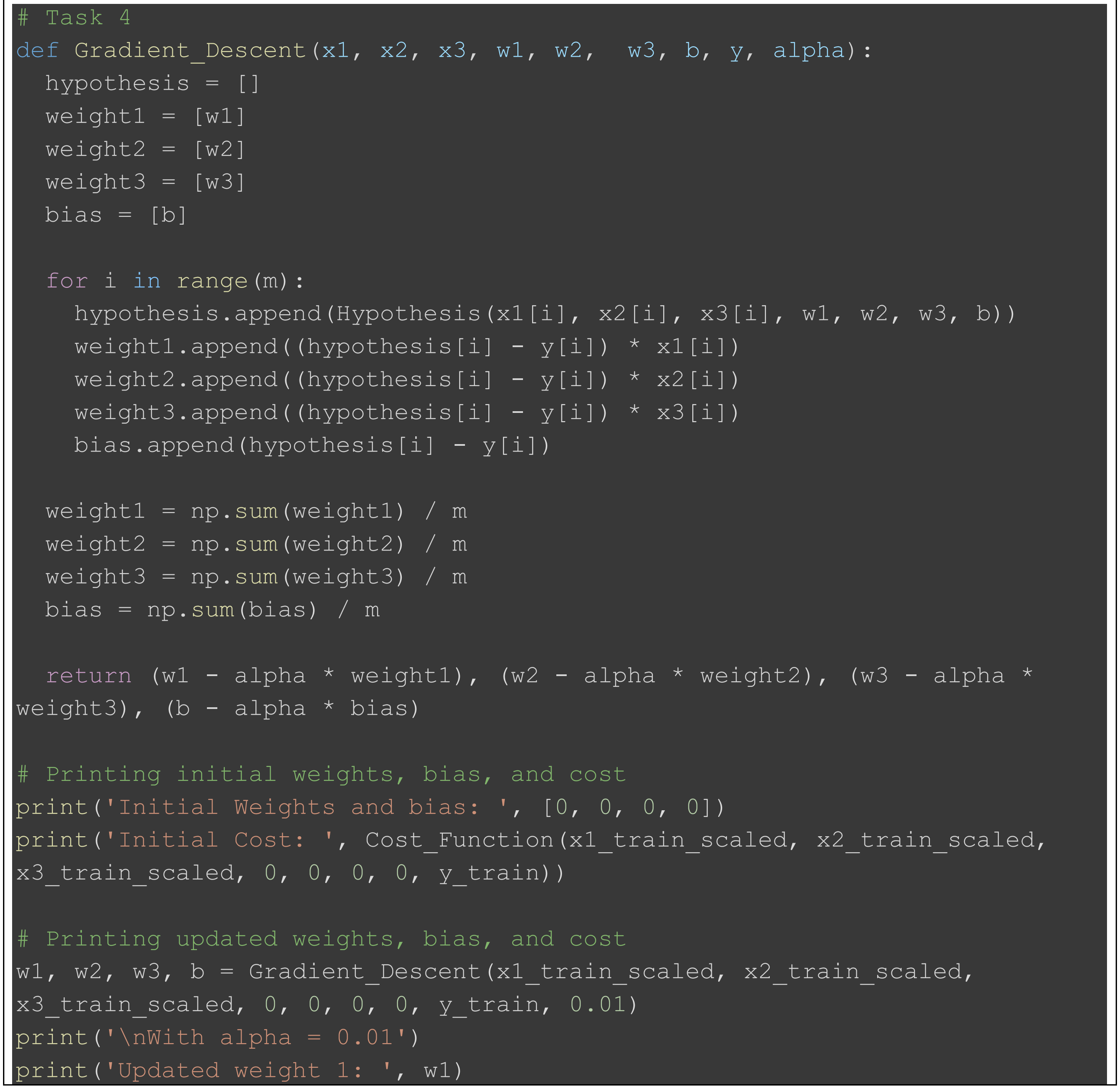
$$db = \frac{\partial \mathrm{J}}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(\mathrm{h}(\mathrm{x}^{(i)}) - \mathrm{y}^{(i)})$$

$$w_j := w_j - \alpha\frac{\partial \mathrm{J}}{\partial w_j}$$

$$b := b - \alpha\frac{\partial \mathrm{J}}{\partial w_j}$$

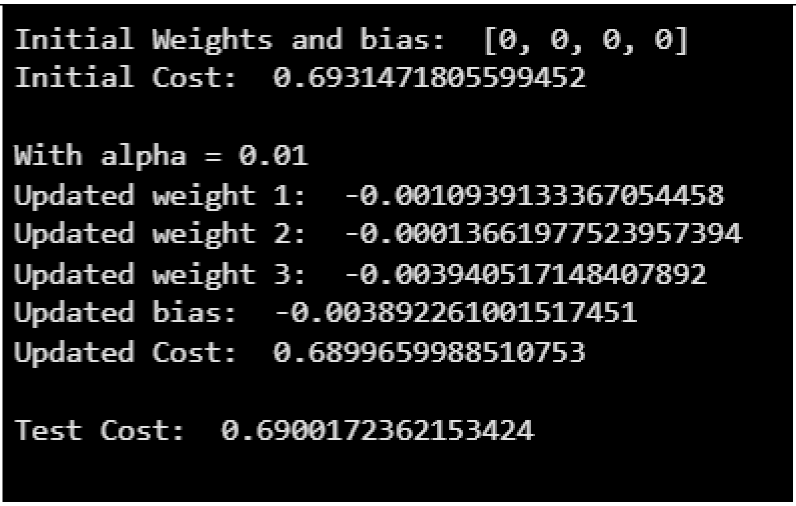
The gradient descent for logistic regression may seem identical to that in linear regression, however, it should be noted that they are not the same formulas as the cost function for the logistic regression is different from that of linear regression.

For the submission, you will need to run the gradient descent algorithm once to update the weights. You will need to print the weights, training cost and validation cost both before and after the weight update. Provide the code and all relevant screenshots of the final output.

Code


```
# Task 4
def Gradient_Descent(x1, x2, x3, w1, w2,  w3, b, y, alpha):
  hypothesis = []
  weight1 = [w1]
  weight2 = [w2]
  weight3 = [w3]
  bias = [b]

  for i in range(m):
    hypothesis.append(Hypothesis(x1[i], x2[i], x3[i], w1, w2, w3, b))
    weight1.append((hypothesis[i] - y[i]) * x1[i])
    weight2.append((hypothesis[i] - y[i]) * x2[i])
    weight3.append((hypothesis[i] - y[i]) * x3[i])
    bias.append(hypothesis[i] - y[i])

  weight1 = np.sum(weight1) / m
  weight2 = np.sum(weight2) / m
  weight3 = np.sum(weight3) / m
  bias = np.sum(bias) / m

  return (w1 - alpha * weight1), (w2 - alpha * weight2), (w3 - alpha *
weight3), (b - alpha * bias)

# Printing initial weights, bias, and cost
print('Initial Weights and bias: ', [0, 0, 0, 0])
print('Initial Cost: ', Cost_Function(x1_train_scaled, x2_train_scaled,
x3_train_scaled, 0, 0, 0, 0, y_train))

# Printing updated weights, bias, and cost
w1, w2, w3, b = Gradient_Descent(x1_train_scaled, x2_train_scaled,
x3_train_scaled, 0, 0, 0, 0, y_train, 0.01)
print('\nWith alpha = 0.01')
print('Updated weight 1: ', w1)
print('Updated weight 2: ', w2)
print('Updated weight 3: ', w3)
print('Updated bias: ', b)
print('Updated Cost: ', Cost_Function(x1_train_scaled, x2_train_scaled,
x3_train_scaled, w1, w2, w3, b, y_train))

# Printing test cost with updated weights
print('\nTest Cost: ', Cost_Function(x1_test_scaled, x2_test_scaled,
x3_test_scaled, w1, w2, w3, b, y_test))
```

Output Console



```
Initial Weights and bias:  [0, 0, 0, 0]
Initial Cost:  0.6931471805599452

With alpha = 0.01
Updated weight 1:  -0.0010939133367054458
Updated weight 2:  -0.00013661977523957394
Updated weight 3:  -0.003940517148407892
Updated bias:  -0.003892261001517451
Updated Cost:  0.6899659988510753

Test Cost:  0.6900172362153424
```

Lab Task 5 – Training and Validation Implementation

In this task, you will use the functions from the previous two tasks to write a "main" function that performs the actual training and validation. Use the cost function and gradient descent function on the training examples to determine the training loss and update the weights. Then, use the cost function on the validation examples to determine the validation loss. This single iteration over the entire dataset (both training and validation) marks completion of one epoch. You will need to perform the training and validation over several epochs (the epoch number is another hyperparameter that must be chosen). Ensure that at the end of each epoch, the training loss, validation loss, precision value and recall value are stored for plotting purposes. To get the precision and recall, use the following:

$$Precision = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

$$Recall = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}$$

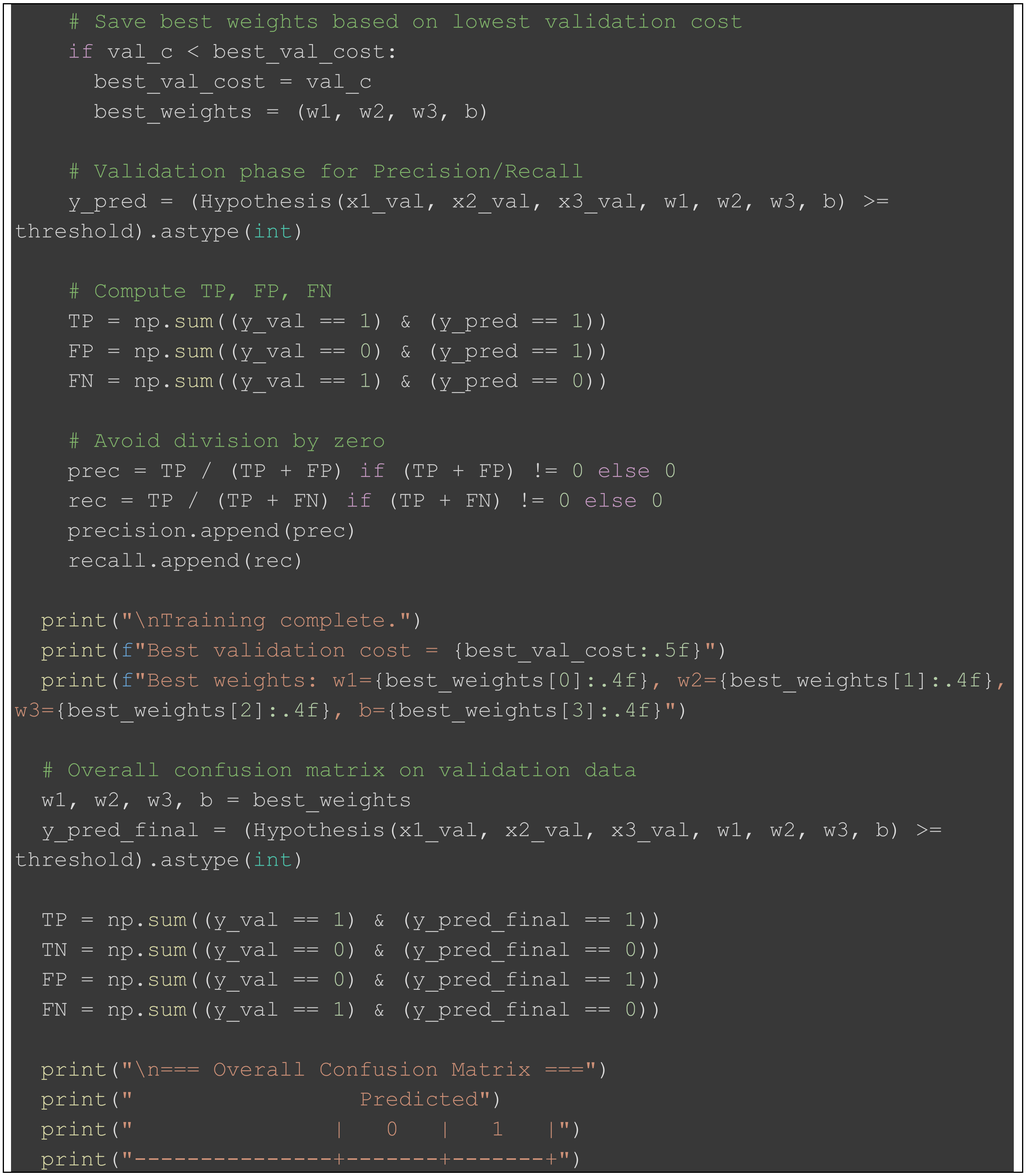
Start the training at some value of alpha. Try multiple training attempts with various alpha values and find the best value of the step size. Once you have found the value for the step size, showcase the output by making three plots:

1. training loss and validation loss vs. the epoch number
2. precision and recall vs. the epoch number
3. precision vs. recall

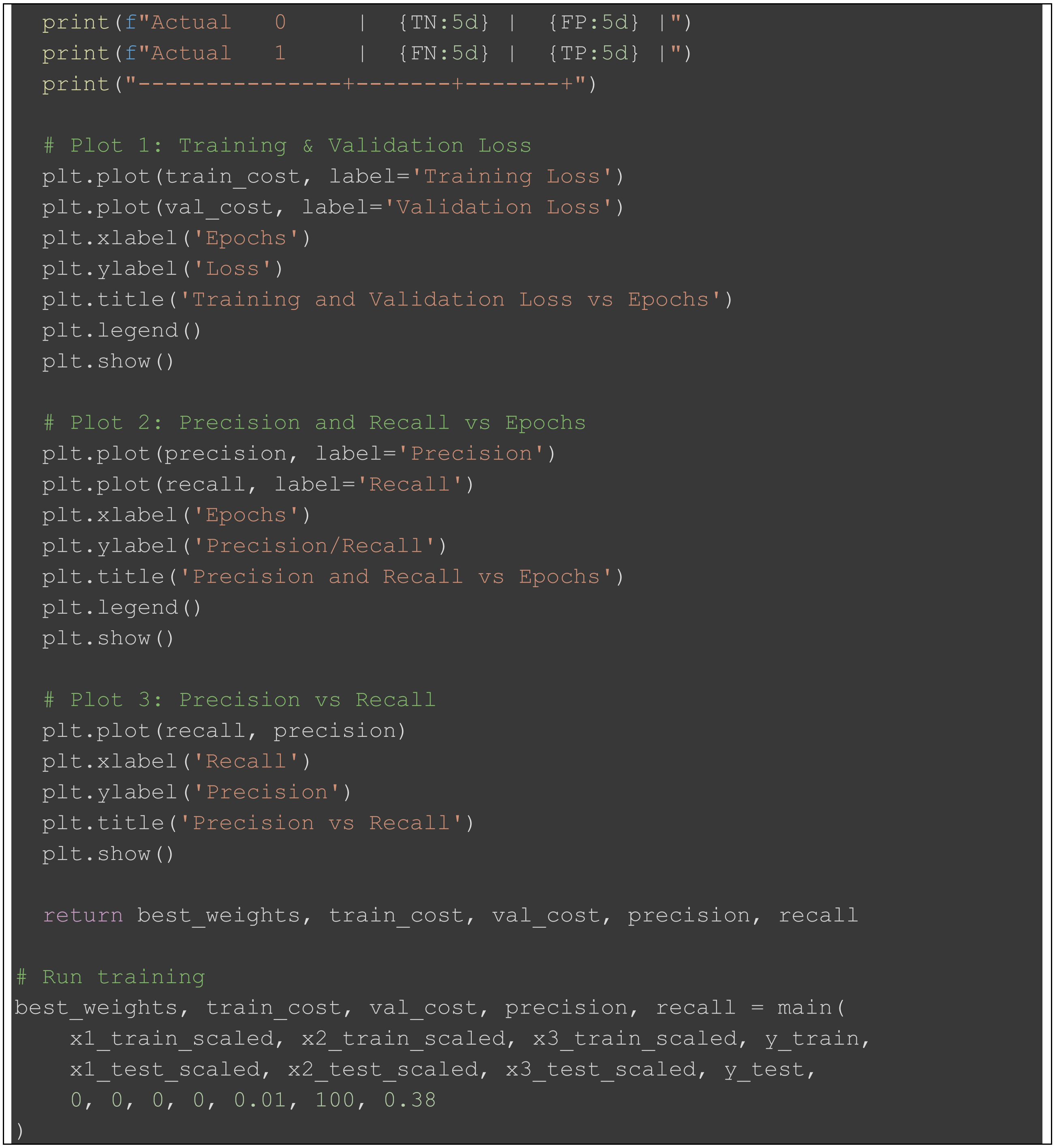
Ensure all axes are labeled appropriately. Provide the code (excluding function definitions) and all plots of the final output.

Code

```
# Task 5
def main(x1_train, x2_train, x3_train, y_train,
         x1_val, x2_val, x3_val, y_val,
         w1, w2, w3, b, alpha, epochs, threshold):

  # Initialize trackers
  train_cost = []
  val_cost = []
  precision = []
  recall = []

  weight1 = [w1]
  weight2 = [w2]
  weight3 = [w3]
  bias = [b]

  best_val_cost = float('inf')
  best_weights = (w1, w2, w3, b)

  for i in range(epochs):
    # Training Phase
    w1, w2, w3, b = Gradient_Descent(x1_train, x2_train, x3_train, w1, w2,
w3, b, y_train, alpha)

    # Store weights
    weight1.append(w1)
    weight2.append(w2)
    weight3.append(w3)
    bias.append(b)

    # Compute cost for training and validation
    tr_cost = Cost_Function(x1_train, x2_train, x3_train, w1, w2, w3, b,
y_train)
    val_c = Cost_Function(x1_val, x2_val, x3_val, w1, w2, w3, b, y_val)
    train_cost.append(tr_cost)
    val_cost.append(val_c)
```

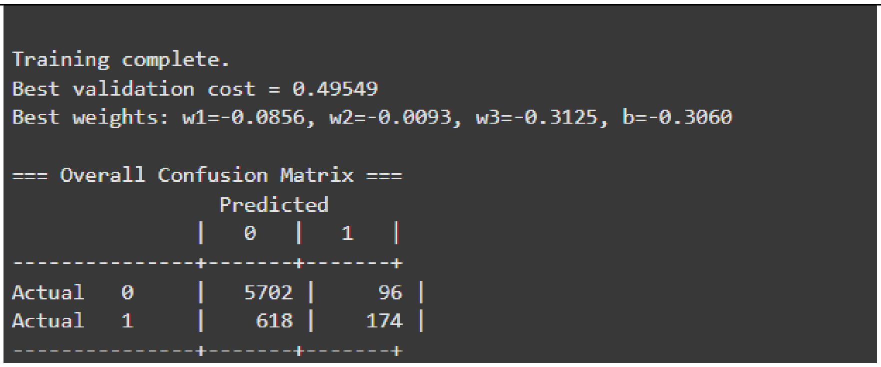


```
    # Save best weights based on lowest validation cost
    if val_c < best_val_cost:
      best_val_cost = val_c
      best_weights = (w1, w2, w3, b)

    # Validation phase for Precision/Recall
    y_pred = (Hypothesis(x1_val, x2_val, x3_val, w1, w2, w3, b) >=
threshold).astype(int)

    # Compute TP, FP, FN
    TP = np.sum((y_val == 1) & (y_pred == 1))
    FP = np.sum((y_val == 0) & (y_pred == 1))
    FN = np.sum((y_val == 1) & (y_pred == 0))

    # Avoid division by zero
    prec = TP / (TP + FP) if (TP + FP) != 0 else 0
    rec = TP / (TP + FN) if (TP + FN) != 0 else 0
    precision.append(prec)
    recall.append(rec)

  print("\nTraining complete.")
  print(f"Best validation cost = {best_val_cost:.5f}")
  print(f"Best weights: w1={best_weights[0]:.4f}, w2={best_weights[1]:.4f},
w3={best_weights[2]:.4f}, b={best_weights[3]:.4f}")

  # Overall confusion matrix on validation data
  w1, w2, w3, b = best_weights
  y_pred_final = (Hypothesis(x1_val, x2_val, x3_val, w1, w2, w3, b) >=
threshold).astype(int)

  TP = np.sum((y_val == 1) & (y_pred_final == 1))
  TN = np.sum((y_val == 0) & (y_pred_final == 0))
  FP = np.sum((y_val == 0) & (y_pred_final == 1))
  FN = np.sum((y_val == 1) & (y_pred_final == 0))

  print("\n=== Overall Confusion Matrix ===")
  print("                 Predicted")
  print("               |   0   |   1   |")
  print("---------------+-------+-------+")
```

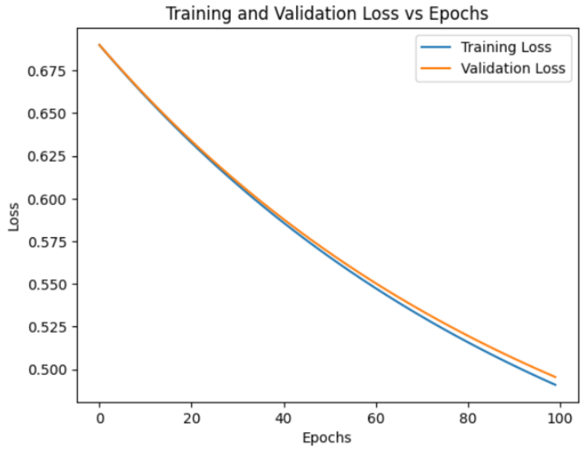


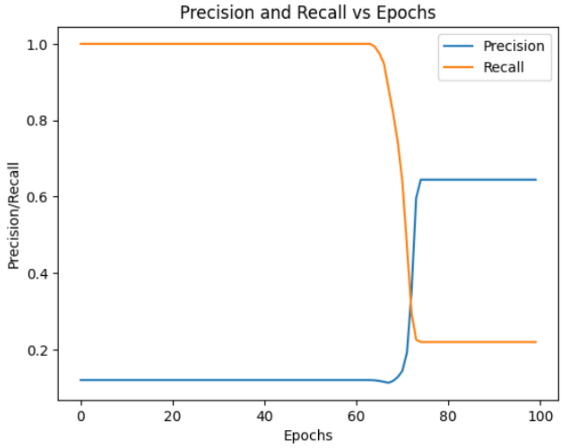
```
  print(f"Actual   0      |  {TN:5d} |  {FP:5d} |")
  print(f"Actual   1      |  {FN:5d} |  {TP:5d} |")
  print("---------------+-------+-------+")

  # Plot 1: Training & Validation Loss
  plt.plot(train_cost, label='Training Loss')
  plt.plot(val_cost, label='Validation Loss')
  plt.xlabel('Epochs')
  plt.ylabel('Loss')
  plt.title('Training and Validation Loss vs Epochs')
  plt.legend()
  plt.show()

  # Plot 2: Precision and Recall vs Epochs
  plt.plot(precision, label='Precision')
  plt.plot(recall, label='Recall')
  plt.xlabel('Epochs')
  plt.ylabel('Precision/Recall')
  plt.title('Precision and Recall vs Epochs')
  plt.legend()
  plt.show()

  # Plot 3: Precision vs Recall
  plt.plot(recall, precision)
  plt.xlabel('Recall')
  plt.ylabel('Precision')
  plt.title('Precision vs Recall')
  plt.show()

  return best_weights, train_cost, val_cost, precision, recall

# Run training
best_weights, train_cost, val_cost, precision, recall = main(
    x1_train_scaled, x2_train_scaled, x3_train_scaled, y_train,
    x1_test_scaled, x2_test_scaled, x3_test_scaled, y_test,
    0, 0, 0, 0, 0.01, 100, 0.38
)
```

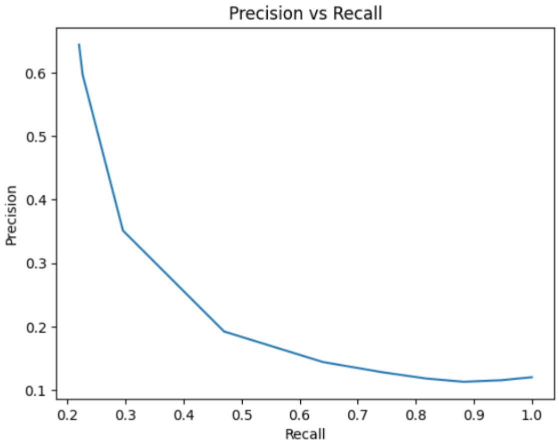
Console Output



```
Training complete.
Best validation cost = 0.49549
Best weights: w1=-0.0856, w2=-0.0093, w3=-0.3125, b=-0.3060

=== Overall Confusion Matrix ===
                 Predicted
               |   0   |   1   |
---------------+-------+-------+
Actual   0     |  5702 |    96 |
Actual   1     |   618 |   174 |
---------------+-------+-------+
```







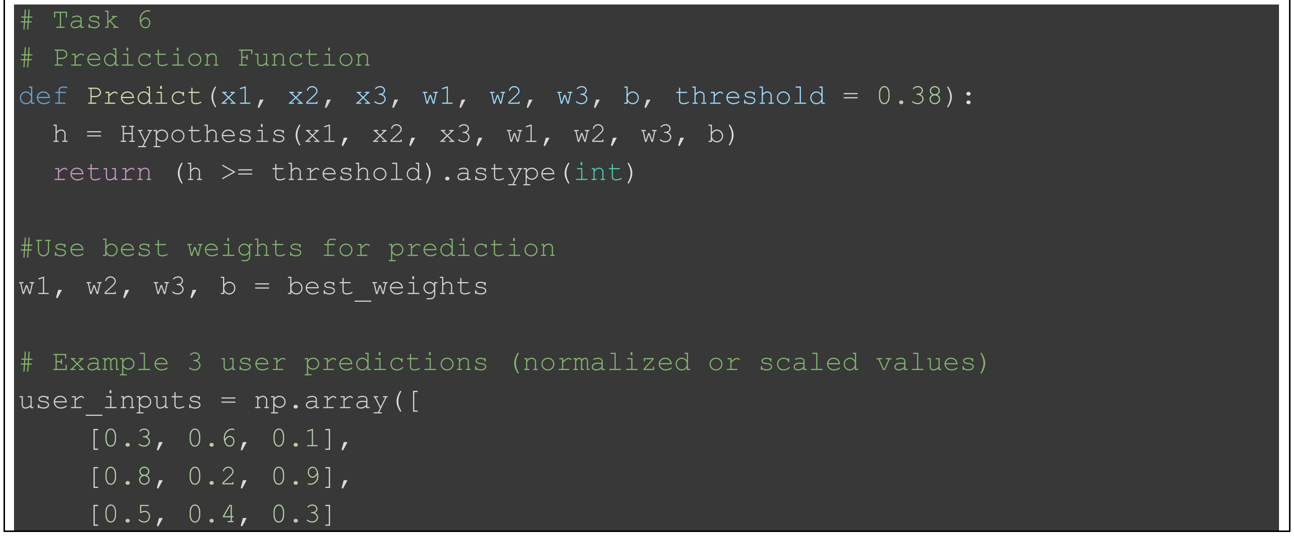
Lab Task 6 – Prediction and Scatter Plot

Save the weights that fit the best model and use them to create a prediction function. The prediction function will take the features as input and output the predicted class of the label. To convert the output to 0 or 1, a threshold of 0.5 needs to be applied to the predicted value h(x). Call your prediction function by giving it some input values to make at least three predictions. Print your predictions and take a screenshot. Additionally, your program must make a scatter plot showing the training and validation examples. The coordinates in the scatter plot correspond to the inputs (x). The class is denoted by:

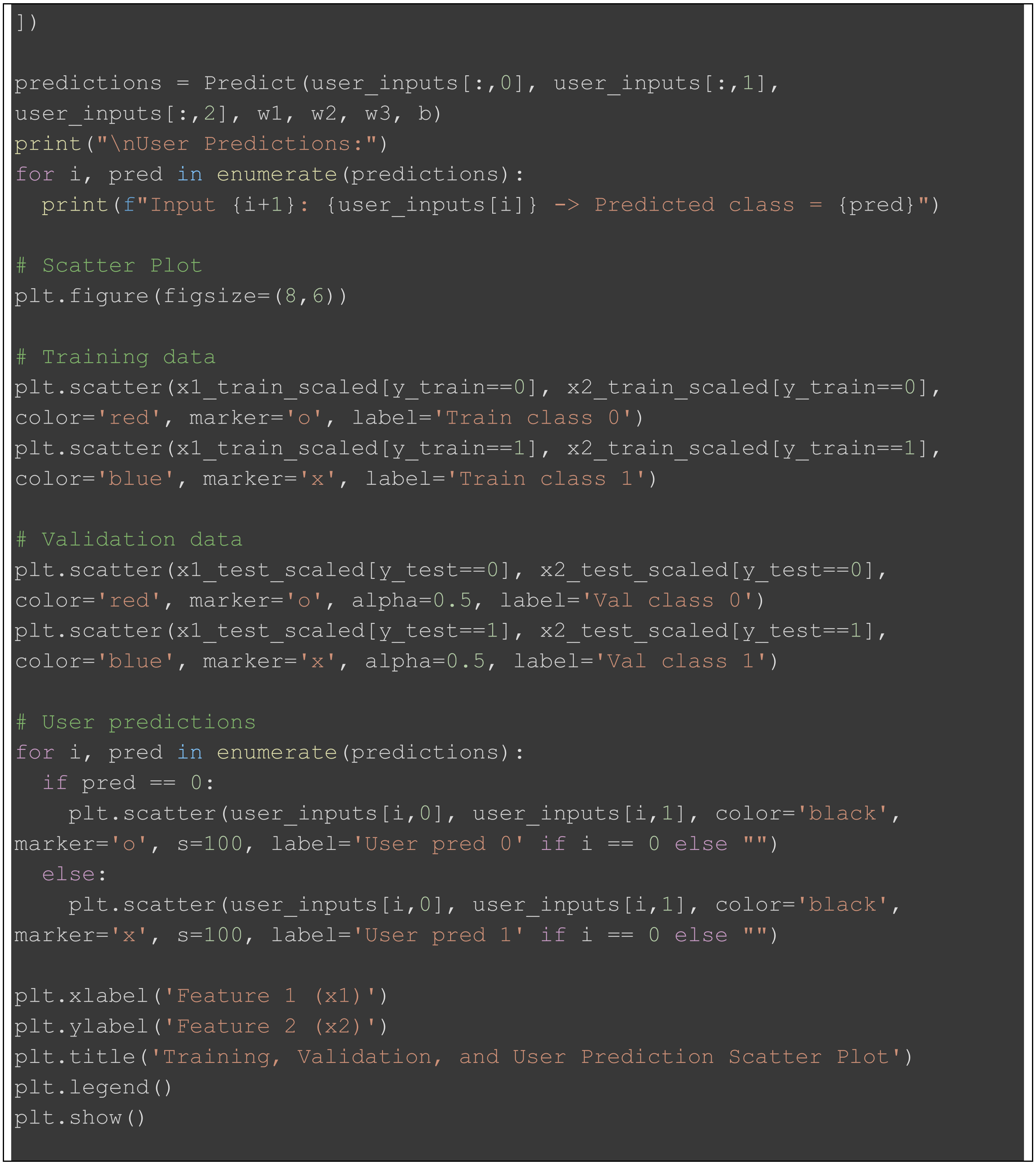
- red “o” for zero class (training and validation examples)
- blue “x” for one class (training and validation examples)
- black “O” for zero class (user prediction)
- black “X” for one class (user prediction)

Provide the code and screenshot of it being used.

Code

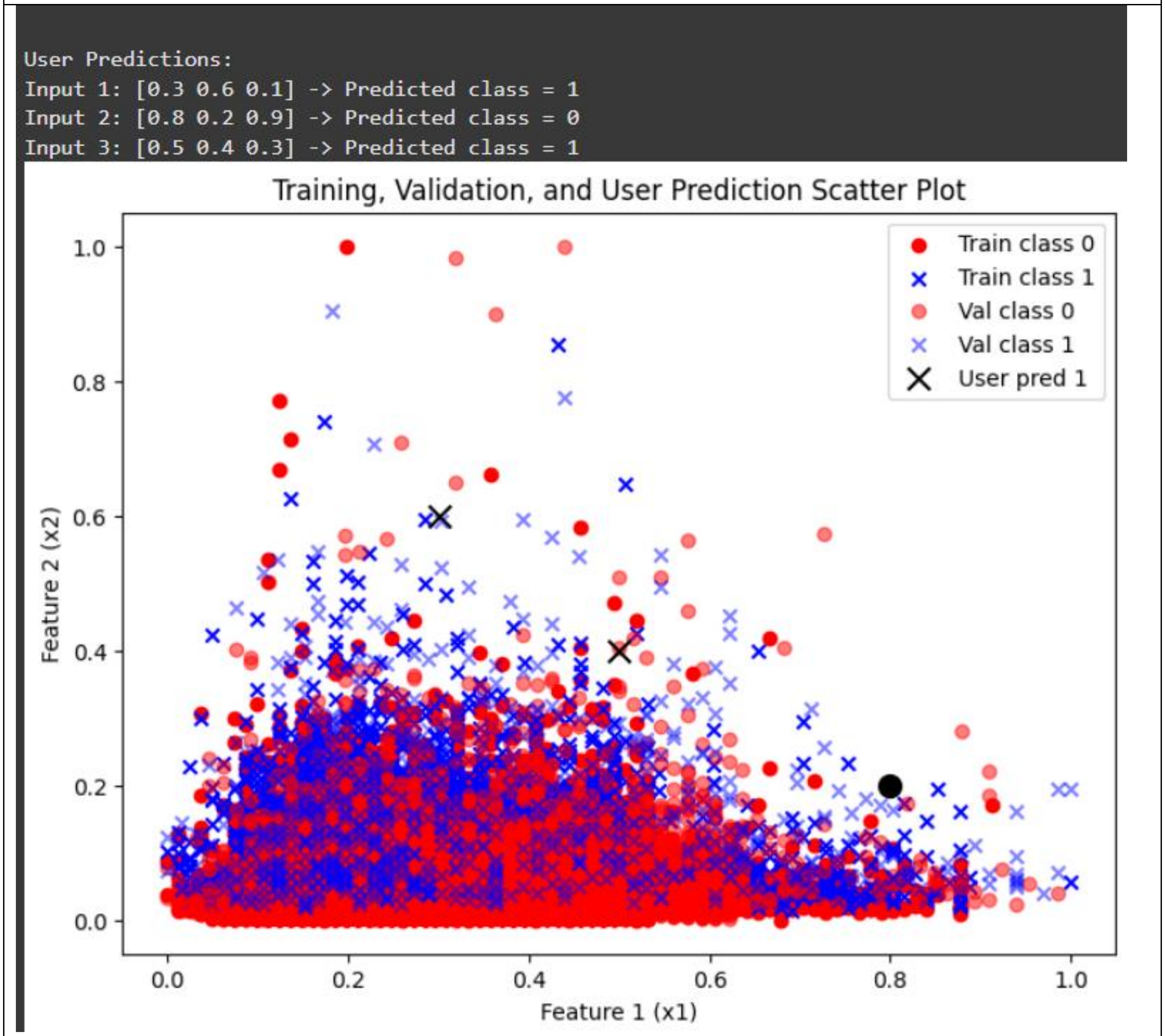


```
# Task 6
# Prediction Function
def Predict(x1, x2, x3, w1, w2, w3, b, threshold = 0.38):
  h = Hypothesis(x1, x2, x3, w1, w2, w3, b)
  return (h >= threshold).astype(int)

#Use best weights for prediction
w1, w2, w3, b = best_weights

# Example 3 user predictions (normalized or scaled values)
user_inputs = np.array([
    [0.3, 0.6, 0.1],
    [0.8, 0.2, 0.9],
    [0.5, 0.4, 0.3]
```



```
])

predictions = Predict(user_inputs[:,0], user_inputs[:,1],
user_inputs[:,2], w1, w2, w3, b)
print("\nUser Predictions:")
for i, pred in enumerate(predictions):
  print(f"Input {i+1}: {user_inputs[i]} -> Predicted class = {pred}")

# Scatter Plot
plt.figure(figsize=(8,6))

# Training data
plt.scatter(x1_train_scaled[y_train==0], x2_train_scaled[y_train==0],
color='red', marker='o', label='Train class 0')
plt.scatter(x1_train_scaled[y_train==1], x2_train_scaled[y_train==1],
color='blue', marker='x', label='Train class 1')

# Validation data
plt.scatter(x1_test_scaled[y_test==0], x2_test_scaled[y_test==0],
color='red', marker='o', alpha=0.5, label='Val class 0')
plt.scatter(x1_test_scaled[y_test==1], x2_test_scaled[y_test==1],
color='blue', marker='x', alpha=0.5, label='Val class 1')

# User predictions
for i, pred in enumerate(predictions):
  if pred == 0:
    plt.scatter(user_inputs[i,0], user_inputs[i,1], color='black',
marker='o', s=100, label='User pred 0' if i == 0 else "")
  else:
    plt.scatter(user_inputs[i,0], user_inputs[i,1], color='black',
marker='x', s=100, label='User pred 1' if i == 0 else "")

plt.xlabel('Feature 1 (x1)')
plt.ylabel('Feature 2 (x2)')
plt.title('Training, Validation, and User Prediction Scatter Plot')
plt.legend()
plt.show()
```

Output Console



```
User Predictions:
Input 1: [0.3 0.6 0.1] -> Predicted class = 1
Input 2: [0.8 0.2 0.9] -> Predicted class = 0
Input 3: [0.5 0.4 0.3] -> Predicted class = 1
```