**Department of Electrical Engineering and   
Computer Science**

**Faculty Member:** Dr. Ahmad Salman **Dated:** 1/11/2023

**Semester:** 7th **Section:** BEE 12C

**CS-471 Machine Learning**

Lab 8: Logistic Regression

**Group Members**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | **PLO4 - CLO4** | | **PLO5 -CLO5** | **PLO8 -CLO6** | **PLO9 -CLO7** |
| **Name** | **Reg. No** | **Viva / Quiz / Lab Performance** | **Analysis of Data in Lab Report** | **Modern Tool Usage** | **Ethics and Safety** | **Individual and Teamwork** |
|  |  | **5 Marks** | **5 Marks** | **5 Marks** | **5 Marks** | **5 Marks** |
| Afif Arif Siddiqi | 344504 |  |  |  |  |  |
| Muhammad Ali Farooq | 331879 |  |  |  |  |  |
| Muhammad Ahmed Mohsin | 333060 |  |  |  |  |  |
| Danial Ahmad | 331388 |  |  |  |  |  |
| Ehtishaam Tanveer | 333074 |  |  |  |  |  |
| Muhammad Umer | 345834 |  |  |  |  |  |

**Table of Contents**

[2 Logistic Regression 3](#_Toc150716156)

[2.1 Introduction 3](#_Toc150716157)

[2.2 Objectives 3](#_Toc150716158)

[2.3 Theory 3](#_Toc150716159)

[3 Lab Tasks 4](#_Toc150716160)

[3.1 Task 1 – Dataset Preparation, Feature Scaling 4](#_Toc150716161)

[3.2 Task 2 – Sigmoid Activation 5](#_Toc150716162)

[3.3 Task 3 – Cost Function 5](#_Toc150716163)

[3.4 Task 4 – Gradient Descent 6](#_Toc150716164)

[3.5 Task 5 – Training and Test Implementation 8](#_Toc150716165)

[3.6 Task 6 – Training and Test Implementation 11](#_Toc150716166)

[4 Conclusion 14](#_Toc150716167)

# Logistic Regression

## 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. Like regression, weight parameters are trained on a dataset 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 test 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 test losses
* Make scatter plots of the classification

## Theory

Logistic regression is a supervised learning algorithm that can be used for binary classification tasks. It works by modeling the probability of a binary outcome, such as yes/no, 0/1, or true/false, based on a set of independent variables. Logistic regression uses a sigmoid function to transform the linear combination of the independent variables into a probability value. The sigmoid function is a non-linear function that squashes its input to a value between 0 and 1. This makes it ideal for modeling probabilities, as the probability of an outcome must be between 0 and 1.

# Lab Tasks

## 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. The “Sale Price” is the label column that your model will predict. The dataset examples are to be divided into 2 separate portions: training and test datasets (choose from 80-20 to 70-30 ratios). Save the prepared datasets as CSV files. Next, load the datasets into your python program and store them as NumPy arrays (Xtrain, ytrain, Xtest, ytest,). Next, use feature scaling to rescale the feature columns of 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).

### TASK 1 CODE STARTS HERE ###

*# Load the dataset into your python program as NumPy arrays (Xtrain ,ytrain).*

dataset = pd.read\_csv(path\_data)

features = ["duration", "age", "job"]

label = "y"

X = dataset[features].values

y = dataset[label].values

*def* feature\_scaling(*X*, *axis*=0):

    return (X - X.min(*axis*=axis)) / (X.max(*axis*=axis) - X.min(*axis*=axis))

*# Split the dataset into training and test datasets*

*def* random\_split(*X*, *y*, *ratio*=0.8, *scaling*=feature\_scaling):

    m = len(X)

    split = *int*(m \* ratio)

    X\_train, y\_train = scaling(X[:split]), y[:split]

    X\_test, y\_test = scaling(X[split:]), y[split:]

    return X\_train, y\_train, X\_test, y\_test

X\_train, y\_train, X\_test, y\_test = random\_split(X, y)

print("X train:")

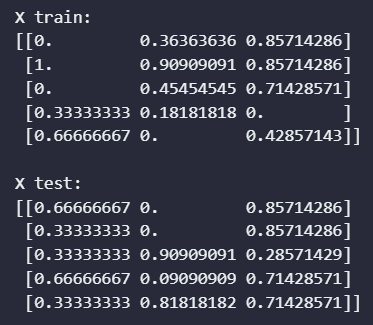
print(X\_train[:5])

print("\nX test:")

print(X\_test[:5])

### TASK 1 CODE ENDS HERE ###

### TASK 1 OUTPUT SCREENSHOT STARTS HERE ###



### TASK 1 OUTPUT SCREENSHOT ENDS HERE ###

## Task 2 – Sigmoid Activation

For logistic regression, you will implement the following hypothesis:

h(x) = g(w0 + w1x1 + w2x2 + w3x3 + …)

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:

### TASK 2 CODE STARTS HERE ###

*def* sigmoid(*z*):

    return 1 / (1 + np.exp(-z))

*# Test the sigmoid function*

print("Input:")

input\_val = np.array([0, 1, 2, 3, 4, 5])

print(input\_val)

print("\nSigmoid Test:")

print(sigmoid(input\_val))

### TASK 2 CODE ENDS HERE ###

## 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 test phases of the program.

cost\_function(X, y)

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

The m is the number of the training/test examples in the dataset. Remember that the hypothesis requires the 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.

### TASK 3 CODE STARTS HERE ###

*def* hypothesis(*X*, *w*):

    return sigmoid(np.dot(X, w))

*def* cost\_function(*X*, *y*, *w*):

    m = len(X)

    h = hypothesis(X, w)

    return 1/m \* np.sum(-y \* np.log(h) - (1 - y) \* np.log(1 - h))

*# Initialize the weights to random values between 0 and 1.*

w = np.random.rand(X\_train.shape[1])

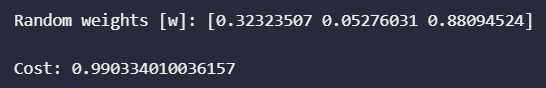
*# Verify the cost function*

print("Random weights [w]: ", w)

print("\nCost: ", cost\_function(X\_train, y\_train, w))

### TASK 3 CODE ENDS HERE ###

### TASK 3 OUTPUT SCREENSHOT STARTS HERE ###



### TASK 3 OUTPUT SCREENSHOT ENDS HERE ###

## 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:

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 hypothesis for the logistic regression is different from that of linear regression. Provide the code and all relevant screenshots showcasing the use of your gradient descent function.

### TASK 4 CODE STARTS HERE ###

*def* gradient\_descent(*X*, *y*, *w*, *alpha*):

    m = len(X)

    h = hypothesis(X, w)

    dw = (1 / m) \* np.dot(X.T, (h - y))

    w -= alpha \* dw

    return w

*# Initialize the weights to random values between 0 and 1.*

w = np.random.rand(X\_train.shape[1])

print("Random weights [w]: ", w)

alpha = 0.01

*# Verify the gradient descent function*

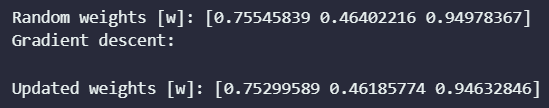
w = gradient\_descent(X\_train, y\_train, w, alpha)

print("Gradient descent: ")

print("\nUpdated weights [w]: ", w)

### TASK 4 CODE ENDS HERE ###

### TASK 4 OUTPUT SCREENSHOTS START HERE ###



### TASK 4 OUTPUT SCREENSHOTS START HERE ###

## Task 5 – Training and Test 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 testing. 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 test examples to determine the test loss. This single iteration over the entire dataset (both training and test) marks completion of one epoch. You will need to perform the training and testing over several epochs (the epoch number is another hyperparameter that must be chosen). Ensure that at the end of each epoch, the training loss, test loss, precision value and recall value are stored for plotting purposes. To get the precision and recall, use the following:

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

1. training loss and test 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.

### TASK 5 CODE STARTS HERE ###

*def* get\_metrics(*y\_true*, *y\_pred*):

    tp = np.sum((y\_pred == 1) & (y\_true == 1))

    fp = np.sum((y\_pred == 1) & (y\_true == 0))

    tn = np.sum((y\_pred == 0) & (y\_true == 0))

    fn = np.sum((y\_pred == 0) & (y\_true == 1))

    return tp, fp, tn, fn

*def* main(*X\_train*, *y\_train*, *X\_test*, *y\_test*, *alpha*, *epochs*):

    w = np.random.rand(X\_train.shape[1])

    train\_loss = []

    test\_loss = []

    precision\_list = []

    recall\_list = []

    for epoch in range(epochs):

*# Training phase*

        w = gradient\_descent(X\_train, y\_train, w, alpha)

        train\_loss.append(cost\_function(X\_train, y\_train, w))

*# Testing phase*

        test\_loss.append(cost\_function(X\_test, y\_test, w))

        y\_test\_pred = hypothesis(X\_test, w) >= 0.35

        y\_test\_pred = y\_test\_pred.astype(*int*)

*# Calculate precision and recall*

        tp, fp, tn, fn = get\_metrics(y\_test, y\_test\_pred)

        precision\_val = tp / (tp + fp) if tp + fp != 0 else 0

        recall\_val = tp / (tp + fn) if tp + fn != 0 else 0

        precision\_list.append(precision\_val)

        recall\_list.append(recall\_val)

    return train\_loss, test\_loss, precision\_list, recall\_list, w

max\_epochs = 200

alpha = 0.1

train\_loss, test\_loss, precision\_list, recall\_list, w = main(

    X\_train, y\_train, X\_test, y\_test, alpha, max\_epochs

)

*# Training and test loss vs. epoch number*

plt.plot(train\_loss, *label*="Training Loss")

plt.plot(test\_loss, *label*="Test Loss", *linestyle*="", *marker*="x", *markevery*=25)

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.legend()

plt.grid(*alpha*=0.35)

plt.show()

*# Precision and recall vs. epoch number*

plt.plot(precision\_list, *label*="Precision")

plt.plot(recall\_list, *label*="Recall")

plt.xlabel("Epochs")

plt.ylabel("Precision/Recall")

plt.legend()

plt.grid(*alpha*=0.35)

plt.show()

*# Precision vs. recall*

plt.plot(recall\_list, precision\_list)

plt.xlabel("Precision")

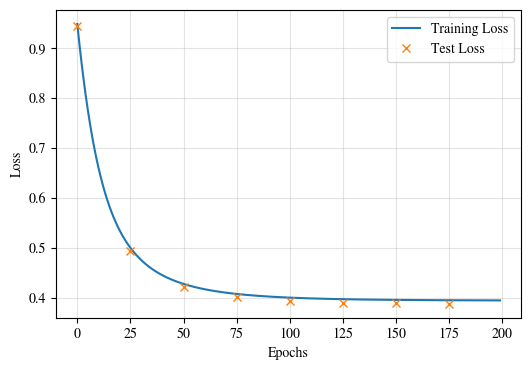
plt.ylabel("Recall")

plt.grid(*alpha*=0.35)

plt.show()

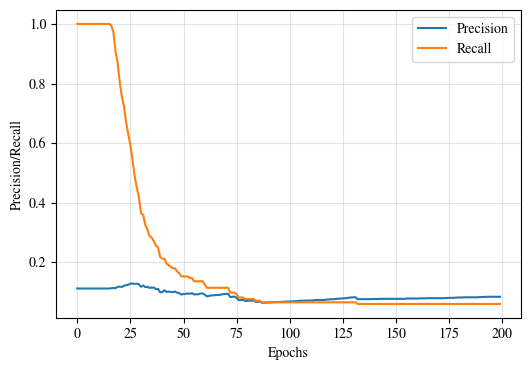
### TASK 5 CODE ENDS HERE ###

### TASK 5 PLOT 1 STARTS HERE ###



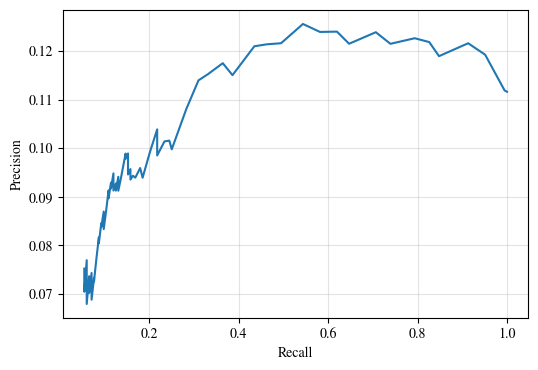
### TASK 5 PLOT 1 ENDS HERE ###

### TASK 5 PLOT 2 STARTS HERE ###



### TASK 5 PLOT 2 ENDS HERE ###

### TASK 5 PLOT 3 STARTS HERE ###



### TASK 5 PLOT 3 ENDS HERE ###

## Task 6 – Training and Test Implementation

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. Additionally, the prediction function must make a scatter plot showing the training and test examples. The coordinates in the scatter plot correspond to the inputs (x). The class is denoted by (red) O and (blue) X for 0 and 1 respectively. The predicted value must be shown as a black O or X. Provide the code and screenshot of is being used.

### TASK 6 CODE STARTS HERE ###

*# Save the weights that fit the best model*

np.save("weights.npy", w)

*# Load the weights and bias*

w = np.load("weights.npy")

*# Prediction function*

*def* predict(*X*, *w*):

    return hypothesis(X, w) >= 0.5

*# Use 30 samples for the scatter plot*

indices = np.random.choice(len(X\_test), 50)

X\_scat = X\_test[indices]

y\_scat = y\_test[indices]

*# Scatter plot*

plt.figure(*figsize*=(8, 6))

plt.title("Scatter Plot")

*# Plot examples*

plt.scatter(

    X\_scat[y\_scat == 0][:, 0],

    X\_scat[y\_scat == 0][:, 1],

*color*="red",

*marker*="o",

*label*="Class 0",

)

plt.scatter(

    X\_scat[y\_scat == 1][:, 0],

    X\_scat[y\_scat == 1][:, 1],

*color*="blue",

*marker*="x",

*label*="Class 1",

)

plt.xlabel("Feature 1")

plt.ylabel("Feature 2")

plt.legend(*loc*="upper right")

plt.show()

*# Make predictions using the best weights*

y\_pred = predict(X\_scat, w)

y\_pred\_class = (y\_pred).astype(*int*)

*# Plot the predictions*

plt.figure(*figsize*=(8, 6))

plt.title("Scatter Plot with Predictions")

plt.scatter(

    X\_scat[y\_scat == 0][:, 0],

    X\_scat[y\_scat == 0][:, 1],

*color*="red",

*marker*="o",

*label*="Class 0",

)

plt.scatter(

    X\_scat[y\_scat == 1][:, 0],

    X\_scat[y\_scat == 1][:, 1],

*color*="blue",

*marker*="x",

*label*="Class 1",

)

plt.scatter(

    X\_scat[y\_pred\_class == 0][:, 0],

    X\_scat[y\_pred\_class == 0][:, 1],

*color*="black",

*marker*="o",

*label*="Predicted Class 0",

)

plt.scatter(

    X\_scat[y\_pred\_class == 1][:, 0],

    X\_scat[y\_pred\_class == 1][:, 1],

*color*="black",

*marker*="x",

*label*="Predicted Class 1",

)

plt.xlabel("Feature 1")

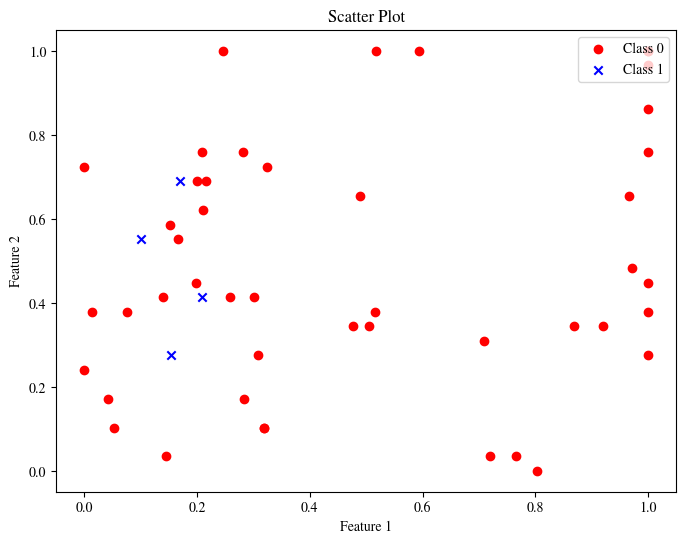
plt.ylabel("Feature 2")

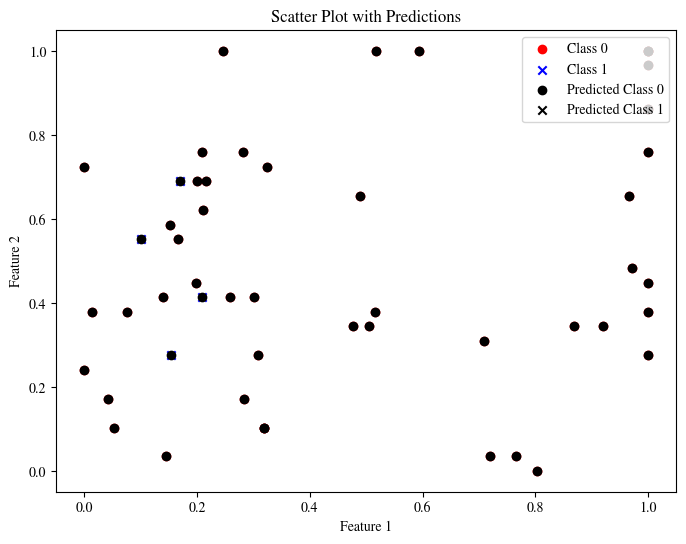
plt.legend(*loc*="upper right")

plt.show()

### TASK 6 CODE ENDS HERE ###

### TASK 6 SCREENSHOT STARTS HERE ###





### TASK 6 SCREENSHOT ENDS HERE ###

# Conclusion

In conclusion, we have reviewed the basic theory of logistic regression, a supervised learning algorithm that can be used for binary classification tasks. We discussed how logistic regression models the probability of a binary outcome based on a set of independent variables, and how it uses a sigmoid function to transform the linear combination of the independent variables into a probability value. We also discussed how logistic regression can be used to predict the probability of the binary outcome for new data points.