Name: Bhargav Chhaya Student ID: 22223499 MSc Data Analytics (1CSD1)
Name: Jash Prakash Rana Student ID: 22222806 MSc Artificial Intelligence(1MAI1)

Assignment 2

Aim: The goal of this assignment is to read, understand, implement, and evaluate the machine learning algorithms called Perceptron and Multi-Layer Perceptron.

Theory/Working:

Neural Networks in Machine Learning topic is a varied topic, but it stems from the simple idea of the nervous system in the biological context, where the brain sends the input to a muscle or an organ through something called 'neurons' which are responsible for the processing of the input and sending it to the right organ/muscle. In a simple neural network system in Machine Learning, the user sends an input, and the nodes (like neurons) are responsible for calculations and output predictions given a training on a historical data and their labels. A neural network link that contains computations to track features and uses Artificial Intelligence in the input data is known as Perceptron. This neural links to the artificial neurons using simple logic gates with binary outputs. An artificial neuron invokes the mathematical function and has node, input, weights, and output equivalent to the cell nucleus, dendrites, synapse, and axon, respectively, compared to a biological neuron. [1]

In our assignment, we are assigned with the wildfires dataset which contains fire as the target label with two labels, yes{1} and no{0}. There are 9 features in the dataset defining the numerical data which determined the conditions during a fire, ranging from temp, humidity, rainfall, drought_code, buildup_index, etc. The values range from 0-1 in some features, while some features have values ranging from 0-50 or so, hence we have also used StandardScaler() to normalize the data. We are expected to train, validate using k fold cross validation, and test the data, so we have taken 67-33% train-test data, and will take 5 splits for k fold cross validation. This dataset is imbalanced with the label value counts as:

```
y = data['fire_type']
y.value_counts()
✓ 0.8s
1    107
0     97
Name: fire_type, dtype: int64
```

Figure A Unique value counts in the label set.

There are two types of Perceptron used in this assignment:

1. Single Layer Perceptron:

A single layer Perceptron (figure 1.A) is a binary classification algorithm capable only of learning linearly separable patterns. As the name suggests, they are a type of supervised learning algorithm which contains only one layer containing the net input function and then an activation function which predicts the new data using the processing power and training from the old data. Some of the jargons around single layer perceptron is:

A. Input Layers: The input layer in Perceptron is made of features in the dataset taken into the system for further processing. The number of neurons is equal to the number of features in the dataset
B. Weights: They determine the importance of a feature in the whole dataset i.e., higher the number, higher the importance of a feature. The single layer perceptron doesn't have prior knowledge, so the initial weights are assigned randomly and then adjusted to minimize errors using the perceptron learning algorithm while predicting labels for new data.
C. Bias: It is the same as intercept in linear equations, and is an additional parameter used to modify the output to give correct predictions after the training along with the weighted sum of the inputs. We will initialize the bias = 0.
D. Net Sum: Calculates the total sum of the product of the weights and the inputs .
E. Activation Function: A neuron can be activated (1) or not (0) and activation function is the determiner. The weighted sums and bias are compared with the activation function to give the predictions. We will use the sigmoid function as our activation function. The sigmoid function is used as it outputs probabilities between zero and one which is helpful for binary classification tasks as inputs spread over a large range of values will be reduced to probabilities between zero and one thus making it easy to set a threshold value for classification. The input to the activation function is the net sum plus the bias .
F. Output: the output of a single layer perceptron is calculated by the equation *y= activation_function(w.x +b)*
G. Error: The error in the single layer perceptron is calculated *by the difference between the desired output and the actual output.*
H. Architecture of the required neural network: The number of input units are 9 corresponding to the number of independent variables. The number of weights is 9 corresponding to the number of input units , the activation function used is sigmoid . the output is the dependent variable "fire" which has two values "yes" or "no" thus making the given problem a binary classification problem. The threshold is set equal to the bias.

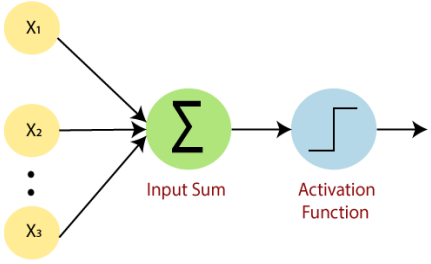

Figure 1.A Single Layer Perceptron.

$$A = sigmoid(z) = \frac{1}{1+e^{-z}}$$

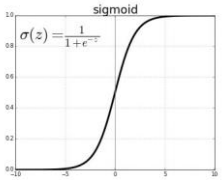

Figure 1.B Sigmoid function.

We set a learning rate and iteration rate when initializing an object of the Perceptron class. The learning rate is the adjustment value which adjusts the error by modifying the weights and bias by additions or subtractions. To get the best accuracy score, we set an iteration rate which helps to adjust the weights during those iterations. It's not always ideal to give a really big number but experimenting with the iteration value can help in a good scoring of the weights and bias for predictions.

We initialize weight *w* with some random vector. We then iterate over all the examples in the data, (*P* U *N*) both positive and negative examples. Now if an input *x* belongs to *P*, ideally the dot product *w.x* should be greater than or equal to 0. And if x belongs to N, the dot product must be less than 0.

Case 1: When *x* belongs to *P* and its dot product $w.x < 0$

Case 2: When *x* belongs to *N* and its dot product $w.x \geq 0$

Only for these cases, we are updating our randomly initialized *w*. Otherwise, we don't touch **w** at all because Case 1 and Case 2 are violating the very rule of a perceptron. So, we are adding ***x*** to ***w*** (ahem vector addition ahem) in Case 1 and subtracting *x* from *w* in Case 2. [2]

The input layers provide each row according to the features, and the weighted sums and biases are added to the equation to calculate the net sum & passed to the activation function. The output is then compared to the threshold (usually the bias) which converts the values to a predicted class. To validate the data, we pass through the cross-validation function with folds = 5 which calculates the accuracy of the output with the labels, and then we select the best accuracy score model and test on that data.

```
scores = []
validate = 0
for train, test in cv.split(X_train, y_train):
    per.fit(X_train[train], y_train[train])
    score = accuracy_score(y_train[test], per.predict(X_train[test]))
    scores.append(score)
    for i in scores:
        if i > validate:
            validate = i
            slp = per.fit(X_train[train],y_train[train])

print(f"Scores: {scores}")
print(f"Best Score: {validate}")
print(f"Model Taken: {slp}")
```

✓ 4.7s

```
Scores: [0.8571428571428571, 0.9259259259259259, 0.7777777777777778, 0.8888888888888888, 0.8148148148148148]
Best Score: 0.9259259259259259
Model Taken: <__main__.Perceptron object at 0x00000190CBE86E10>
```

Figure 1.C K-fold Cross Validation results using 5 folds on training data in Single Layer Perceptron.

Once we are done with training the data, and then cross-validation, we predict the data with test set. For each row in test set, we pass the weighted sum and bias as an equation to the activation function, which gives us the probability between 0 and 1, and then we check the classification using comparison with the threshold. If result is less than or equal to the threshold, we predict it as no{0} label else we predict the output as yes{1} label in terms of fire. To find the accuracy score, we compare the output with the testing labels to find the accuracy of the predictions on new data.

```
print(f"Accuracy Score: {accuracy_score(y_test, y_pred)} \n")
```

✓ 0.6s

```
Accuracy Score: 0.8529411764705882
```

```
print(classification_report(y_test, y_pred))
```

✓ 0.6s

	precision	recall	f1-score	support
0	1.00	0.69	0.81	32
1	0.78	1.00	0.88	36
accuracy			0.85	68
macro avg	0.89	0.84	0.85	68
weighted avg	0.88	0.85	0.85	68

Figure 1.D Accuracy and Classification Report calculated on the Perceptron

```
print(f"Accuracy Score: {accuracy_score(y_test, log_pred)} \n")
```

✓ 0.5s

```
Accuracy Score: 0.7794117647058824
```

```
print(classification_report(y_test, log_pred))
```

✓ 0.7s

	precision	recall	f1-score	support
0	0.76	0.78	0.77	32
1	0.80	0.78	0.79	36
accuracy			0.78	68
macro avg	0.78	0.78	0.78	68
weighted avg	0.78	0.78	0.78	68

Figure 1.E Accuracy and Classification Report calculated on Scikit Learn's Logistic Regression.

```
cf_matrix = confusion_matrix(y_test,y_pred)
sns.heatmap(cf_matrix, annot = True)
```

✓ 0.3s

```
<AxesSubplot: >
```

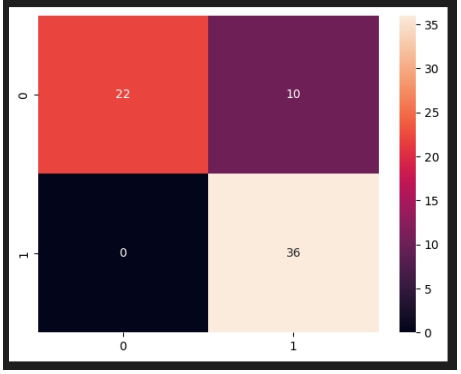

Figure 1.F Confusion Matrix calculated on the Perceptron

```
cf_matrix = confusion_matrix(y_test,log_pred)
sns.heatmap(cf_matrix, annot = True)
```

✓ 0.4s

```
<AxesSubplot: >
```

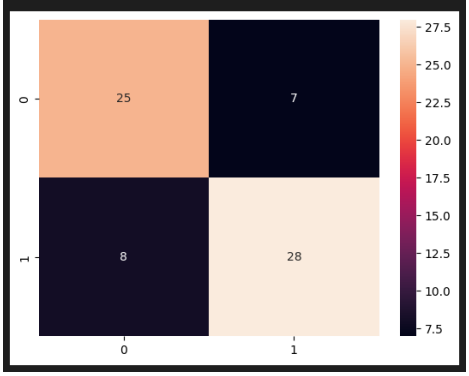

Figure 1.G Confusion Matrix calculated on Scikit Learn's Logistic Regression

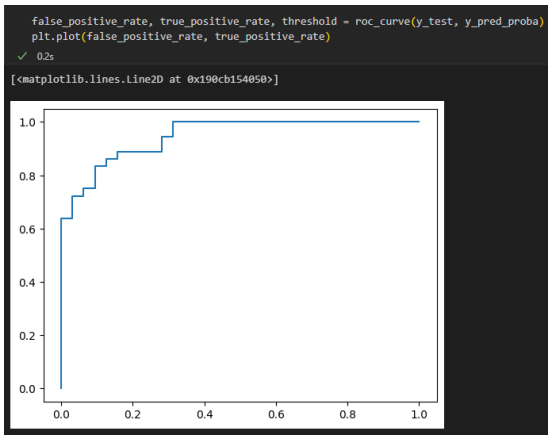

Figure 1.H ROC Curve of the Perceptron model's probability values with y_test

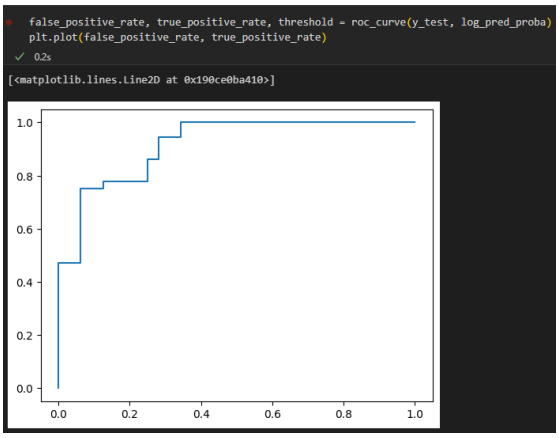

Figure 1.H ROC Curve of the Logistic Regression model's probability values with y_test

Limitations: A single layer perceptron can only implement linearly separable classes. It can execute all logic gates except the XOR gate as the classes aren't linearly separable . Thus, this limitation provides the need for multi-layer perceptron, though it cannot guarantee the best results in every situation, but it tries to calculate with many layers which increases the chances of having a better model.

2. Multi-Layer Perceptron:

A multi-layer Perceptron is a neural network which is a feedforward neural network as the inputs and the weights are multiplied together and the weighted sum is input to the activation function along with the bias . a multilayer perceptron has input layers corresponding to the number of features and output layer corresponding to the dependent variable, but it has also had a number of hidden layers where each hidden layer consists of one or more neurons .

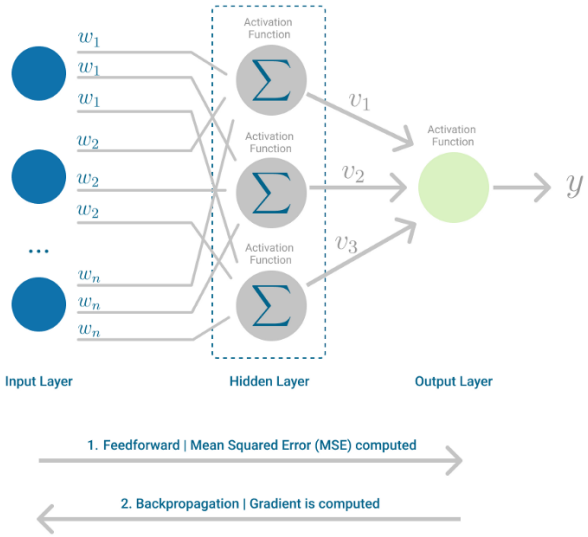

Figure 2.A Multi-Layer Perceptron

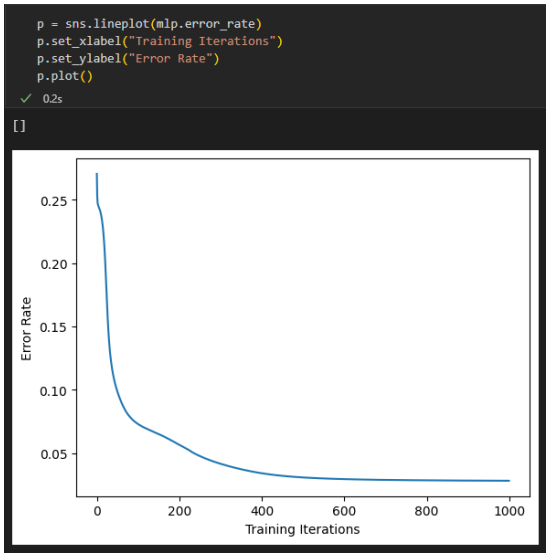

Figure 2.B Error Rate(Loss function) across each iteration in the Perceptron

a. Input layer: The input layer in Perceptron is made of features in the dataset taken into the system for further processing. The number of neurons is equal to the number of features in the dataset.
b. Hidden layers: The hidden layers have one or more neurons stacked inside them . The hidden layers can choose any activation function for the weighted sum . The i^{th} hidden layer output is fed into the $(i+1)^{th}$ hidden layer as an input with the respective weights and bias. The first hidden layer has an input of weighted sum of features and initial weights along with the bias.
c. Output layer: The output layer is corresponding to the number of dependent variables in the dataset . the output of the output layer is the final output of the multi-layer perceptron network. Unlike the hidden layers the output layer cannot chose any arbitrary function as the choice of function is determined by the problem which is being solved. For a binary classification problem, we need a function which has output between range zero and one, the function for the hidden layer and output layer is chosen as sigmoid .
d. Activation functions: We will use the sigmoid function as our activation function. The sigmoid function is used as it outputs probabilities between zero and one which is helpful for binary classification tasks as inputs spread over a large range of values will be reduced to probabilities between zero and one thus making it easy to set a threshold value for classification. The input to the activation function is the net sum plus the bias
e. Loss function: The loss function is the difference between the current output and the actual output as shown in Figure 2.B. The goal of training a model is to reduce the loss function (mean square error).
f. Backpropagation: The algorithm to calculate the gradient vector to adjust the weights and biases is called the backpropagation which computes the partial derivates of the cost function with respect to the weights.
g. Gradient descent: The gradient descent algorithm is used to find minimum of the cost function by computing the gradient and taking small steps along the negative of the gradient .

h. Weights: They determine the importance of a feature in the whole dataset i.e., higher the number, higher the importance of a feature. The weights for the input layer and the weights for the hidden layers are also set randomly but the difference is the input layer weights are multiplied with the inputs and the hidden layer weights are multiplied with the output of the previous hidden layer.

Process of training:

The input layer consists of the number of features in the dataset . the output layers consist of the number of dependent variable so for the multilayer perceptron the number of neurons in the input layer are 9 and output layer are one. The first hidden layer is stacked with 3 neurons which gets an input of the weighted sum i.e., inputs *weights plus bias and the output is then given by an activation function which in the given case is sigmoid thus the output is then feedforward as input to the next neurons multiplied by the weights plus bias for the respective layer . The output of the last hidden layer is then fed into the output layer as an input multiplied by the weights which is fed into the activation function which is sigmoid to give an actual output .

The reason for multi-layer perceptron should having multiple activation functions is to give continuous output to compute the gradient for the gradient descent algorithm for updating the weights. The error for the multilayer perceptron is used by calculating the cost function which is for the given problem is mean squared error which computes the square of the difference between the expected and actual outputs for a single neuron. The final cost function is the mean of the cost functions of each neuron is given by the formula

$$MSE = (1/n) * \Sigma(actual - forecast)^2$$

To find the minimum value of the cost function the gradient of a function will give the direction of the steepest ascent. The negative of the gradient will give the steepest descent . The length of the gradient will give the indication of the slope. The algorithm to minimize the function is to compute the gradient of the function and take small step sizes to reduce the cost function is called backpropagation.

The process of continuously taking small steps along the negative gradient is called gradient descent. Backpropagation relies on the chain rule which is for each neuron it calculates the sensitivity of the cost function with respect to a random weight *w* which is then equal to the sensitivity of the linear output with respect to the weight multiplied by sensitivity of *y* with respect to *w* is also calculated where *y* is output of the activation function and for each output *y* the sensitivity of the cost function with respect to *y*. [3]

The sensitivity of the cost function with respect to the bias is identical where we replace the sensitivity of cost function with respect to random bias *b* and sensitivity of *y* with respect to *b* which is a bias

Since the final cost function is summation of cost function of each layer the final sensitivity of the cost function with the weights will also be summation of the sensitivity of the cost function of all neurons with respect to the weights and the same goes for the sensitivity of the cost function with respect to biases. Thus, the cost function is indirectly sensitive to the activation function of a neuron which introduces the idea of backpropagation. To reduce the cost function, we backpropagate i.e., calculate the influences of the *n-1* th hidden layer on the n^{th} layer and to get the desired outputs the weights are changed such that the output neuron matches the desired and actual outputs. Thus the weight changes are proportional to the larger activation values i.e. based on the idea of neurons that wire together fire together and the one's which are to be desired to be active. The sensitivity of given variable with respect to another variable is calculated as the partial derivative of the given variable with respect to the second variable [4]. Backpropagation can be summed up using the equations in figure 2.B.

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \quad \text{(BP1)}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \quad \text{(BP2)}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \quad \text{(BP3)}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \quad \text{(BP4)}$$

Figure 2.C Summary of the Equations used for Back propagation

```
scores = []
validate = 0
X_new = None
y_new = None
for train, test in cv.split(X_train, y_train):
    per.fit(X_train[train], pd.DataFrame(y_train[train]), epochs = 1000, learning_rate = 0.1)
    score = accuracy_score(y_train[test], per.predict(X_train[test]))
    scores.append(score)
    for i in scores:
        if i > validate:
            validate = i
            X_new = X_train[train]
            y_new = pd.DataFrame(y_train[train])
mlp.fit(X_new, y_new, epochs = 1000, learning_rate = 0.1)

print(f"Scores: {scores}")
print(f"Best Score: {validate}")
print(f"Model Taken: {mlp}")
```

```
✓ 45.9s

Scores: [0.7857142857142857, 0.9259259259259259, 0.9259259259259259, 0.9629629629629629, 0.9629629629629629]
Best Score: 0.9629629629629629
Model Taken: <__main__.MultiLayerPerceptron object at 0x00000142CBFA5AD0>
```

Figure 2.D Cross Validation on the Training Dataset using 5 folds.

After creating the Multi-Layer Perceptron class defining all the parameters and functions and initializing things, we perform cross validation on the training set and choose the best model and fit on that data. The output is then compared to the threshold (usually the bias) which converts the values to a predicted class. For this example, we set a threshold to 0.5 as we are passing each inputs through multiple nodes. To validate the data, we pass through the cross-validation function with folds = 5 which calculates the accuracy of the output with the labels, and then we select the best accuracy score model and test on that data. (see figure 2.C)

Once we are done with training the data, and then cross-validation, we predict the data with test set. For each row in test set, we pass the weighted sum and bias as an equation to the forward propagate function, which gives us the probability between 0 and 1 using sigmoid and sigmoid derivative functions, and then we check the classification using comparison with the threshold. If result is less than or equal to the threshold, we predict it as no{0} label else we predict the output as yes{1} label in terms of fire. To find the accuracy score, we compare the output with the testing labels to find the accuracy of the predictions on new data. Figure 2.E to 2.J shows the results in the best format along with the accuracy score, confusion matrix and roc curve while comparing it with Scikit Learn's Logistic Regression model.

Limitations: A multi-layer perceptron works for n-epochs and changes a lot of data between each node in each epoch. That creates a constraint on computation time and the higher the number of nodes and number of iterations, the more computation time required. And it is not guaranteed that it will give the best results so that really just brings a question to the table, which is "Is the computation time and the designing of multi-layer perceptron for a single solution worth it?" Also, it has too many parameters to be taken care of and is fully connected so that really brings up a bunch of stuff to be taken care of for a single solution.

```
print(f"Accuracy Score: {accuracy_score(y_test, y_pred)} \n")
✓ 0.3s
Accuracy Score: 0.8382352941176471

print(classification_report(y_test, y_pred))
✓ 0.3s
              precision    recall  f1-score   support

           0       0.92      0.72      0.81        32
           1       0.79      0.94      0.86        36

    accuracy                           0.84        68
   macro avg       0.86      0.83      0.83        68
weighted avg       0.85      0.84      0.84        68
```

Figure 2.E Accuracy score and classification report on the Multi-Layer Perceptron predictions.

```
print(f"Accuracy Score: {accuracy_score(y_test, log_pred)} \n")
✓ 0.3s
Accuracy Score: 0.7794117647058824

print(classification_report(y_test, log_pred))
✓ 0.3s
              precision    recall  f1-score   support

           0       0.76      0.78      0.77        32
           1       0.80      0.78      0.79        36

    accuracy                           0.78        68
   macro avg       0.78      0.78      0.78        68
weighted avg       0.78      0.78      0.78        68
```

Figure 2.F Accuracy score and classification report on Logistic Regression library predictions.

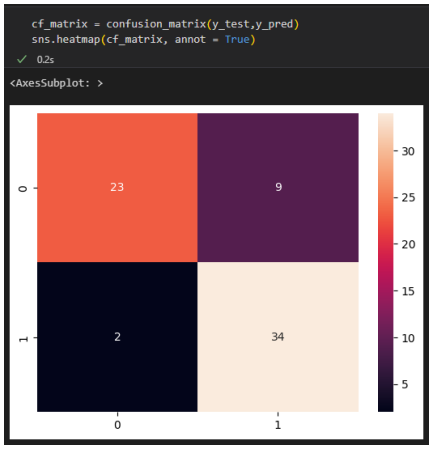

Figure 2.G Classification Matrix of the Multi-Layer Perceptron predictions.

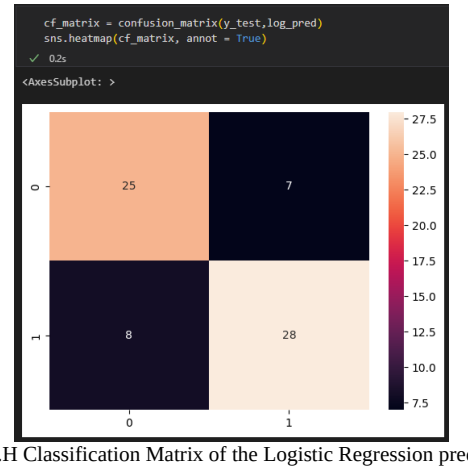

Figure 2.H Classification Matrix of the Logistic Regression predictions.

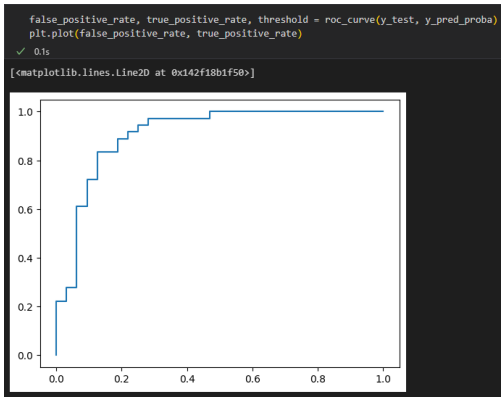

Figure 2.I ROC curve of the Multi-Layer Perception predictions.

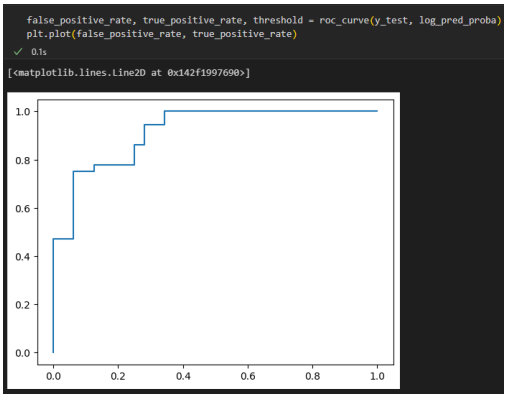

Figure 2.J ROC curve of the Logistic Regression predictions.

Conclusion: By conducting experimental test cases and tuning different parameters, we come to a conclusion that even though Multi-Layer Perception has higher capabilities, it may not always be the best use for you, it may sometime be defeated by the simple Single Layer Perceptron. Overall, Multi-Layer Perceptron has one of the best ideologies to make a machine learn and function on its own, and both the Perceptron worked better than the standard Logistic Regression function of the Scikit-Learn library meaning that the learning feature and weight adjustment feature of the Perceptron makes it robust and tuneable for each and every type of data and can be well suited to be used as an effective classifier to solve, even though it carries the risk of high cost of computation power and time consumption.

References

[1] M. Banoula, “What is Perceptron: A Beginners Guide for Perceptron,” Simpli Learn, 28 October 2022. [Online]. Available: https://www.simplilearn.com/tutorials/deep-learning-tutorial/perceptron.

[2] A. L. Chandra, “Perceptron Learning Algorithm: A Graphical Explanation Of Why It Works,” Towards Data Science, 22 August 2018. [Online]. Available: https://towardsdatascience.com/perceptron-learning-algorithm-d5db0deab975.

[3] A. F. Ali Ghodsi, “Statistical Learning- Classifcation,” 03 October 2007. [Online]. Available: https://sas.uwaterloo.ca/~aghodsib/courses/f07stat841/notes/lecture10.pdf.

[4] 3Blue1Brown, “ What is backpropagation really doing? | Chapter 3, Deep learning,” YouTube, [Online]. Available: https://www.youtube.com/watch?v=Ilg3gGewQ5U.

Work Division:

1. Bhargav Chhaya:
    - Pre-processing & Train-Test split.
    - Sigmoid Function
    - Sigmoid Derivative Function
    - Back Propagation Function
    - Gradient Descent Function
    - Accuracy Score
    - Classification Report

2. Jash Prakash Rana:
    - Data conversion into Excel file
    - Classifier Function
    - Mean Squared Error Function
    - Forward Propagation
    - K-Fold Cross Validation
    - Confusion Matrix Visualization
    - ROC Curve Visualization

3. Equal Contribution in:
    - Constructor Function
    - Fit Function
    - Predict Function
    - Scaler model decision
    - Parameter selection
    - Report Generation

```
#Single Layer Perceptron
class Perceptron:

  #1. Define an initial constructor
  def __init__ (self, learning_rate = 0.1, n_iters = 1000):
    self.weights = None
    self.bias = None
    self.iterations = n_iters
    self.learning_rate = learning_rate

  #sigmoid function
  def sigmoid(self, x):
    return (1.0 / (1.0 + np.exp(-x)))   #Reference:
    https://www.digitalocean.com/community/tutorials/sigmoid-activation-function-python

  #Classification based on comparison between activation result and bias
  def classifier(self, activation):
    threshold = self.bias
    if activation <= threshold:
      return 0
    else:
      return 1

  #Train the single neural node using the fit method to make it ready for use ahead
  def fit(self, X, Y):
    #Making an array of ones for each column in the dataframe
    self.weights = np.random.rand(X.shape[1])

    #Initial assignment of bias to calculate further
    self.bias = 0

    #Iterating for 'n_iters' time to adjust weight and bias each time we fit the data
    for i in range(self.iterations):
      for row, label in zip(X, Y):
        result = self.sigmoid(np.dot(self.weights, row) + self.bias)
        y_pred = self.classifier(result)

        #To minimise error, we adjust the weights and the bias according to the learning
        rate
        if label == 1 and y_pred == 0:
          self.weights = self.weights + self.learning_rate * row
          self.bias = self.bias - self.learning_rate * 1
        elif label == 0 and y_pred == 1:
          self.weights = self.weights - self.learning_rate * row
          self.bias = self.bias + self.learning_rate * 1

    return self

  #Defining the predict method to pass new data and calculating labels for them using
  the existing model
  def predict(self, new_data):
    #List to hold the predictions given for 'n' rows of data
    predictions = []
    self.pred_proba = []

    #Predicting label for each row and appending it to the list
    for row in new_data:
      result = self.sigmoid(np.dot(self.weights, row) + self.bias)
      self.pred_proba.append(result)
      predictions.append(self.classifier(result))

    #Returning an Numpy Array of predictions
    return np.array(predictions)

---------------------------------------
#Multi-Layer Perceptron
```

```
class MultiLayerPerceptron():

    def __init__(self, num_inputs=3, hidden_layers=[3, 3], num_outputs=2):
        self.num_inputs = num_inputs
        self.hidden_layers = hidden_layers
        self.num_outputs = num_outputs

        # create a generic representation of the layers
        layers = [num_inputs] + hidden_layers + [num_outputs]

        # create random connection weights for the layers
        weights = []
        for i in range(len(layers) - 1):
            w = np.random.rand(layers[i], layers[i + 1])
            weights.append(w)
        self.weights = weights

        # save derivatives per layer
        derivatives = []
        for i in range(len(layers) - 1):
            d = np.zeros((layers[i], layers[i + 1]))
            derivatives.append(d)
        self.derivatives = derivatives

        # save activations per layer
        activations = []
        for i in range(len(layers)):
            a = np.zeros(layers[i])
            activations.append(a)
        self.activations = activations


    def forward_propagate(self, inputs):

        # the input layer activation is just the input itself
        activations = inputs

        # save the activations for backpropogation
        self.activations[0] = activations

        # iterate through the network layers
        for i in range(len(self.weights)):
            # calculate matrix multiplication between previous activation and weight
            matrix
            net_inputs = np.dot(activations, self.weights[i])

            # apply sigmoid activation function
            activations = self.sigmoid(net_inputs)

            # save the activations for backpropogation
            self.activations[i + 1] = activations

        # return output layer activation
        return activations

    def back_propagate(self, error):

        # iterate backwards through the network layers
        for i in reversed(range(len(self.derivatives))):

            # get activation for previous layer
            activations = self.activations[i+1]

            # apply sigmoid derivative function
            delta = error * self.sigmoid_derivative(activations)

```

```
129             # reshape delta as to have it as a 2d array
130             delta_re = delta.reshape(delta.shape[0], -1).T
131 
132             # get activations for current layer
133             current_activations = self.activations[i]
134 
135             # reshape activations as to have them as a 2d column matrix
136             current_activations = current_activations.reshape(current_activations.shape[0
                ],-1)
137 
138             # save derivative after applying matrix multiplication
139             self.derivatives[i] = np.dot(current_activations, delta_re)
140 
141             # backpropogate the next error
142             error = np.dot(delta, self.weights[i].T)
143 
144     #Gradient Descent, used for calculating weights by multiplying derivatives and
145     def gradient_descent(self, learningRate=1):
146 
147         # update the weights by stepping down the gradient
148         for i in range(len(self.weights)):
149             weights = self.weights[i]
150             derivatives = self.derivatives[i]
151             weights += derivatives * learningRate
152 
153     def classifier(self, x):
154         if x <= 0.5:
155             return 0
156         else:
157             return 1
158 
159     #sigmoid function
160     def sigmoid(self, x):
161         return (1.0 / (1.0 + np.exp(-x)))   #Reference:
            https://www.digitalocean.com/community/tutorials/sigmoid-activation-function-pyth
            on
162 
163 
164     def sigmoid_derivative(self, x):
165         return x * (1.0 - x)
166 
167 
168     def mse(self, target, output):
169         return np.average((target - output) ** 2)
170 
171 
172     def fit(self, inputs, targets, epochs, learning_rate):
173 
174 
175         targets=targets.to_numpy()
176 
177         self.error_rate = []
178         for i in range(epochs):
179             sum_errors = 0
180 
181             # iterate through all the training data
182             for j, input in enumerate(inputs):
183                 target = targets[j]
184 
185                 # activate the network!
186                 output = self.forward_propagate(input)
187 
188                 error = target - output
189 
190                 self.back_propagate(error)
191 
```

```
                # now perform gradient descent on the derivatives
                # (this will update the weights
                self.gradient_descent(learning_rate)

                # keep track of the MSE for reporting later
                sum_errors += self.mse(target, output)

            # Epoch complete, report the training error
            self.error_rate.append(sum_errors / len(inputs))

        return self

    def predict(self,new_data):
        self.result = []
        self.pred_proba = []
        temp = self.forward_propagate(new_data)


        for i in temp:
            for point in i:
                self.pred_proba.append(point)
                self.result.append(self.classifier(point))

        return np.array(self.result)
```