Assignment 2

Name: Anushka Anil Padwal Name: Smitesh Nitin Patil

Student ID: 22222312 Student ID: 22223696

Class : 1CSD1 Class : 1CSD1

In this study, we have documented our efforts to create a perceptron and multi-layer perceptron from scratch and compare them to pre-defined models from the sci-kit learn library.

* Creating a Model for Perceptron –

@Anushka Anil Padwal

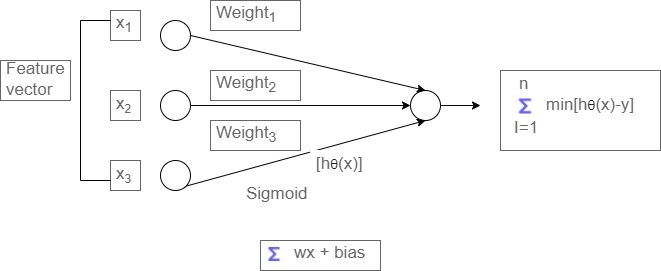
Perceptron is a supervised learning algorithm; it is the basic building block of a neural network. A linear classifier that classifies given input into binary outputs.

1. Structure of a perceptron:

In perceptron, we take our features or independent variables as input. For example, if we have 10 features, we have 10 input nodes in our perceptron. These features have certain weights associated with them which determine how strongly that independent variable is associated with the output. The output is calculated by taking a linear combination of these weights and features along with a bias. This output is not a distinct value hence it is fed to an activation function like tanh, sigmoid, or relu so that the output is in a continuous scale between two values.

For this study we had a dataset with many features we selected [‘temp’, ‘drought\_code’, ‘buildup\_index’] as they are most correlated to the dependent variable that predicts wildfire.





So, these 3 features would be input to our model and the hypothesis will be generated on these three input features.

1. Feedforward & adjusting weights:

During the training process, we initially randomise weights and feed training samples to the models to generate the hypotheses, then this hypotheses value is compared with the actual value from the training sample, and the error generated is used to adjust weights and bias. the adjustment of the weights is done using a learning parameter ‘alpha’. The smaller the alpha value the longer it takes to converge and if the alpha value is high it may not converge at all, and the model generated would be under fitted.

For this study, we have taken the value of alpha or ‘learning rate’ as 0.00001 and sigmoid as the activation function.

* Creating model for Multi-Layer Perceptron –

@Smitesh Nitin Patil

A multi-layer perceptron is made of many perceptrons connected with hidden layers to develop complex non-linear hypotheses that would be hard for a single perceptron to comprehend. Various architectures of multi-layer perceptron are used for machine learning tasks. Like RNN(Recurrent Neural Networks) for Natural Language Processing and CNN (Convolutional Neural Networks) for image classification.

1. Structure and design decision:

In this implementation of multi-layer perceptron, we have taken three features like we did while implementing perceptron for similar reasons. These three input nodes are then connected to a hidden layer with 25 nodes after applying the activation function. Then the weights of this hidden layer are used to generate our hypotheses after applying the activation function.

Diagram

Description automatically generated

1. Forward Pass:

While going through training examples we initially randomize the weights of all the layers while traversing through all the layers after assigning the weights we also assign them to an activation function and generate the hypotheses.

Hence for this model, we traverse the network in the following way:

Input -> Activation() -> Hidden\_layer() -> Activation() -> binary\_output

1. Backward Pass:

After generating hypotheses we calculate the error using a loss function. In this model, we have used Mean Squared Error to calculate the error value. This is later used to calculate the partial derivatives of the previous nodes. In other words, to adjust the weights of a node we need to calculate the difference in the weight concerning the partial derivatives of the previous nodes. The value of the difference is regulated using the learning parameter ‘alpha’. The same process for bias.

1. Adjusting the weights and biases for nodes in the models:
2. Loss Function(Mean Squared Error):

The value of alpha determines how our model converges to global minima, partial derivatives of weights and biases are slopes of the gradient’s tangent. If the slope i.e., the value of a partial derivative is negative then the gradient moves to the right, and similarly if the gradient is to the left side the slope is positive hence the value of weight increases, and moves to the right side. Now if the value of alpha is too big, the gradient takes large steps and hence fails to converge and reach the optimal solution as can be seen in the third graph of the image. Hence, alpha, or the learning parameter is an important hyperparameter that determines the accuracy of the multi-layer perceptron.

For the instance of MLP used in this study we have taken alpha as 0.01 and epoch as 35. Epoch is a hyperparameter that determines the number of iterations the algorithm will go through the training data.

Diagram

Description automatically generated

Image reference: https://www.jeremyjordan.me/nn-learning-rate/

* Training for the model developed for perceptron and multi-layer perceptron

@Anushka Anil Padwal

The dataset for wildfires was initially divided into two sets training and testing. With 2/3 data for training and 1/3 for testing. Later the training data was divided into 5 bins. For 5 iterations, one bin is assigned for testing and the rest for training. During each iteration a model is trained, after the five iterations, we have 5 models. The final hypotheses are calculated using the mean of these 5 models’ hypotheses.

This process is similar for both perceptron and multi-layer perceptron developed from scratch and references from sklearn libraries.

1. Cross-validation training results for Perceptron 2. Cross-validation training results for MLP

Table

Description automatically generated with medium confidence Text

Description automatically generated with low confidence

1. Cross-Validation training for sklearn Perceptron 2. Cross-Validation for sklearn MLP

Text

Description automatically generated with medium confidence Table

Description automatically generated

* Testing results for modelled trained from scratch and sklearn libraries.

@Smitesh Nitin Patil

For testing on the test set we have taken the average of all the 5 models trained using cross-validation. The output is between the range 0 and 1 as a probability. It is rounded off to discrete values 0 and 1. The probabilities are kept for plotting the roc curve later.

1. Testing results for Perceptron from scratch: 2. Testing results for MLP from scratch

Table

Description automatically generated Table

Description automatically generated

1. Testing results for sklearn Perceptron 2. Testing results for sklearn MLP

Table

Description automatically generated Table

Description automatically generated

One key difference between the models trained by us and the sklearn models is that the sklearn models output the predictions as discrete values 1 and 0 and not a probability. Hence, while comparing the algorithms we cannot use the ROC curve for sklearn models as there are no cut-off values.

* Conclusion

@Smitesh Nitin Patil

It can be observed that models from sklearn generally performed better than the algorithms we developed from scratch. The Structure of both the algorithms Perceptron and Sklearn Perceptron, Multi-layer Perceptron and, Sklearn Perceptron is the same as well. It can be inferred that because machine learning libraries are finely tuned based on many years of development, they are more robust and thus produce better results.

For Perceptron :

It can be observed that perceptron developed from scratch wrongly classifies 6 data samples as False positives whereas for sklearn perceptron only 1 gets classified wrongly. Hence Perceptron with sklearn is more robust in detecting false positives, but as a trade-off number of false negatives are higher for the sklearn perceptron. . For Machine Learning tasks like predicting if a tumour is malignant or benign from sample images, it is better to use an algorithm that classifies less false negatives as we need to get as many true positives correctly classified as possible.

@Anushka Anil Padwal

For Multi-layer Perceptron:

Multi-layer perceptron from scratch performs better than perceptron from scratch but still, sklearn MLPClassifier works better in this case as well. Scratch MLP improves on Scratch Perceptron as it classifies only two sample as a false positive but in terms of False negatives it performs worse than sklearn MLP.

Chart

Description automatically generated

From the roc curve, it can be interpreted that scratch mlp works better than scratch perceptron as it curves closer to the top left corner. Also, the Area under the curve is better for scratch MLP.

Code Appendix

import numpy as np  
import pandas as pd  
import seaborn as sns  
import matplotlib.pyplot as plt  
from sklearn.metrics import roc\_curve, auc  
from sklearn.preprocessing import StandardScaler

data = pd.read\_table("wildfires.txt", delim\_whitespace=True)  
data['fire'] = data['fire'].map({"yes":1, "no":0})

corr = data.corr()  
plt.figure(figsize = (16,5))  
sns.heatmap(corr, cmap="Blues",annot = True)  
plt.savefig("Heatmap")

A screenshot of a computer

Description automatically generated with medium confidence

# Perceptron From Scratch - @Anushka Anil Padwal

class Perceptron:  
   
 #activation function for the perceptron  
 def sigmoid(self, x):  
 return 1/(1+np.exp(-x))  
  
 #constructor call  
 def \_\_init\_\_(self):  
 self.learning\_rate = None  
 self.epochs = None  
 self.weights = None  
 self.bias = None  
  
 #fit function to adjust weights and bias for a certain number of epochs  
 def fit(self, X\_train, Y\_train, epochs = 100, learning\_rate= 0.00001):  
 data\_size, vector\_size = X\_train.shape  
 self.epochs = epochs  
 self.learning\_rate = learning\_rate  
 self.weight = np.random.rand(vector\_size)  
 self.bias = 0  
 for i in range(self.epochs):  
 for x, y in zip(X\_train, Y\_train):  
 Y\_pred = self.sigmoid(np.dot(x, self.weight) + self.bias)  
 self.weight += self.learning\_rate \* (y - Y\_pred) \* x  
 self.bias += self.learning\_rate \* (y - Y\_pred)  
  
 def predict(self, X):  
 return self.sigmoid(np.dot(X, self.weight) + self.bias)

#splitiing datta 1/3 for test 2/3 for training and crossvalidation  
from sklearn.model\_selection import train\_test\_split  
train\_data, test\_data = train\_test\_split(data, test\_size=0.33, random\_state = 100)

#scaling data such that mean is zero and standard deviation as one  
scaler = StandardScaler()  
X\_train = scaler.fit\_transform(train\_data)  
X\_test = scaler.transform(test\_data)

#splitting x and y data  
X\_train = train\_data[["temp","drought\_code","buildup\_index"]]  
Y\_train = train\_data[['fire']]  
X\_test = test\_data[["temp","drought\_code","buildup\_index"]]  
Y\_test = test\_data[['fire']]

#pandas to numpy for mathematical operations  
X\_train = X\_train.to\_numpy().reshape(X\_train.shape[0], 3)  
Y\_train = Y\_train.to\_numpy().reshape(Y\_train.shape[0],)   
X\_test = X\_test.to\_numpy().reshape(X\_test.shape[0], 3)  
Y\_test = Y\_test.to\_numpy().reshape(Y\_test.shape[0],)

from sklearn.model\_selection import KFold  
from sklearn.metrics import accuracy\_score  
  
#function outputs 5 models trained using crossvalidation  
def crossvalidation(X\_train, Y\_train, splits):  
 cv = KFold(n\_splits=splits)  
 models = []  
 for train, test in cv.split(X\_train):  
 roundoff = lambda x: np.round(x)  
 x\_train = X\_train[train]  
 y\_train = Y\_train[train]  
 x\_val = X\_train[test]  
 y\_val = Y\_train[test]  
  
 p = Perceptron()  
 p.fit(x\_train, y\_train, 100, 0.00001)  
 y\_pred = roundoff(p.predict(x\_val))  
 print("model accuracy : ",accuracy\_score(y\_val, y\_pred))  
 models.append(p)  
 return models

models = crossvalidation(X\_train, Y\_train, 5)

model accuracy : 0.8928571428571429  
model accuracy : 0.7407407407407407  
model accuracy : 0.7037037037037037  
model accuracy : 0.8888888888888888  
model accuracy : 0.6296296296296297

#this fucntion outputs probabilites and discrete values of the preditions made on Y\_test as a list  
def predictions\_cross\_val(models, X\_test):  
 probablities = np.empty(0, )  
 discrete\_vals = np.empty(0, )  
 for test in X\_test:  
 roundoff = lambda x: np.round(x)  
 pred = np.average([models[0].predict(test),   
 models[1].predict(test),  
 models[2].predict(test),  
 models[3].predict(test),  
 models[4].predict(test)])  
 probablities = np.append(probablities, pred)  
 discrete\_vals = np.append(discrete\_vals, roundoff(pred))  
   
 return probablities, discrete\_vals

Y\_pred\_probabilities, Y\_pred = predictions\_cross\_val(models, X\_test)

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report  
accuracy\_score(Y\_pred, Y\_test)

0.8382352941176471

confusion\_matrix(Y\_test, Y\_pred)

array([[26, 5],  
 [ 6, 31]], dtype=int64)

print(classification\_report(Y\_test, Y\_pred))

precision recall f1-score support  
  
 0 0.81 0.84 0.83 31  
 1 0.86 0.84 0.85 37  
  
 accuracy 0.84 68  
 macro avg 0.84 0.84 0.84 68  
weighted avg 0.84 0.84 0.84 68

#saving roc data for plotting later  
from sklearn.metrics import roc\_curve, auc  
  
fpr\_perceptron, tpr\_preceptron, threshold = roc\_curve(Y\_test, Y\_pred\_probabilities)  
auc\_perceptron = auc(fpr\_perceptron, tpr\_preceptron)

# MLP Classifier from scratch @Smitesh Nitin Patil

class Layer:  
 def \_\_init\_\_(self):  
 self.input = None  
 self.output = None  
  
 # computes the output Y of a layer for a given input X  
 def forward\_propagation(self, input):  
 pass  
  
 # computes partial derivatives for adjusting weights  
 def backward\_propagation(self, output\_error, learning\_rate):  
 pass

class WeightedLayer(Layer):  
 # input no of layers output no of outputs  
 def \_\_init\_\_(self, input\_size, output\_size):  
 self.weights = np.random.rand(input\_size, output\_size) -0.5  
 self.bias = np.random.rand(1, output\_size) -0.5  
  
 # returns wx+b  
 def forward\_propagation(self, input\_data):  
 self.input = input\_data  
 self.output = np.dot(self.input, self.weights) + self.bias  
 return self.output  
   
 # computes partial derivatives and updates weights {weight = weight - alpha\*partialderivative}  
 def backward\_propagation(self, output\_error, learning\_rate):  
 input\_error = np.dot(output\_error, self.weights.T)  
 weights\_error = np.dot(self.input.T, output\_error)  
   
 self.weights -= learning\_rate \* weights\_error  
 self.bias -= learning\_rate \* output\_error  
 return input\_error

class Sigmoid(Layer):  
   
 #activation function  
 def sigmoid(self, x):  
 return 1/(1+np.exp(-x))  
   
 #activation derivative  
 def sigmoid\_derivative(self, x):  
 return self.sigmoid(x)\*(1-self.sigmoid(x))  
   
 def \_\_init\_\_(self):  
 pass  
   
 # applying sigmoid to the FClayer output  
 def forward\_propagation(self, input\_data):  
 self.input = input\_data  
 self.output = self.sigmoid(self.input)  
 return self.output  
  
   
 # applying sigmoid derivative to the Activation layer  
 def backward\_propagation(self, output\_error, learning\_rate):  
 return self.sigmoid\_derivative(self.input) \* output\_error

class Network:  
   
 def \_\_init\_\_(self):  
 self.layers = []  
   
 #loss function   
 def mean\_squared\_error\_derivative(self, Y\_true, Y\_pred):  
 return 2\*(Y\_pred - Y\_true)/ Y\_true.size  
  
 # add layers to network  
 def add(self, layer):  
 self.layers.append(layer)  
  
 #predict output for a list of inputdata  
 def predict(self, input\_data):  
 result = np.empty(0, )  
  
 #calculate results for all the data  
 for i in range(len(input\_data)):  
 # forward propagation  
 output = input\_data[i]  
 for layer in self.layers:  
 output = layer.forward\_propagation(output)  
 result = np.append(result, output)  
  
 return result  
  
 # train the network  
 def fit(self, x\_train, y\_train, epochs, learning\_rate):  
  
 #training for a number of epochs  
 for i in range(epochs):  
 for j in range(len(x\_train)):  
 # forward propagation  
 output = x\_train[j]  
 for layer in self.layers:  
 output = layer.forward\_propagation(output)  
  
 # backward propagation  
 error = self.mean\_squared\_error\_derivative(y\_train[j], output)  
 for layer in reversed(self.layers):  
 error = layer.backward\_propagation(error, learning\_rate)

from sklearn.model\_selection import train\_test\_split  
train\_data, test\_data = train\_test\_split(data, test\_size=0.33, random\_state = 100)

X\_train = train\_data[["temp","drought\_code","buildup\_index"]]  
Y\_train = train\_data[['fire']]  
X\_test = test\_data[["temp","drought\_code","buildup\_index"]]  
Y\_test = test\_data[['fire']]

X\_train = X\_train.to\_numpy().reshape(X\_train.shape[0], 1, 3)  
Y\_train = Y\_train.to\_numpy().reshape(Y\_train.shape[0],)  
X\_test = X\_test.to\_numpy().reshape(X\_test.shape[0], 1, 3)  
Y\_test = Y\_test.to\_numpy().reshape(Y\_test.shape[0],)

from sklearn.model\_selection import KFold  
from sklearn.metrics import accuracy\_score  
  
def cross\_validation(X\_train, Y\_train, splits):  
 roundoff = lambda x: np.round(x)  
 cv = KFold(n\_splits=splits)  
 models = []  
 for train, test in cv.split(X\_train):  
 x\_train = X\_train[train]  
 y\_train = Y\_train[train]  
 x\_val = X\_train[test]  
 y\_val = Y\_train[test]  
  
 net = Network()  
 net.add(WeightedLayer(3, 25))   
 net.add(Sigmoid())  
 net.add(WeightedLayer(25, 1))   
 net.add(Sigmoid())  
 net.fit(x\_train, y\_train, epochs = 35, learning\_rate = 0.001)  
 y\_pred = roundoff(net.predict(x\_val))  
 print(accuracy\_score(y\_val, y\_pred))  
 models.append(net)  
 return models

models = cross\_validation(X\_train, Y\_train, 5)

0.9285714285714286  
0.8148148148148148  
0.8148148148148148  
0.8518518518518519  
0.7777777777777778

Y\_pred\_probabilities, Y\_pred = predictions\_cross\_val(models, X\_test)

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report  
  
accuracy\_score(Y\_test, Y\_pred)

0.8529411764705882

confusion\_matrix(Y\_test, Y\_pred)

array([[23, 8],  
 [ 2, 35]], dtype=int64)

print(classification\_report(Y\_test, Y\_pred))

precision recall f1-score support  
  
 0 0.92 0.74 0.82 31  
 1 0.81 0.95 0.88 37  
  
 accuracy 0.85 68  
 macro avg 0.87 0.84 0.85 68  
weighted avg 0.86 0.85 0.85 68

from sklearn.metrics import roc\_curve, auc  
  
fpr\_mlp, tpr\_mlp, threshold = roc\_curve(Y\_test, Y\_pred\_probabilities)  
auc\_mlp = auc(fpr\_mlp, tpr\_mlp)

# Perceptron Sklearn for Reference @ Anushka Anil Padwal

from sklearn.model\_selection import train\_test\_split  
train\_data, test\_data = train\_test\_split(data, test\_size=0.33, random\_state = 100)

#scaling data such that mean is zero and standard deviation as one  
scaler = StandardScaler()  
X\_train = scaler.fit\_transform(train\_data)  
X\_test = scaler.transform(test\_data)

#splitting data in X and y  
X\_train = train\_data[["temp","drought\_code","buildup\_index"]]  
Y\_train = train\_data[['fire']]  
X\_test = test\_data[["temp","drought\_code","buildup\_index"]]  
Y\_test = test\_data[['fire']]

# converting to numpy for mathematical operations  
X\_train = X\_train.to\_numpy().reshape(X\_train.shape[0], 3)  
Y\_train = Y\_train.to\_numpy().reshape(Y\_train.shape[0],)  
X\_test = X\_test.to\_numpy().reshape(X\_test.shape[0], 3)  
Y\_test = Y\_test.to\_numpy().reshape(Y\_test.shape[0],)

#function outputs 5 models trained using crossvalidation  
from sklearn.model\_selection import KFold  
from sklearn.linear\_model import Perceptron  
from sklearn.metrics import accuracy\_score  
  
def cross\_validation(X\_train, Y\_train, splits):  
 roundoff = lambda x: np.round(x)  
 cv = KFold(n\_splits=splits)  
 models = []  
 for train, test in cv.split(X\_train):  
 x\_train = X\_train[train]  
 y\_train = Y\_train[train]  
 x\_val = X\_train[test]  
 y\_val = Y\_train[test]  
  
 p = Perceptron(alpha = 0.0001)  
 p.fit(X\_train, Y\_train)  
 y\_pred = roundoff(p.predict(x\_val))  
 print(accuracy\_score(y\_val, y\_pred))  
 models.append(p)  
 return models

models = cross\_validation(X\_train, Y\_train, 5)

0.9285714285714286  
0.8518518518518519  
0.8148148148148148  
0.8888888888888888  
0.6666666666666666

#this function outputs probabilites and discrete values of the preditions made on Y\_test as a list  
  
def predictions\_cross\_val(models, X\_test):  
 probablities = np.empty(0, )  
 discrete\_vals = np.empty(0, )  
 for test in X\_test:  
 roundoff = lambda x: np.round(x)  
 pred = np.average([models[0].predict(test.reshape(1, -1)),   
 models[1].predict(test.reshape(1, -1)),  
 models[2].predict(test.reshape(1, -1)),  
 models[3].predict(test.reshape(1, -1)),  
 models[4].predict(test.reshape(1, -1))])  
 probablities = np.append(probablities, pred)  
 discrete\_vals = np.append(discrete\_vals, roundoff(pred))  
   
 return probablities, discrete\_vals

Y\_pred\_probabilities, Y\_pred = predictions\_cross\_val(models, X\_test)

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report  
# performance metrics  
accuracy\_score(Y\_test, Y\_pred)

0.8823529411764706

confusion\_matrix(Y\_test, Y\_pred)

array([[24, 7],  
 [ 1, 36]], dtype=int64)

print(classification\_report(Y\_test, Y\_pred))

precision recall f1-score support  
  
 0 0.96 0.77 0.86 31  
 1 0.84 0.97 0.90 37  
  
 accuracy 0.88 68  
 macro avg 0.90 0.87 0.88 68  
weighted avg 0.89 0.88 0.88 68

# MLP Sklearn for reference @Smitesh Nitin Patil

from sklearn.model\_selection import train\_test\_split  
train\_data, test\_data = train\_test\_split(data, test\_size=0.33, random\_state = 100)

#scaling data such that mean is zero and standard deviation as one  
scaler = StandardScaler()  
X\_train = scaler.fit\_transform(train\_data)  
X\_test = scaler.transform(test\_data)

#splitting data in X and y  
X\_train = train\_data[["temp","drought\_code","buildup\_index"]]  
Y\_train = train\_data[['fire']]  
X\_test = test\_data[["temp","drought\_code","buildup\_index"]]  
Y\_test = test\_data[['fire']]

# converting to numpy for mathematical operations  
X\_train = X\_train.to\_numpy().reshape(X\_train.shape[0], 3)  
Y\_train = Y\_train.to\_numpy().reshape(Y\_train.shape[0],)  
X\_test = X\_test.to\_numpy().reshape(X\_test.shape[0], 3)  
Y\_test = Y\_test.to\_numpy().reshape(Y\_test.shape[0],)

#function outputs 5 models trained using crossvalidation  
from sklearn.model\_selection import KFold  
from sklearn.neural\_network import MLPClassifier  
from sklearn.metrics import accuracy\_score  
  
def cross\_validation(X\_train, Y\_train, splits):  
 roundoff = lambda x: np.round(x)  
 cv = KFold(n\_splits=splits)  
 models = []  
 for train, test in cv.split(X\_train):  
 x\_train = X\_train[train]  
 y\_train = Y\_train[train]  
 x\_val = X\_train[test]  
 y\_val = Y\_train[test]  
  
 mlp= MLPClassifier(solver = 'sgd', hidden\_layer\_sizes = (25, ) ,alpha= 0.00001)  
 mlp.fit(X\_train, Y\_train)  
 y\_pred = roundoff(mlp.predict(x\_val))  
 print(accuracy\_score(y\_val, y\_pred))  
 models.append(mlp)  
 return models

models = cross\_validation(X\_train, Y\_train, 5)

0.9285714285714286  
0.8518518518518519  
0.8888888888888888  
0.8888888888888888  
0.7777777777777778

Y\_pred\_probabilities, Y\_pred = predictions\_cross\_val(models, X\_test)

from sklearn.metrics import accuracy\_score, confusion\_matrix, classification\_report  
accuracy\_score(Y\_test, Y\_pred)

0.8970588235294118

confusion\_matrix(Y\_test, Y\_pred)

array([[26, 5],  
 [ 2, 35]], dtype=int64)

print(classification\_report(Y\_test, Y\_pred))

precision recall f1-score support  
  
 0 0.93 0.84 0.88 31  
 1 0.88 0.95 0.91 37  
  
 accuracy 0.90 68  
 macro avg 0.90 0.89 0.90 68  
weighted avg 0.90 0.90 0.90 68

plt.plot(fpr\_perceptron, tpr\_preceptron, linestyle='-', label=' scratch percepton auc = %0.3f' % auc\_perceptron)  
plt.plot(fpr\_mlp, tpr\_mlp, linestyle='-', label=' scratch mlp auc = %0.3f)' % auc\_mlp)  
  
  
plt.xlabel('False Positive Rate -->')  
plt.ylabel('True Positive Rate -->')  
   
plt.legend()  
plt.savefig("Roc curve")  
plt.show()

A picture containing graphical user interface

Description automatically generated