LAB REPORT 3

November 10, 2020

Submitted by:

Kazi Amit Hasan
Roll: 1503089
Department of Computer Science & Engineering
Rajshahi University of Engineering & Technology

November 10, 2020

0.1 Title

Implementation of **Multilayer Perceptron Learning** algorithm.

0.2 Objectives

1. Learning the concept of multi layer perceptron algorithm.
2. Learning the maths behind the algorithm
3. Learning and implementing in a real dataset

0.3 Methodology

A multilayer perceptron (MLP) is a class of feedforward artificial neural network (ANN. Artificial neural networks (ANNs) or connectionist systems are computing systems inspired by the biological neural networks that constitute animal brains. MLP utilizes a supervised learning technique called backpropagation for training. Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable. The Multilayer Perceptron Networks are characterized by the presence of many intermediate layers (hidden) in your structure, located between input layer and output layer. With this, such networks have the advantage of being able to classify more than two different classes and It also solve non-linearly separable problems.

We can summarize the operation of the perceptron as follows it:

- Step 1: Initialize the weights and bias with small-randomized values;
- Step 2: Propagate all values in the input layer until output layer(Forward Propagation)
- Step 3: Update weight and bias in the inner layers(Backpropagation)
- Step 4: Do it until that the stop criterion is satisfied !

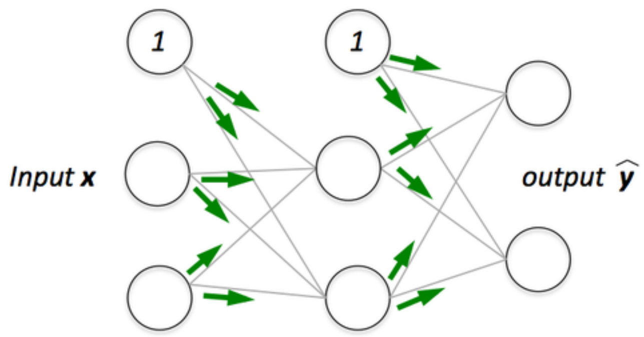

Figure 1: Forward propagation Algorithm

0.4 Dataset description

The Iris Flower Dataset, also called Fisher's Iris, is a dataset introduced by Ronald Fisher, a British statistician, and biologist, with several contributions to science. Ronald Fisher has well known worldwide for his paper The use of multiple measurements in taxonomic problems as an example of linear discriminant analysis. It was in this paper that Ronald Fisher introduced the Iris flower dataset.

The iris database consists of 50 samples distributed among three different species of iris. Each of these samples has specific characteristics, which allows them to be classified into three categories: Iris Setosa, Iris Virginica, and Iris versicolor. In this tutorial, we will use multilayer perceptron to separate and classify the iris samples.

0.5 Implementation

0.5.1 Forward propagation Algorithm

In order to proceed we need to improve the notation we have been using. That for, for each layer $1 \geq l \geq L$, the activations and outputs are calculated as:

$$L^l_j = \sum_i w^l_{ji} x^l_i = w^l_{j,0} x^l_0 + w^l_{j,1} x^l_1 + w^l_{j,2} x^l_2 + ... + w^l_{j,n} x^l_n,$$
$$Y^l_j = g^l(L^l_j),$$
$$\{y_i, x_{i1}, \ldots, x_{ip}\}^n_{i=1}$$

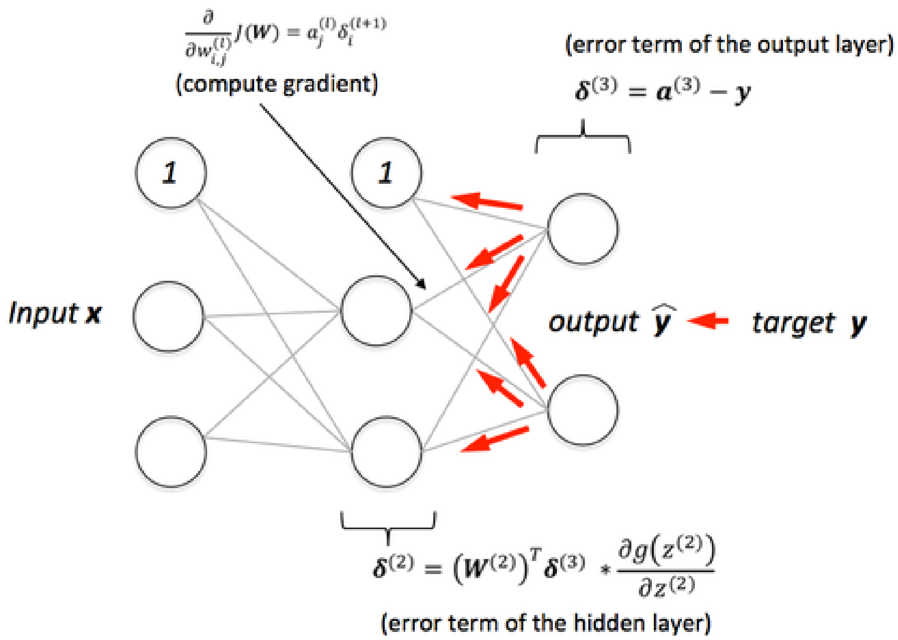

Figure 2: Backpropagation Algorithm

0.5.2 Calculation our Erro function

It is used to measure performance locality associated with the results produced by the neurons in output layer and the expected result. $E(k) = \frac{1}{2}\sum_{k=1}^{K}(d_j(k) - y_j(k))^2$.

0.5.3 Activation Functions

There are different types of activation functions like sigmoid, RELU etc.

0.5.4 Backpropagation Algorithm

In Output and Input Layer, the following steps are performed:

- Calculate error in output layer:
- Update all weight between hidden and output layer
- Update bias value in output layer.
- Calculate error in hidden layer

- Update all weight between hidden and output layer
- Update bias value in output layer

0.6 Code

0.6.1 Implementation the Multilayer Perceptron in Python

```
from sklearn.base import BaseEstimator, ClassifierMixin, RegressorMixin
import random

class MultiLayerPerceptron(BaseEstimator, ClassifierMixin):
    def __init__(self, params=None):
        if (params == None):
            self.inputLayer = 4                          # Input Layer
            self.hiddenLayer = 5                         # Hidden Layer
            self.outputLayer = 3                         # Outpuy Layer
            self.learningRate = 0.005                    # Learning rate
            self.max_epochs = 600                        # Epochs
            self.iasHiddenValue = -1                     # Bias HiddenLayer
            self.BiasOutputValue = -1                    # Bias OutputLayer
            self.activation = self.ativacao['sigmoid'] # Activation function
            self.deriv = self.derivada['sigmoid']
        else:
            self.inputLayer = params['InputLayer']
            self.hiddenLayer = params['HiddenLayer']
            self.OutputLayer = params['OutputLayer']
            self.learningRate = params['LearningRate']
            self.max_epochs = params['Epocas']
            self.BiasHiddenValue = params['BiasHiddenValue']
            self.BiasOutputValue = params['BiasOutputValue']
            self.activation = self.ativacao[params['ActivationFunction']]
            self.deriv = self.derivada[params['ActivationFunction']]
```

```
    'Starting Bias and Weights'
    self.WEIGHT_hidden = self.starting_weights
    (self.hiddenLayer, self.inputLayer)
    self.WEIGHT_output = self.starting_weights
    (self.OutputLayer, self.hiddenLayer)
    self.BIAS_hidden = np.array([self.BiasHiddenValue
    for i in range(self.hiddenLayer)])
    self.BIAS_output = np.array([self.BiasOutputValue
    for i in range(self.OutputLayer)])
    self.classes_number = 3

pass

def starting_weights(self, x, y):
    return [[2  * random.random() - 1 for i in range(x)] for j in range(y)]

ativacao = {
     'sigmoid': (lambda x: 1/(1 + np.exp(-x))),
        'tanh': (lambda x: np.tanh(x)),
        'Relu': (lambda x: x*(x > 0)),
           }
derivada = {
     'sigmoid': (lambda x: x*(1-x)),
        'tanh': (lambda x: 1-x**2),
        'Relu': (lambda x: 1 * (x>0))
           }

def Backpropagation_Algorithm(self, x):
    DELTA_output = []
    'Stage 1 - Error: OutputLayer'
    ERROR_output = self.output - self.OUTPUT_L2
    DELTA_output = ((-1)*(ERROR_output) * self.deriv(self.OUTPUT_L2))

    arrayStore = []
    'Stage 2 - Update weights OutputLayer and HiddenLayer'
```

```
        for i in range(self.hiddenLayer):
            for j in range(self.OutputLayer):
                self.WEIGHT_output[i][j] -= (self.learningRate *
                (DELTA_output[j] * self.OUTPUT_L1[i]))
                self.BIAS_output[j] -= (self.learningRate * DELTA_output[j])

        'Stage 3 - Error: HiddenLayer'
        delta_hidden = np.matmul(self.WEIGHT_output, DELTA_output)*
        self.deriv(self.OUTPUT_L1)

        'Stage 4 - Update weights HiddenLayer and InputLayer(x)'
        for i in range(self.OutputLayer):
            for j in range(self.hiddenLayer):
                self.WEIGHT_hidden[i][j] -= (self.learningRate *
                (delta_hidden[j] *
                x[i]))
                self.BIAS_hidden[j] -= (self.learningRate * delta_hidden[j])

    def show_err_graphic(self,v_erro,v_epoca):
        plt.figure(figsize=(9,4))
        plt.plot(v_epoca, v_erro, "m-",color="b", marker=11)
        plt.xlabel("Number of Epochs")
        plt.ylabel("Squared error (MSE) ");
        plt.title("Error Minimization")
        plt.show()

    def predict(self, X, y):
        'Returns the predictions for every element of X'
        my_predictions = []
        'Forward Propagation'
        forward = np.matmul(X,self.WEIGHT_hidden) + self.BIAS_hidden
        forward = np.matmul(forward, self.WEIGHT_output) + self.BIAS_output

        for i in forward:
            my_predictions.append(max(enumerate(i), key=lambda x:x[1])[0])
```

```
    print(" Number of Sample  | Class |  Output |  Hoped Output  ")
    for i in range(len(my_predictions)):
        if(my_predictions[i] == 0):
            print("id:{}    |
            Iris-Setosa  |  Output: {}  ".format(i, my_predictions[i],
            y[i]))
        elif(my_predictions[i] == 1):
            print("id:{}    |
            Iris-Versicolour    |  Output: {}  ".format(i, my_predictions[i],
            y[i]))
        elif(my_predictions[i] == 2):
            print("id:{}    |
            Iris-Iris-Virginica   |  Output: {}  ".format(i, my_predictions[i],
            y[i]))

    return my_predictions
    pass

def fit(self, X, y):
    count_epoch = 1
    total_error = 0
    n = len(X);
    epoch_array = []
    error_array = []
    W0 = []
    W1 = []
    while(count_epoch <= self.max_epochs):
        for idx,inputs in enumerate(X):
            self.output = np.zeros(self.classes_number)
            'Stage 1 - (Forward Propagation)'
            self.OUTPUT_L1 = self.activation
            ((np.dot(inputs, self.WEIGHT_hidden) +

            self.BIAS_hidden.T))
```

```
        self.OUTPUT_L2 = self.activation
        ((np.dot(self.OUTPUT_L1, self.WEIGHT_output) + self.BIAS_output.T))
        'Stage 2 - One-Hot-Encoding'
        if(y[idx] == 0):
            self.output = np.array([1,0,0]) #Class1 {1,0,0}
        elif(y[idx] == 1):
            self.output = np.array([0,1,0]) #Class2 {0,1,0}
        elif(y[idx] == 2):
            self.output = np.array([0,0,1]) #Class3 {0,0,1}

        square_error = 0
        for i in range(self.OutputLayer):
            erro = (self.output[i] - self.OUTPUT_L2[i])**2
            square_error = (square_error + (0.05 * erro))
            total_error = total_error + square_error

        'Backpropagation : Update Weights'
        self.Backpropagation_Algorithm(inputs)

    total_error = (total_error / n)
    if((count_epoch % 50 == 0)or(count_epoch == 1)):
        print("Epoch ", count_epoch, "- Total Error: ",total_error)
        error_array.append(total_error)
        epoch_array.append(count_epoch)

    W0.append(self.WEIGHT_hidden)
    W1.append(self.WEIGHT_output)


    count_epoch += 1
self.show_err_graphic(error_array,epoch_array)

plt.plot(W0[0])
plt.title('Weight Hidden update during training')
plt.legend(['neuron1', 'neuron2', 'neuron3', 'neuron4', 'neuron5'])
```

```
    plt.ylabel('Value Weight')
    plt.show()

    plt.plot(W1[0])
    plt.title('Weight Output update during training')
    plt.legend(['neuron1', 'neuron2', 'neuron3'])
    plt.ylabel('Value Weight')
    plt.show()

    return self
```

0.6.2 Finding the best parameters

For find the best parameters, it was necessary to realize various tests using different values to the parameters. The graphs below denote all tests made to select the best configuration for the multilayer perceptron. These tests were important in selecting the best settings and ensuring the best accuracy. The graph was drawn manually, but you can change the settings and note the results obtained. The tests involve different activation functions and the number of neurons for each layer.

```
def show_test():
    ep1 = [0,100,200,300,400,500,600,700,800,900,1000,1500,2000]
    h_5 = [0,60,70,70,83.3,93.3,96.7,86.7,86.7,76.7,73.3,66.7,66.7]
    h_4 = [0,40,70,63.3,66.7,70,70,70,70,66.7,66.7,43.3,33.3]
    h_3 = [0,46.7,76.7,80,76.7,76.7,76.6,73.3,73.3,73.3,73.3,76.7,76.7]
    plt.figure(figsize=(10,4))
    l1, = plt.plot(ep1, h_3, "m-",color='b',label="node-3", marker=11)
    l2, = plt.plot(ep1, h_4, "m-",color='g',label="node-4", marker=8)
    l3, = plt.plot(ep1, h_5, "m-",color='r',label="node-5", marker=5)
    plt.legend(handles=[l1,l2,l3], loc=1)
    plt.xlabel("number of Epochs");plt.ylabel("% Hits");
    plt.title("Number of Hidden Layers - Performance")

    ep2 = [0,100,200,300,400,500,600,700]
    tanh = [0.18,0.027,0.025,0.022,0.0068,0.0060,0.0057,0.00561]
```

```
sigm = [0.185,0.0897,0.060,0.0396,0.0343,0.0314,0.0296,0.0281]
Relu = [0.185,0.05141,0.05130,0.05127,0.05124,0.05123,0.05122,0.05121]
plt.figure(figsize=(10,4))
l1 , = plt.plot(ep2, tanh, "m-",color='b',label="Hyperbolic Tangent",marker=11)
l2 , = plt.plot(ep2, sigm, "m-",color='g',label="Sigmoide", marker=8)
l3 , = plt.plot(ep2, Relu, "m-",color='r',label="ReLu", marker=5)
plt.legend(handles=[l1,l2,l3], loc=1)
plt.xlabel("Epoch");plt.ylabel("Error");
plt.title("Activation Functions - Performance")

fig, ax = plt.subplots()
names = ["Hyperbolic Tangent","Sigmoide","ReLU"]
x1 = [2.0,4.0,6.0]
plt.bar(x1[0],53.4,0.4,color='b')
plt.bar(x1[1],96.7,0.4,color='g')
plt.bar(x1[2],33.2,0.4,color='r')
plt.xticks(x1,names)
plt.ylabel('% Hits')
plt.title('Hits - Activation Functions')
plt.show()
```

0.6.3 Training our MultiLayer Perceptron

```
dictionary = {'InputLayer':4, 'HiddenLayer':5, 'OutputLayer':3,
              'Epocas':700, 'LearningRate':0.005,'BiasHiddenValue':-1,
              'BiasOutputValue':-1, 'ActivationFunction':'sigmoid'}

Perceptron = MultiLayerPerceptron(dictionary)
Perceptron.fit(train_X,train_y)
```

0.6.4 Testing the result

```
prev = Perceptron.predict(test_X,test_y)
hits = n_set = n_vers = n_virg = 0
score_set = score_vers = score_virg = 0
for j in range(len(test_y)):
```

```
    if(test_y[j] == 0): n_set += 1
    elif(test_y[j] == 1): n_vers += 1
    elif(test_y[j] == 2): n_virg += 1

for i in range(len(test_y)):
    if test_y[i] == prev[i]:
        hits += 1
    if test_y[i] == prev[i] and test_y[i] == 0:
        score_set += 1
    elif test_y[i] == prev[i] and test_y[i] == 1:
        score_vers += 1
    elif test_y[i] == prev[i] and test_y[i] == 2:
        score_virg += 1

hits = (hits / len(test_y))*100
faults = 100 - hits
```

0.6.5 Accuracy and precision the Multilayer Perceptron

```
graph_hits = []
print("Porcents :","%.2f"%(hits),"% hits","and","%.2f"%(faults),"% faults")
print("Total samples of test",n_samples)
print("*Iris-Setosa:",n_set,"samples")
print("*Iris-Versicolour:",n_vers,"samples")
print("*Iris-Virginica:",n_virg,"samples")

graph_hits.append(hits)
graph_hits.append(faults)
labels = 'Hits', 'Faults';
sizes = [96.5, 3.3]
explode = (0, 0.14)

fig1, ax1 = plt.subplots();
ax1.pie(graph_hits, explode=explode,colors=['blue','red'],
labels=labels, autopct='%1.1f%%',
```

```
shadow=True, startangle=90)
ax1.axis('equal')
plt.show()
```

0.6.6 Accuracy

```
acc_set = (score_set/n_set)*100
acc_vers = (score_vers/n_vers)*100
acc_virg = (score_virg/n_virg)*100
print("- Acurracy Iris-Setosa:","%.2f"%acc_set, "%")
print("- Acurracy Iris-Versicolour:","%.2f"%acc_vers, "%")
print("- Acurracy Iris-Virginica:","%.2f"%acc_virg, "%")
names = ["Setosa","Versicolour","Virginica"]
x1 = [2.0,4.0,6.0]
fig, ax = plt.subplots()
r1 = plt.bar(x1[0], acc_set,color='orange',label='Iris-Setosa')
r2 = plt.bar(x1[1], acc_vers,color='green',label='Iris-Versicolour')
r3 = plt.bar(x1[2], acc_virg,color='purple',label='Iris-Virginica')
plt.ylabel('Scores %')
plt.xticks(x1, names);plt.title('Scores by iris flowers - Multilayer Perceptron')
plt.show()
```

0.7 Results and Performance Analysis

The resulted parts of each coding part is given below: The Fig.3 represents the hidden layers vs num of epoch performance. In this lab, I used three activation functions. The performance of three activation functions are shown in Fig.4.

The Fig. 5 represents the total error with different epoch values. We can see that the error rate is reducing gradually with higher epochs.

The Fig. 6 represents the MSE error vs num of epochs grapgh. It is also showing that the errors are reducing with increasing number of epochs.

The Fig. 7 and Fig. 8 are representing the output performance of our MLP algorithm.

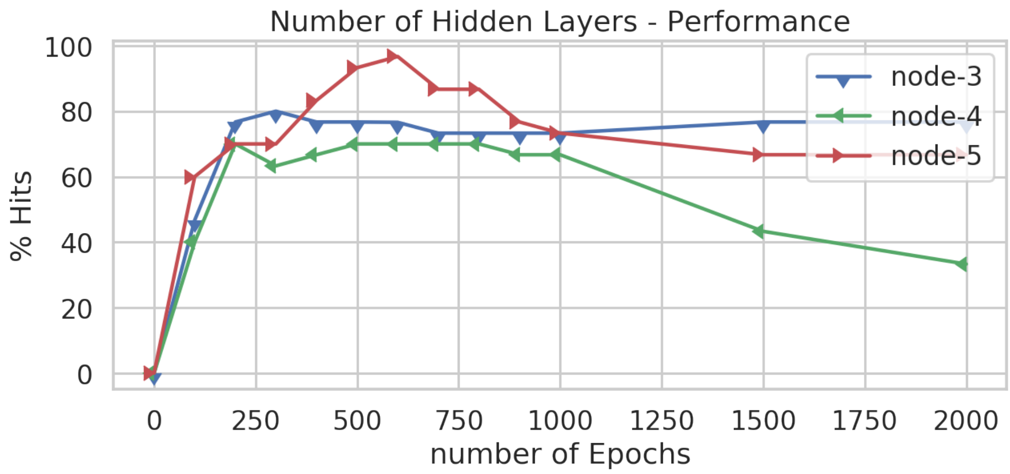

Figure 3: Performance of hidden layers

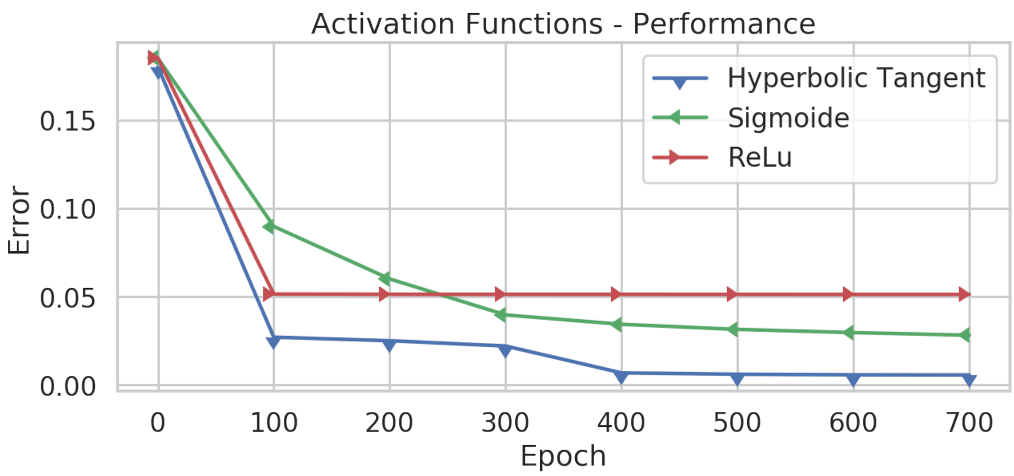

Figure 4: Performance of activation functions

```
Epoch  1 - Total Error:  0.08939914265311079
Epoch  50 - Total Error:  0.05376852115527734
Epoch  100 - Total Error:  0.03773105860576402
Epoch  150 - Total Error:  0.0311644504972821
Epoch  200 - Total Error:  0.028406909135682064
Epoch  250 - Total Error:  0.02669256194214157
Epoch  300 - Total Error:  0.02524003405939519
Epoch  350 - Total Error:  0.02384525367109564
Epoch  400 - Total Error:  0.022427464754471382
Epoch  450 - Total Error:  0.020948415886624147
Epoch  500 - Total Error:  0.01936513137210054
Epoch  550 - Total Error:  0.017597424493928912
Epoch  600 - Total Error:  0.01579217059564909
Epoch  650 - Total Error:  0.014194648016335552
Epoch  700 - Total Error:  0.012855583329293723
```

Figure 5: Loss measurements with epochs

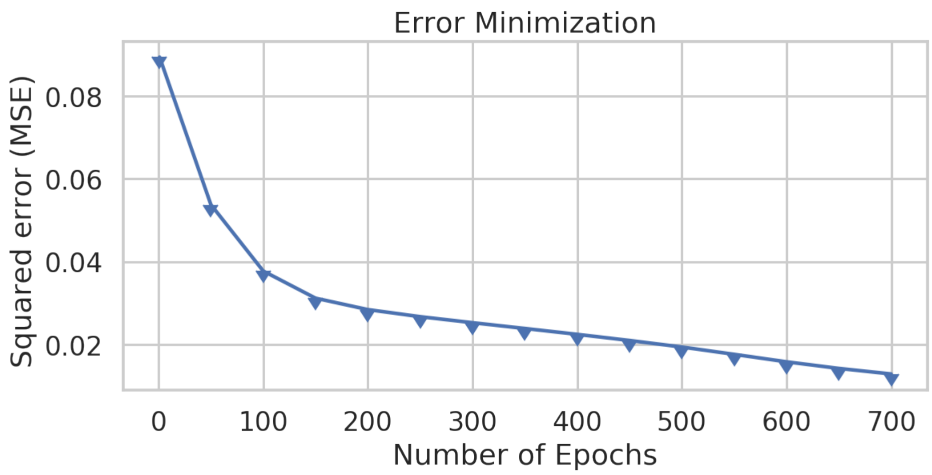

Figure 6: Error minimization

```
 Number of Sample   | Class |  Output |  Hoped Output
id:0     | Iris-Setosa   |  Output: 0
id:1     | Iris-Versicolour     |  Output: 1
id:2     | Iris-Versicolour     |  Output: 1
id:3     | Iris-Setosa   |  Output: 0
id:4     | Iris-Versicolour     |  Output: 1
id:5     | Iris-Versicolour     |  Output: 1
id:6     | Iris-Versicolour     |  Output: 1
id:7     | Iris-Versicolour     |  Output: 1
id:8     | Iris-Setosa   |  Output: 0
id:9     | Iris-Iris-Virginica    |  Output: 2
id:10     | Iris-Iris-Virginica    |  Output: 2
id:11     | Iris-Versicolour     |  Output: 1
id:12     | Iris-Versicolour     |  Output: 1
id:13     | Iris-Versicolour     |  Output: 1
id:14     | Iris-Setosa   |  Output: 0
id:15     | Iris-Iris-Virginica    |  Output: 2
id:16     | Iris-Iris-Virginica    |  Output: 2
id:17     | Iris-Versicolour     |  Output: 1
id:18     | Iris-Versicolour     |  Output: 1
id:19     | Iris-Versicolour     |  Output: 1
id:20     | Iris-Versicolour     |  Output: 1
id:21     | Iris-Iris-Virginica    |  Output: 2
id:22     | Iris-Versicolour     |  Output: 1
id:23     | Iris-Setosa   |  Output: 0
id:24     | Iris-Setosa   |  Output: 0
id:25     | Iris-Iris-Virginica    |  Output: 2
id:26     | Iris-Iris-Virginica    |  Output: 2
```

Figure 7: Output result

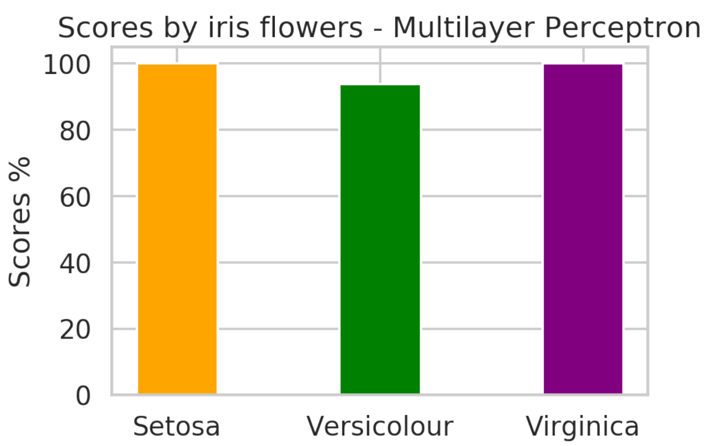

Figure 8: Accuracy per class

0.8 Conclusion

By implementing this, we knew about the fundamentals of one of the most basic machine learning algorithms. It has some basic advantages. In this lab, I tried to implement the multi layer perceptron learning algorithm without using any automated library functions All the codes and neccesary files are available in my github profile. The codes will be publicly available after my finals grades. My Github profile: https://github.com/AmitHasanShuvo/