Manraj Singh

IE 7860

Mar. 6, 2023

Dr. Ratna Chinnam

IE 7860 – Multi-Layer Perceptron Networks Report

Multi-layer perceptrons are a type of artificial neural network that can be used to solve classification and regression problems in machine learning. An MLP consists of one or more hidden layers of interconnected neurons, each of which applies an appropriate activation function to its inputs and passes to the next layer. The final output layer takes these inputs and produces a set of probabilities for each possible class and can be used to make predictions.

MLPs are beneficial for classification problems because they can solve problems containing non-linear decision boundaries, so they can separate classes of data that a linear classifier could not (SLP). This in turn can accommodate for more potential classes of data than only binary classification would allow. MLPs can learn complex patterns in input data which allows them to extract meaningful features and perform well with a vast array of input data types. MLPs also have robustness handling noise, able to handle noisy input data, filter out irrelevant information, and produce accurate results. In addition, MLPs can be scaled with more hidden layers and neurons to handle more large scale and complex data.

In this assignment, we look at three different datasets with 3 different classification problems in order to train and apply 3 unique MLP models to best accurately predict the output data from trained and tested input data.

Model 1 (HR Data)

This dataset utilized MLP to classify and predict how several different variables, such as salary, commute time, departments, evaluations, satisfaction levels, # projects, etc., would decide whether a worker would be likely to leave or not. In this case, with many different variables leading towards binary classification, it would be difficult to linearly separate the jumbled data, which would rule out the abilities of an SLP.

The data was loaded into Google Colab with appropriate imports, and explored:

```
[16] import numpy as np
     import pandas as pd
     import seaborn as sns
     from sklearn import preprocessing #Import Label Encoder

     #Bring in the train test and split function
     from sklearn.model_selection import train_test_split

     #Load data
     data=pd.read_csv('HR_comma_sep.csv')
     #Load first few rows
     data.head()
```

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years	department	salary
0	0.38	0.53	2	157	3	0	1	0	sales	low
1	0.80	0.86	5	262	6	0	1	0	sales	medium
2	0.11	0.88	7	272	4	0	1	0	sales	medium
3	0.72	0.87	5	223	5	0	1	0	sales	low
4	0.37	0.52	2	159	3	0	1	0	sales	low

As we can see, there are 9 input variables and 1 output variable (left). Describing of the data showed:

```
[17] data.describe()
```

	satisfaction_level	last_evaluation	number_project	average_montly_hours	time_spend_company	Work_accident	left	promotion_last_5years
count	14999.000000	14999.000000	14999.000000	14999.000000	14999.000000	14999.000000	14999.000000	14999.000000
mean	0.612834	0.716102	3.803054	201.050337	3.498233	0.144610	0.238083	0.021268
std	0.248631	0.171169	1.232592	49.943099	1.460136	0.351719	0.425924	0.144281
min	0.090000	0.360000	2.000000	96.000000	2.000000	0.000000	0.000000	0.000000
25%	0.440000	0.560000	3.000000	156.000000	3.000000	0.000000	0.000000	0.000000
50%	0.640000	0.720000	4.000000	200.000000	3.000000	0.000000	0.000000	0.000000
75%	0.820000	0.870000	5.000000	245.000000	4.000000	0.000000	0.000000	0.000000
max	1.000000	1.000000	7.000000	310.000000	10.000000	1.000000	1.000000	1.000000

We can see that there are about 15,000 rows of data, and they are all numerical, except for salary and department, which are strings. We then transform these two strings to numerical for the ability to alter the data and model it:

```
#Label Encoder variable
labelEncoder = preprocessing.LabelEncoder()

#Convert strings to numbers (0,1,etc.)
data['department']=labelEncoder.fit_transform(data['department'])
data['salary']=labelEncoder.fit_transform(data['salary'])
```

Now we have all numerical data, utilizing sklearn's preprocessing abilities to manipulate data before modeling. We then visualize the relationships between variables and can see how the data is related in accordance with the outputs. The top left shows clusters of outputs for (left), and we can see it is not linearly separable.

```
sns.pairplot( data=data, vars=('satisfaction_level', 'last_evaluation', 'number_project', 'average_montly_hours',
       'time_spend_company', 'Work_accident', 'promotion_last_5years', 'department', 'salary'), hue='left' )
```

Once the data was preprocessed and analyzed, we could split the data into features and labels, and begin modeling:

```
#Split data
x=data[['satisfaction_level', 'last_evaluation', 'number_project', 'average_montly_hours',
        'time_spend_company', 'Work_accident', 'promotion_last_5years', 'department', 'salary']]
y=data['left']

#Split the data into 75% train and 25% test
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25, random_state=42)
```

The data was split, followed by utilizing sklearn's neural network MLP Classifier library, to optimize a classification model and initialize it with our own custom values for hidden layer sizes, learning rates, etc.

```
#Import MLPClassifer
from sklearn.neural_network import MLPClassifier

# Create model object
"""
Creating instance of the MLPClassifier class from scikit-learn library in Python
The parameters passed to the constructor of the MLPClassifier instance include:
    hidden_layer_sizes: a tuple that specifies the number of neurons in each
    hidden layer of the neural network. In this case, there are
    two hidden layers with 20 and 19 neurons, respectively.

    random_state: a parameter that sets the seed for the random number
    generator used for weight initialization.

    activation='relu': The activation function to use for the hidden layers.
    In this case, the Rectified Linear Unit (ReLU) activation function is used.

    solver='adam': The optimization algorithm to use. Adam is an adaptive
    learning rate optimization algorithm that is well-suited for large datasets
    and complex neural networks

    alpha=0.0001: The L2 regularization parameter. This parameter controls
    the amount of weight decay applied to the network weights to prevent overfitting.

    verbose: a boolean flag that determines whether to print progress
    messages during training.

    learning_rate_init: parameter that specifies initial learning rate for NN.
"""
mlpFunction = MLPClassifier(hidden_layer_sizes = (20,19), activation = 'relu',
                            solver = 'adam', max_iter = 200, alpha = 0.0001,
                            random_state = 5, verbose = True, learning_rate_init = 0.01)

#Fit the data to the model
mlpFunction.fit(x_train,y_train)
```

In the above screenshot of code, we can see that the function is created, and is given hidden layer sizes for 2 layers of 20 and 19 neurons within each layer, respectively. This was chosen through trial and error, providing the best accuracy score from the several attempts. The comments in the code describe what each element of the function (through MLPClassifier's object) is responsible for, and how it affects the model development. The learning rate was established as 0.01 to hyper parametrize with smaller step sizes, ensuring more precision in the neural network. An 'adam' solver was added to optimize and fine tune an adaptive learning rate if it would be more appropriate than a set learning rate (it wasn't and can be discarded, as it did not change the accuracy). A rectilinear activation function was used as it provided the most accuracy when tested against the other main 4 functions (tanh, logistic, identity).

The results of the training and validation were outputted below:

```
Iteration 160, loss = 0.14850118
Iteration 161, loss = 0.14582332
Iteration 162, loss = 0.15619239
Iteration 163, loss = 0.16184947
Iteration 164, loss = 0.14689752
Iteration 165, loss = 0.15811022
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
```

```
MLPClassifier(hidden_layer_sizes=(20, 19), learning_rate_init=0.01,
              random_state=5, verbose=True)
```

```
#Use test data to make predictions
yPred = mlpFunction.predict(x_test)

#Import accuracy score and calculate accuracy
from sklearn.metrics import accuracy_score
accuracy_score(y_test,yPred)
```

```
0.9517333333333333
```

We can see that we got an accuracy score of 95.17%, meaning that 95% of predictions made by the MLP model would be accurate to predict whether an employee would leave the company based on the HR input variables.

Model 2 (MNIST)

The MNIST dataset has handwritten digits from 0-9, with a training set of 60,000 images, and a test set of 10,000 images. The data was normalized to fit inside 28x28 pixel boxes with a luminosity of grayscale pixels from 0-255 (white – black). The purpose of this dataset is to take the images and make an MLP model predict what digit they are through classification of 10 different possible outputs (0-9).

The analysis began by importing libraries and splitting the data (which is already split from the data source):

```
#Import modules
"""
    Sequential: This is the basic Keras model object.
    It allows you to create a linear stack of layers that can be easily trained and evaluated.

    Activation: This is a layer object that applies an activation function
    to the output of the previous layer. It is used to introduce non-linearity
    into the model and can be used with various activation functions such as relu, sigmoid, and tanh.

    Flatten: This layer object is used to flatten the input into a
    one-dimensional array. It is commonly used as the first layer of a
    neural network when the input data is structured as a multi-dimensional array.

    Dense: This layer object creates a fully connected layer of neurons.
    It takes the output of the previous layer and applies a linear
    transformation to it, followed by an activation function. The Dense layer
    has several parameters, such as the number of neurons, activation function,
    and initialization method, that can be used to customize the layer.
"""
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
```

In this MLP model development, we will utilize TensorFlow and its modules from the Keras library to create a neural network model. As we can see from the code output, the four different imports are defined for their use cases.

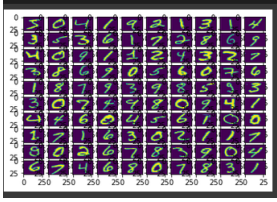
We then transform the data float type data and divide each data image pixel by 255 in order to normalize the data. We then print the description of the data of training and testing and see the shape of the data:

```
# Turn data to float
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

#normalize data by dividing by 255 (luminosity range)
x_train /= 255
x_test /= 255

#Print data output
print("Feature matrix x train:", x_train.shape)
print("Target matrix x test:", x_test.shape)
print("Feature matrix y train:", y_train.shape)
print("Target matrix y test:", y_test.shape)
```

```
Feature matrix x train: (60000, 28, 28)
Target matrix x test: (10000, 28, 28)
Feature matrix y train: (60000,)
Target matrix y test: (10000,)
```

We then plot the data to show a random sample of how the handwritten digits look, just to get a sense of how the data would then be categorized after stratified sampling occurs:

```
#plot a sample of the data in 28x28 pixels
#10 x 10 rows and columns
"""
The loop iterates over two ranges, i and j, and for each pair of i and j, it calls the imshow() function on the array element ax[i][j]. This function displays an image using the data in the array.
The image data is taken from the x_train array at index n, which is reshaped into a 28x28 pixel image.
The n variable is incremented by 1 for each iteration of the loop, so the next image in the x_train array will be used for the next iteration of the loop.
Once all images have been plotted, the plt.show() function is called to display the grid of images
"""
fig, ax = plt.subplots(10, 10)
n = 0
for i in range(10):
    for j in range(10):
        ax[i][j].imshow(x_train[n].reshape(28, 28), aspect='auto')
        n += 1
plt.show()
```



We then create the model with TensorFlow's abilities to create a sequential model object to develop a linear stack of layers that can be easily trained and evaluated:

```
#tf creation of MLP model with sequential stack of layers
mlpModel = Sequential([
    #Reshape and flatten data to 28x28 rows
    Flatten(input_shape=(28, 28)),
    #Layer 1 of neurons, with sigmoid activation function
    Dense(200, activation='sigmoid'),
    #Layer 2 of neurons, with sigmoid activation function
    Dense(100, activation='sigmoid'),
    #Layer 3 of neurons, with sigmoid activation function
    Dense(10, activation='sigmoid'),
])
```

We give the model two hidden layers with 200 and 100 neurons each, respectively, and provide it with sigmoid activation function to introduce non-linearity into the model. The data is first flattened into a one-dimensional array for the first layer of the neural network. The final dense layer of the neural network is given 10 neurons as the output of the previous layer.

```
"""
This code snippet is compiling a neural network model using the Keras library
in Python.

The model is being compiled with the 'adam' optimizer, which is a popular
stochastic gradient descent optimization algorithm that is often used in
deep learning, and can improve training speed and accuracy of NN.

The loss function used is 'sparse_categorical_crossentropy', which is commonly
used for multi-class classification problems where the labels are integers.

The metrics being used to evaluate the model during training and testing are
'accuracy', which is a common metric used for classification problems to
measure the percentage of correctly classified instances.

Overall, this code is configuring the model for training by specifying the
optimizer, loss function, and evaluation metrics.
"""
mlpModel.compile(optimizer='adam',
                 loss='sparse_categorical_crossentropy',
                 metrics=['accuracy'])
```

Once the model is instantiated, we then utilize the compile function from TensorFlow to provide it an optimizer, loss function, and metrics to best build the MLP model for us, as shown above. We use ‘adam’ as the optimizer to improve training speed and accuracy of the NN. We also use a sparse categorical cross entropy loss function because we have a multi-class classification problem, and our labels are integers. Finally, we have accuracy as the metric being used to evaluate the model during testing and training, to measure the percentage of correctly classified instances of handwritten digits being identified.

Then, we run the model and iterate it with 20 epochs, with a batch size of 2,000 training samples at a time before updating weights, and utilize a 75:25 train/test split:

```
#Model will iterate over the entire training dataset 20 times,
#Batch size of 2000, so model will process 2000 training examples at a time
#before updating weights
#Validation split is 0.25, so 25% of training data is used as validation
mlpModel.fit(x_train, y_train, epochs=20,
          batch_size=2000,
          validation_split=0.25)
```

```
Epoch 1/20
23/23 [==============================] - 3s 75ms/step - loss: 2.1761 - accuracy: 0.3639 - val_loss: 1.9202 - val_accuracy: 0.6701
Epoch 2/20
23/23 [==============================] - 2s 88ms/step - loss: 1.6723 - accuracy: 0.6820 - val_loss: 1.3611 - val_accuracy: 0.7496
Epoch 3/20
23/23 [==============================] - 1s 62ms/step - loss: 1.1573 - accuracy: 0.7816 - val_loss: 0.9331 - val_accuracy: 0.8216
Epoch 4/20
23/23 [==============================] - 1s 57ms/step - loss: 0.8221 - accuracy: 0.8361 - val_loss: 0.6799 - val_accuracy: 0.8645
Epoch 5/20
23/23 [==============================] - 1s 60ms/step - loss: 0.6226 - accuracy: 0.8676 - val_loss: 0.5304 - val_accuracy: 0.8848
Epoch 6/20
23/23 [==============================] - 1s 50ms/step - loss: 0.5017 - accuracy: 0.8854 - val_loss: 0.4419 - val_accuracy: 0.8953
Epoch 7/20
23/23 [==============================] - 1s 53ms/step - loss: 0.4274 - accuracy: 0.8968 - val_loss: 0.3868 - val_accuracy: 0.9021
Epoch 8/20
23/23 [==============================] - 1s 56ms/step - loss: 0.3786 - accuracy: 0.9037 - val_loss: 0.3509 - val_accuracy: 0.9090
Epoch 9/20
23/23 [==============================] - 1s 52ms/step - loss: 0.3451 - accuracy: 0.9088 - val_loss: 0.3234 - val_accuracy: 0.9135
Epoch 10/20
23/23 [==============================] - 1s 54ms/step - loss: 0.3191 - accuracy: 0.9142 - val_loss: 0.3031 - val_accuracy: 0.9175
Epoch 11/20
23/23 [==============================] - 2s 80ms/step - loss: 0.2989 - accuracy: 0.9180 - val_loss: 0.2873 - val_accuracy: 0.9211
Epoch 12/20
23/23 [==============================] - 2s 66ms/step - loss: 0.2824 - accuracy: 0.9217 - val_loss: 0.2730 - val_accuracy: 0.9243
Epoch 13/20
23/23 [==============================] - 1s 57ms/step - loss: 0.2675 - accuracy: 0.9253 - val_loss: 0.2618 - val_accuracy: 0.9264
Epoch 14/20
23/23 [==============================] - 1s 53ms/step - loss: 0.2548 - accuracy: 0.9286 - val_loss: 0.2519 - val_accuracy: 0.9292
Epoch 15/20
23/23 [==============================] - 1s 58ms/step - loss: 0.2437 - accuracy: 0.9311 - val_loss: 0.2426 - val_accuracy: 0.9305
Epoch 16/20
23/23 [==============================] - 1s 57ms/step - loss: 0.2333 - accuracy: 0.9341 - val_loss: 0.2343 - val_accuracy: 0.9325
Epoch 17/20
23/23 [==============================] - 1s 57ms/step - loss: 0.2239 - accuracy: 0.9366 - val_loss: 0.2263 - val_accuracy: 0.9349
Epoch 18/20
23/23 [==============================] - 1s 48ms/step - loss: 0.2148 - accuracy: 0.9389 - val_loss: 0.2193 - val_accuracy: 0.9367
Epoch 19/20
23/23 [==============================] - 1s 55ms/step - loss: 0.2065 - accuracy: 0.9414 - val_loss: 0.2128 - val_accuracy: 0.9391
Epoch 20/20
23/23 [==============================] - 2s 72ms/step - loss: 0.1991 - accuracy: 0.9435 - val_loss: 0.2072 - val_accuracy: 0.9413
<keras.callbacks.History at 0x7fede20ad790>
```

The model runs for 20 iterations and stops, outputting each iteration change in loss and accuracy.

We then output the final data accuracy with a print function:

```
#Print results of model accuracy
results = mlpModel.evaluate(x_test,  y_test, verbose = 0)
print('Test loss & Test accuracy:', results)
```

```
Test loss & Test accuracy: [0.20260195434093475, 0.940500020980835]
```

The overall accuracy after 20 epochs was 94.05%, meaning that 94% of handwritten digits were correctly classified. We could improve accuracy by further fine-tuning epoch and batch size to allow the data to continue to iterate and improve for a better score, but overall, the 94% was an improvement after constant tuning of details and ending at this point.

Model 3 (IRIS)

The IRIS dataset is a widely used benchmark dataset with 150 samples of iris flowers, with 50 samples each for three different species (Setosa, Versicolor, and Virginica). Each sample has four input samples characterizing its features: sepal length and width, and petal length and width in cm. While in general cases there can be linear classification to determine each of the three species, there is a lot of overlap in data having 4 features and 3 species all being used, and non-linear sampling can be much more useful to classify the entire dataset holistically with MLP.

Modules and libraries were imported, utilizing sklearn's MLPClassifier used in Model 1 with HR data, and the data was imported and read:

```
#Import modules
#neural network MLP object
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics
import numpy as np
import pandas as pd
import seaborn as sns
```

```
#import and read data
data = pd.read_csv('Iris.csv')
data.head()
```

	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	1	5.1	3.5	1.4	0.2	Iris-setosa
1	2	4.9	3.0	1.4	0.2	Iris-setosa
2	3	4.7	3.2	1.3	0.2	Iris-setosa
3	4	4.6	3.1	1.5	0.2	Iris-setosa
4	5	5.0	3.6	1.4	0.2	Iris-setosa

The data was analyzed and described to showcase its features:

```
#describe data containment
data.describe()
```

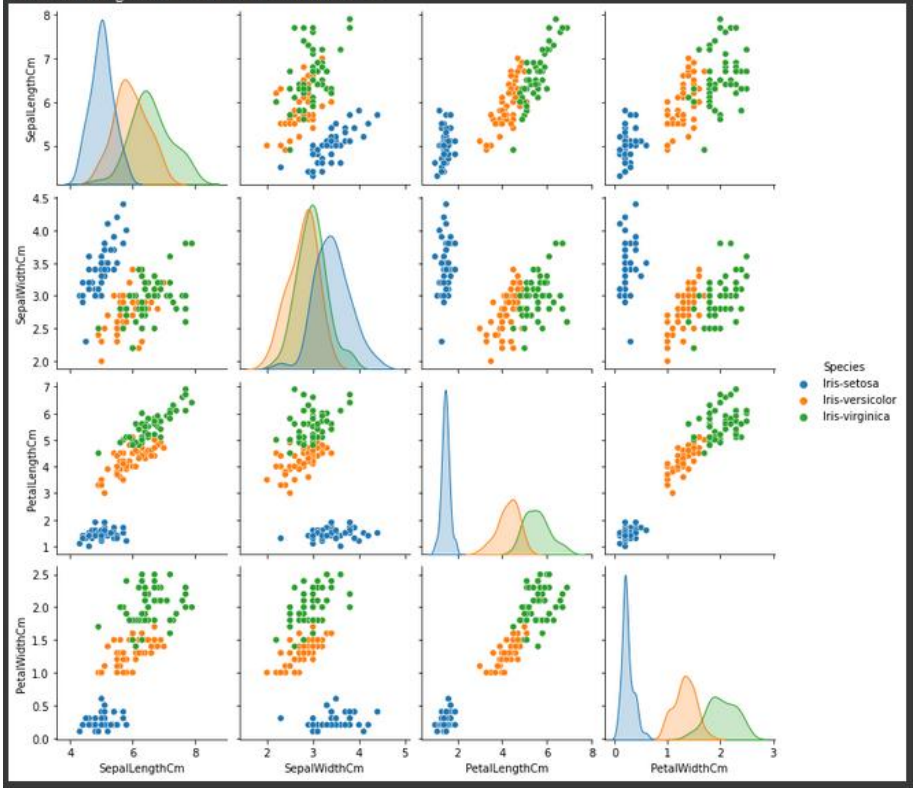
	Id	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm
count	150.000000	150.000000	150.000000	150.000000	150.000000
mean	75.500000	5.843333	3.054000	3.758667	1.198667
std	43.445368	0.828066	0.433594	1.764420	0.763161
min	1.000000	4.300000	2.000000	1.000000	0.100000
25%	38.250000	5.100000	2.800000	1.600000	0.300000
50%	75.500000	5.800000	3.000000	4.350000	1.300000
75%	112.750000	6.400000	3.300000	5.100000	1.800000
max	150.000000	7.900000	4.400000	6.900000	2.500000

We can see that there are 4 features characterizing measurements of each flower, and there is 1 output variable with 3 possible classifications.

A seaborn pair plot was created to visualize the dataset and show how the 3 species of flowers cluster differently based on various combinations of the four input variables:

```
#visualize plot relationship between inputs and output
sns.pairplot(data = data, vars = ('SepalLengthCm','SepalWidthCm','PetalLengthCm','PetalWidthCm'),
             hue = 'Species' )
```

```
<seaborn.axisgrid.PairGrid at 0x7f439b9afd60>
```



As we can see, the data is not neatly linearly separable, and with four input variables, there is no one combination of features that would most accurately predict the species, so a MLP model will be created.

```
#Create model object
"""
When LBFGS is used with MLPClassifier, the algorithm uses the LBFGS optimization
method to minimize the loss function (e.g., cross-entropy loss) of the MLP during
training. The LBFGS algorithm is a popular choice for optimizing the weights of MLPs
because it can handle a large number of parameters efficiently and can converge quickly
to a good solution.

solver = 'lbfgs': This parameter specifies the optimization algorithm to be used
by the MLPClassifier. In this case, the Limited-memory Broyden-Fletcher-Goldfarb-Shanno
(LBFGS) algorithm is used.

alpha = 0.0001: This parameter is the L2 penalty (regularization term) parameter.
It helps to prevent overfitting by adding a penalty term to the loss function that
is proportional to the square of the magnitude of the weight coefficients.

hidden_layer_sizes = (5,4): This parameter specifies the number of neurons in
each hidden layer of the MLP. In this case, the MLP has two hidden layers with
5 and 4 neurons, respectively.

random_state = 5: This parameter sets the seed for the random number generator
used by the MLPClassifier. This ensures that the results are reproducible.

verbose = True: This parameter controls the amount of output printed during training.
When set to True, the MLPClassifier prints progress messages to the console.

max_iter = 200: This parameter sets the maximum number of iterations for the MLPClassifier
to perform during training. If the optimization algorithm has not converged after
this many iterations, training will be stopped.
"""
mlpFunction = MLPClassifier(solver = 'lbfgs', alpha = 0.0001, hidden_layer_sizes = (5,4),
                            random_state = 5, verbose = True, max_iter = 200)

#fit the data to the model
mlpFunction.fit(x_train, y_train)
```

```
MLPClassifier
MLPClassifier(hidden_layer_sizes=(5, 4), random_state=5, solver='lbfgs',
              verbose=True)
```

The model was created similarly to Model 1 with HR data and used smaller sizes of neurons for hidden layers of only 5 and 4 neurons, since it was a smaller dataset. The alpha was the same to prevent overfitting by adding a penalty to the loss function, proportional to the square of the magnitude of the weight coefficient. The solver optimizer was ‘lbfgs,’ which is an algorithm that minimizes the loss function and optimizes weights of MLPs in a quick and efficient manner. Once the model class was created and filled with parameters, it was fit to the training data and run.

The actual values and predicted values were outputted, and an accuracy score was calculated:

```
#actual values being printed of species
predictionValues = mlpFunction.predict(x_test)
print(predictionValues)
```

```
['Iris-versicolor' 'Iris-setosa' 'Iris-virginica' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-setosa' 'Iris-versicolor' 'Iris-virginica'
 'Iris-virginica' 'Iris-versicolor' 'Iris-virginica' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-versicolor'
 'Iris-virginica' 'Iris-versicolor' 'Iris-versicolor' 'Iris-virginica'
 'Iris-setosa' 'Iris-virginica' 'Iris-setosa' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-versicolor'
 'Iris-setosa' 'Iris-setosa' 'Iris-virginica' 'Iris-versicolor'
 'Iris-setosa']
```

```
#predicted values being printed
print(y_test.values)
```

```
['Iris-versicolor' 'Iris-setosa' 'Iris-virginica' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-setosa' 'Iris-versicolor' 'Iris-virginica'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-virginica' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-versicolor'
 'Iris-virginica' 'Iris-versicolor' 'Iris-versicolor' 'Iris-virginica'
 'Iris-setosa' 'Iris-virginica' 'Iris-setosa' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-versicolor'
 'Iris-setosa' 'Iris-setosa' 'Iris-virginica' 'Iris-versicolor'
 'Iris-setosa']
```

```
print('The accuracy of the Multi-Layer Perceptron is:', metrics.accuracy_score(predictionValues,y_test)*100, '%')
```

```
The accuracy of the Multi-Layer Perceptron is: 97.36842105263158 %
```

As we can see, the predicted values and actual values were mostly spot on except for only one sample, which incorrectly predicted a versicolor flower as a virginica flower. We then calculated an accuracy score and found it to be 97.37% accurate, which was the best of the three models we tested MLP on.

Outputs

Model 1:

```
Iteration 160, loss = 0.14850118
Iteration 161, loss = 0.14582332
Iteration 162, loss = 0.15619239
Iteration 163, loss = 0.16184947
Iteration 164, loss = 0.14689752
Iteration 165, loss = 0.15811022
Training loss did not improve more than tol=0.000100 for 10 consecutive epochs. Stopping.
```

```
MLPClassifier(hidden_layer_sizes=(20, 19), learning_rate_init=0.01,
              random_state=5, verbose=True)
```

```
#Use test data to make predictions
yPred = mlpFunction.predict(x_test)

#Import accuracy score and calculate accuracy
from sklearn.metrics import accuracy_score
accuracy_score(y_test,yPred)
```

```
0.9517333333333333
```

Model 2:

```
#Print results of model accuracy
results = mlpModel.evaluate(x_test,  y_test, verbose = 0)
print('Test loss & Test accuracy:', results)
```

```
Test loss & Test accuracy: [0.20260195434093475, 0.940500020980835]
```

Model 3:

```
#actual values being printed of species
predictionValues = mlpFunction.predict(x_test)
print(predictionValues)
```

```
['Iris-versicolor' 'Iris-setosa' 'Iris-virginica' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-setosa' 'Iris-versicolor' 'Iris-virginica'
 'Iris-virginica' 'Iris-versicolor' 'Iris-virginica' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-versicolor'
 'Iris-virginica' 'Iris-versicolor' 'Iris-versicolor' 'Iris-virginica'
 'Iris-setosa' 'Iris-virginica' 'Iris-setosa' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-versicolor'
 'Iris-setosa' 'Iris-setosa' 'Iris-virginica' 'Iris-versicolor'
 'Iris-setosa']
```

```
#predicted values being printed
print(y_test.values)
```

```
['Iris-versicolor' 'Iris-setosa' 'Iris-virginica' 'Iris-versicolor'
 'Iris-versicolor' 'Iris-setosa' 'Iris-versicolor' 'Iris-virginica'
 'Iris-versicolor' 'Iris-versicolor' 'Iris-virginica' 'Iris-setosa'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-versicolor'
 'Iris-virginica' 'Iris-versicolor' 'Iris-versicolor' 'Iris-virginica'
 'Iris-setosa' 'Iris-virginica' 'Iris-setosa' 'Iris-virginica'
 'Iris-virginica' 'Iris-virginica' 'Iris-virginica' 'Iris-virginica'
 'Iris-setosa' 'Iris-setosa' 'Iris-setosa' 'Iris-versicolor'
 'Iris-setosa' 'Iris-setosa' 'Iris-virginica' 'Iris-versicolor'
 'Iris-setosa']
```

```
print('The accuracy of the Multi-Layer Perceptron is:', metrics.accuracy_score(predictionValues,y_test)*100, '%')
```

```
The accuracy of the Multi-Layer Perceptron is: 97.36842105263158 %
```