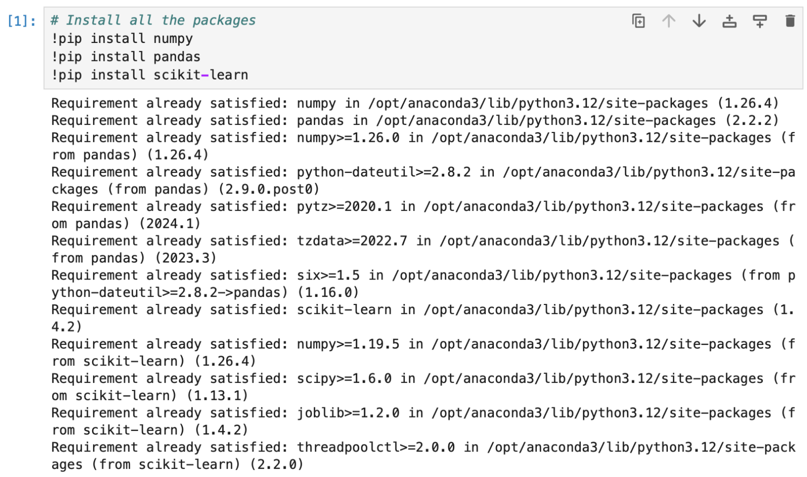
You are asked in this assignment to use the modeling techniques described in lecture 3 to develop and test multilayer perceptron (MLP) that are able to classify Iris type (Setosa, Versicolor or Virginica) in two ways as well as to compare it with another neural net classifier of your own. The first way to classify Iris using built-in MLP (e.g.SKLearn MLPClassifier) and the second way by building your own MLP like the example provided in this link.You are required to compare the three models as well as to test their performances.

In response to the question, I have developed three models as outlined below:

- MLP using MLPClassifier
- MLP implemented with PyTorch
- MLP built using NumPy (custom implementation)

The following are the common steps carried out in all three models:

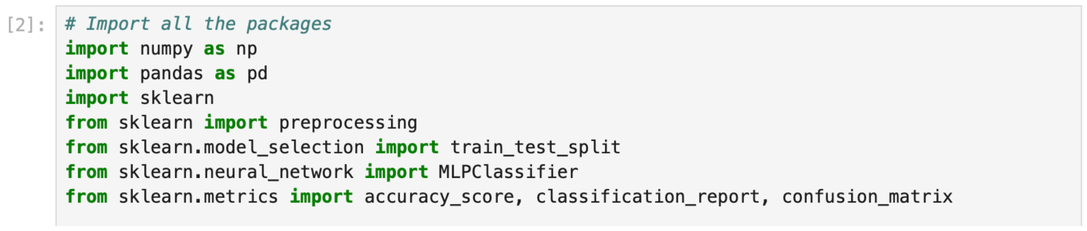
1. Installing and importing all the necessary packages required for the model.
   For MLPClassifier:
      Install:



```
[1]: # Install all the packages
     !pip install numpy
     !pip install pandas
     !pip install scikit-learn
```

```
Requirement already satisfied: numpy in /opt/anaconda3/lib/python3.12/site-packages (1.26.4)
Requirement already satisfied: pandas in /opt/anaconda3/lib/python3.12/site-packages (2.2.2)
Requirement already satisfied: numpy>=1.26.0 in /opt/anaconda3/lib/python3.12/site-packages (f
rom pandas) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /opt/anaconda3/lib/python3.12/site-pa
ckages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /opt/anaconda3/lib/python3.12/site-packages (fr
om pandas) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in /opt/anaconda3/lib/python3.12/site-packages (
from pandas) (2023.3)
Requirement already satisfied: six>=1.5 in /opt/anaconda3/lib/python3.12/site-packages (from p
ython-dateutil>=2.8.2->pandas) (1.16.0)
Requirement already satisfied: scikit-learn in /opt/anaconda3/lib/python3.12/site-packages (1.
4.2)
Requirement already satisfied: numpy>=1.19.5 in /opt/anaconda3/lib/python3.12/site-packages (f
rom scikit-learn) (1.26.4)
Requirement already satisfied: scipy>=1.6.0 in /opt/anaconda3/lib/python3.12/site-packages (fr
om scikit-learn) (1.13.1)
Requirement already satisfied: joblib>=1.2.0 in /opt/anaconda3/lib/python3.12/site-packages (f
rom scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /opt/anaconda3/lib/python3.12/site-pack
ages (from scikit-learn) (2.2.0)
```

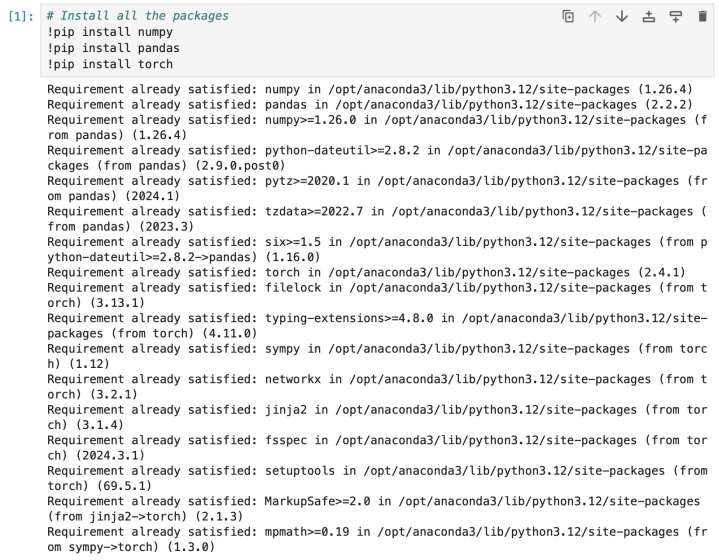
Import:



```
[2]: # Import all the packages
     import numpy as np
     import pandas as pd
     import sklearn
     from sklearn import preprocessing
     from sklearn.model_selection import train_test_split
     from sklearn.neural_network import MLPClassifier
     from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
```

For pytorch:

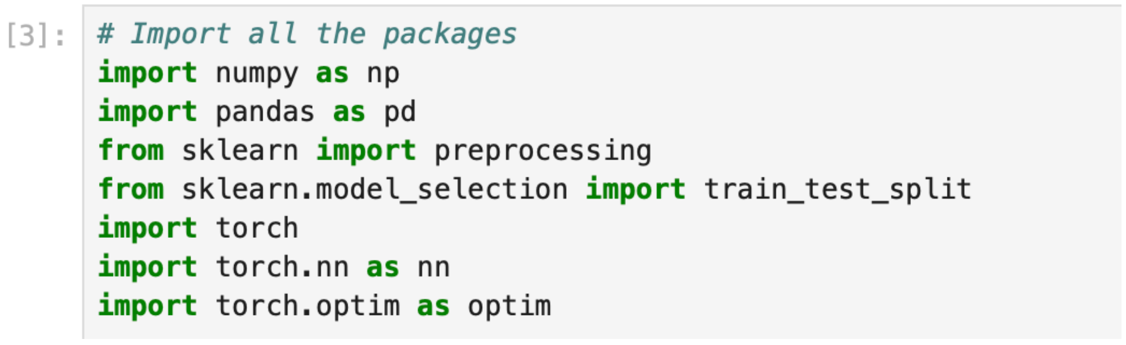
Install:



```
[1]: # Install all the packages
!pip install numpy
!pip install pandas
!pip install torch
```

```
Requirement already satisfied: numpy in /opt/anaconda3/lib/python3.12/site-packages (1.26.4)
Requirement already satisfied: pandas in /opt/anaconda3/lib/python3.12/site-packages (2.2.2)
Requirement already satisfied: numpy>=1.26.0 in /opt/anaconda3/lib/python3.12/site-packages (f
rom pandas) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /opt/anaconda3/lib/python3.12/site-pa
ckages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /opt/anaconda3/lib/python3.12/site-packages (fr
om pandas) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in /opt/anaconda3/lib/python3.12/site-packages (
from pandas) (2023.3)
Requirement already satisfied: six>=1.5 in /opt/anaconda3/lib/python3.12/site-packages (from p
ython-dateutil>=2.8.2->pandas) (1.16.0)
Requirement already satisfied: torch in /opt/anaconda3/lib/python3.12/site-packages (2.4.1)
Requirement already satisfied: filelock in /opt/anaconda3/lib/python3.12/site-packages (from t
orch) (3.13.1)
Requirement already satisfied: typing-extensions>=4.8.0 in /opt/anaconda3/lib/python3.12/site-
packages (from torch) (4.11.0)
Requirement already satisfied: sympy in /opt/anaconda3/lib/python3.12/site-packages (from torc
h) (1.12)
Requirement already satisfied: networkx in /opt/anaconda3/lib/python3.12/site-packages (from t
orch) (3.2.1)
Requirement already satisfied: jinja2 in /opt/anaconda3/lib/python3.12/site-packages (from tor
ch) (3.1.4)
Requirement already satisfied: fsspec in /opt/anaconda3/lib/python3.12/site-packages (from tor
ch) (2024.3.1)
Requirement already satisfied: setuptools in /opt/anaconda3/lib/python3.12/site-packages (from
torch) (69.5.1)
Requirement already satisfied: MarkupSafe>=2.0 in /opt/anaconda3/lib/python3.12/site-packages
(from jinja2->torch) (2.1.3)
Requirement already satisfied: mpmath>=0.19 in /opt/anaconda3/lib/python3.12/site-packages (fr
om sympy->torch) (1.3.0)
```

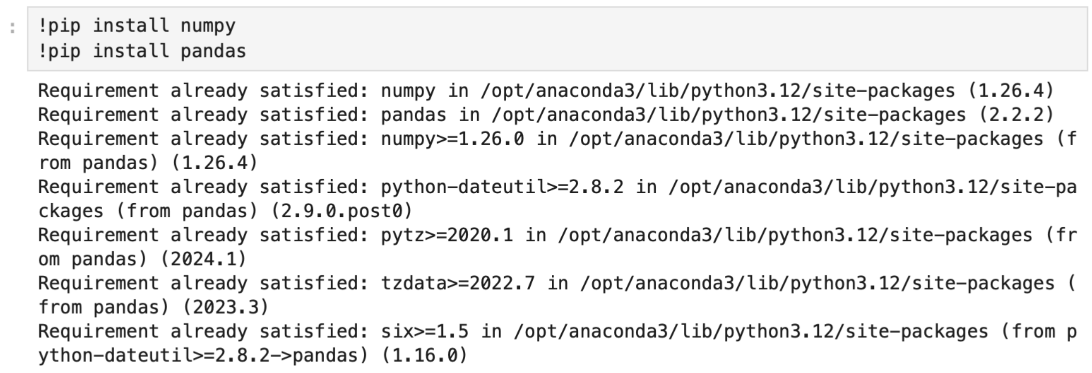
Import:



```
[3]: # Import all the packages
import numpy as np
import pandas as pd
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
import torch
import torch.nn as nn
import torch.optim as optim
```

For numpy:

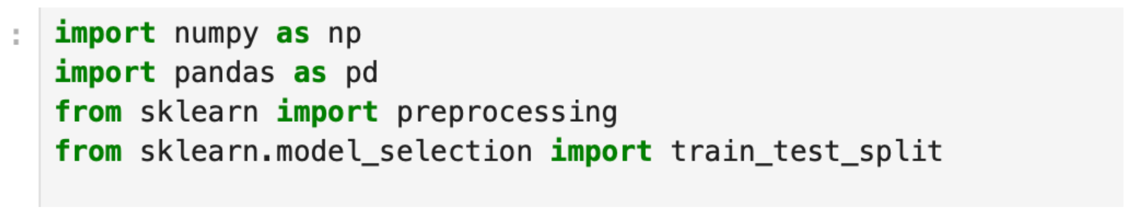
Install:



```
!pip install numpy
!pip install pandas
```

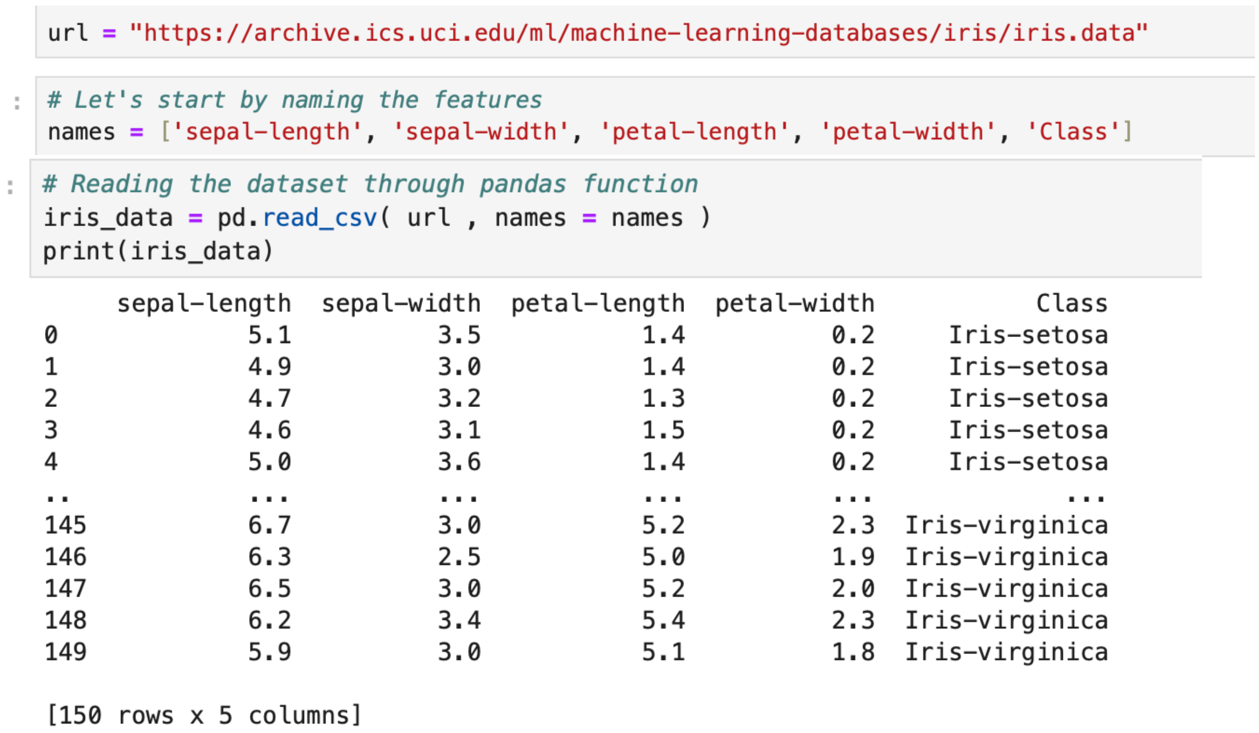
```
Requirement already satisfied: numpy in /opt/anaconda3/lib/python3.12/site-packages (1.26.4)
Requirement already satisfied: pandas in /opt/anaconda3/lib/python3.12/site-packages (2.2.2)
Requirement already satisfied: numpy>=1.26.0 in /opt/anaconda3/lib/python3.12/site-packages (f
rom pandas) (1.26.4)
Requirement already satisfied: python-dateutil>=2.8.2 in /opt/anaconda3/lib/python3.12/site-pa
ckages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in /opt/anaconda3/lib/python3.12/site-packages (fr
om pandas) (2024.1)
Requirement already satisfied: tzdata>=2022.7 in /opt/anaconda3/lib/python3.12/site-packages (
from pandas) (2023.3)
Requirement already satisfied: six>=1.5 in /opt/anaconda3/lib/python3.12/site-packages (from p
ython-dateutil>=2.8.2->pandas) (1.16.0)
```

Import:



```
import numpy as np
import pandas as pd
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
```

2. Read the dataset from the URL and label through pandas function.



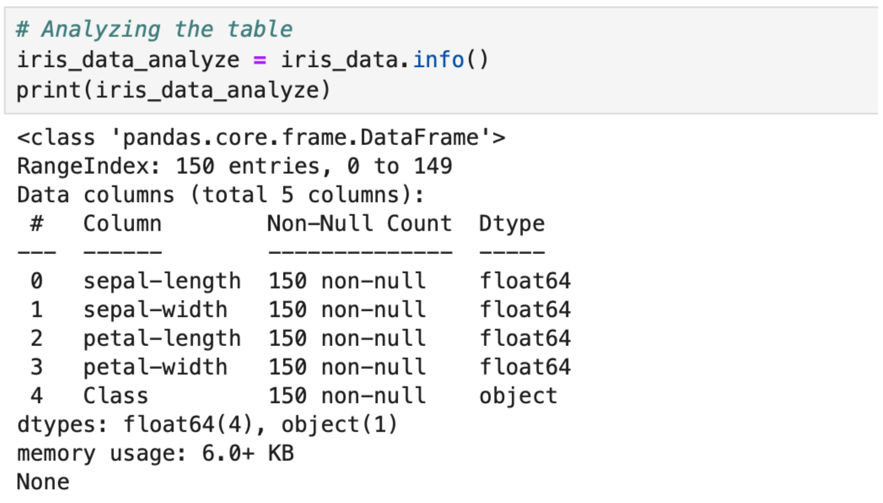
```
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
```

```
# Let's start by naming the features
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']
```

```
# Reading the dataset through pandas function
iris_data = pd.read_csv( url , names = names )
print(iris_data)
```

```
     sepal-length  sepal-width  petal-length  petal-width           Class
0             5.1          3.5           1.4          0.2     Iris-setosa
1             4.9          3.0           1.4          0.2     Iris-setosa
2             4.7          3.2           1.3          0.2     Iris-setosa
3             4.6          3.1           1.5          0.2     Iris-setosa
4             5.0          3.6           1.4          0.2     Iris-setosa
..            ...          ...           ...          ...             ...
145           6.7          3.0           5.2          2.3  Iris-virginica
146           6.3          2.5           5.0          1.9  Iris-virginica
147           6.5          3.0           5.2          2.0  Iris-virginica
148           6.2          3.4           5.4          2.3  Iris-virginica
149           5.9          3.0           5.1          1.8  Iris-virginica

[150 rows x 5 columns]
```

3. Analyzing the dataset .

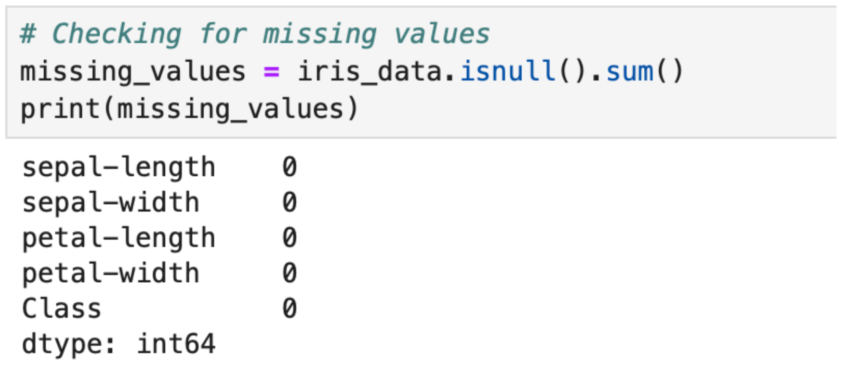


```
# Analyzing the table
iris_data_analyze = iris_data.info()
print(iris_data_analyze)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   sepal-length  150 non-null    float64
 1   sepal-width   150 non-null    float64
 2   petal-length  150 non-null    float64
 3   petal-width   150 non-null    float64
 4   Class         150 non-null    object 
dtypes: float64(4), object(1)
memory usage: 6.0+ KB
None
```

(Note: There is one fields with data type object. It needs to be converted into numerical values.)

4. Checking the missing values.

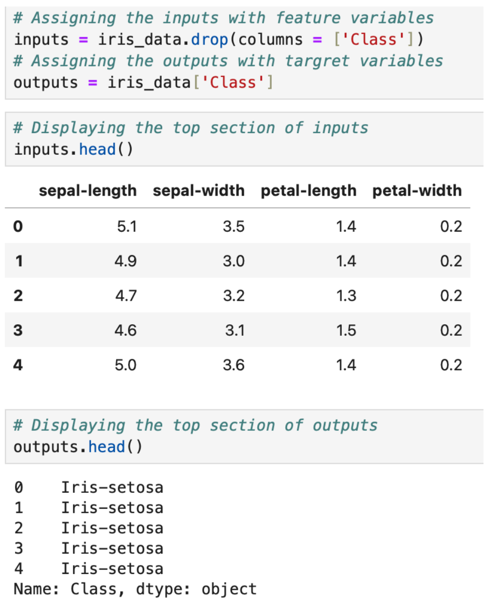


```
# Checking for missing values
missing_values = iris_data.isnull().sum()
print(missing_values)
```

```
sepal-length    0
sepal-width     0
petal-length    0
petal-width     0
Class           0
dtype: int64
```

(Note: There are no missing values. So, there is no need to handle the data.)

5. Assigning the feature variables and target variables.



```
# Assigning the inputs with feature variables
inputs = iris_data.drop(columns = ['Class'])
# Assigning the outputs with targret variables
outputs = iris_data['Class']
```

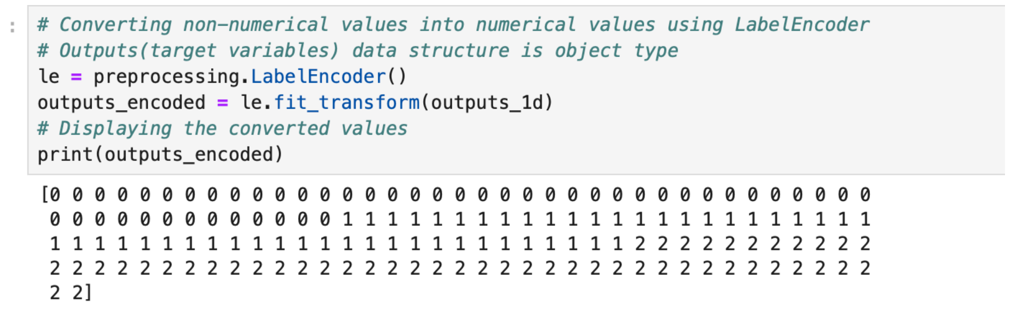
```
# Displaying the top section of inputs
inputs.head()
```

	sepal-length	sepal-width	petal-length	petal-width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

```
# Displaying the top section of outputs
outputs.head()
```

```
0    Iris-setosa
1    Iris-setosa
2    Iris-setosa
3    Iris-setosa
4    Iris-setosa
Name: Class, dtype: object
```

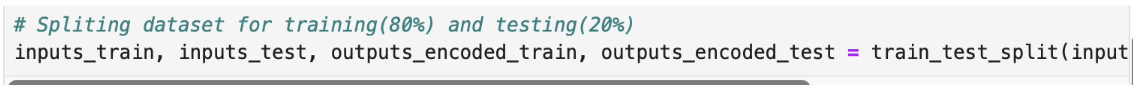
6. Converting non_numerical values into numerical values.



```
# Converting non-numerical values into numerical values using LabelEncoder
# Outputs(target variables) data structure is object type
le = preprocessing.LabelEncoder()
outputs_encoded = le.fit_transform(outputs_1d)
# Displaying the converted values
print(outputs_encoded)
```

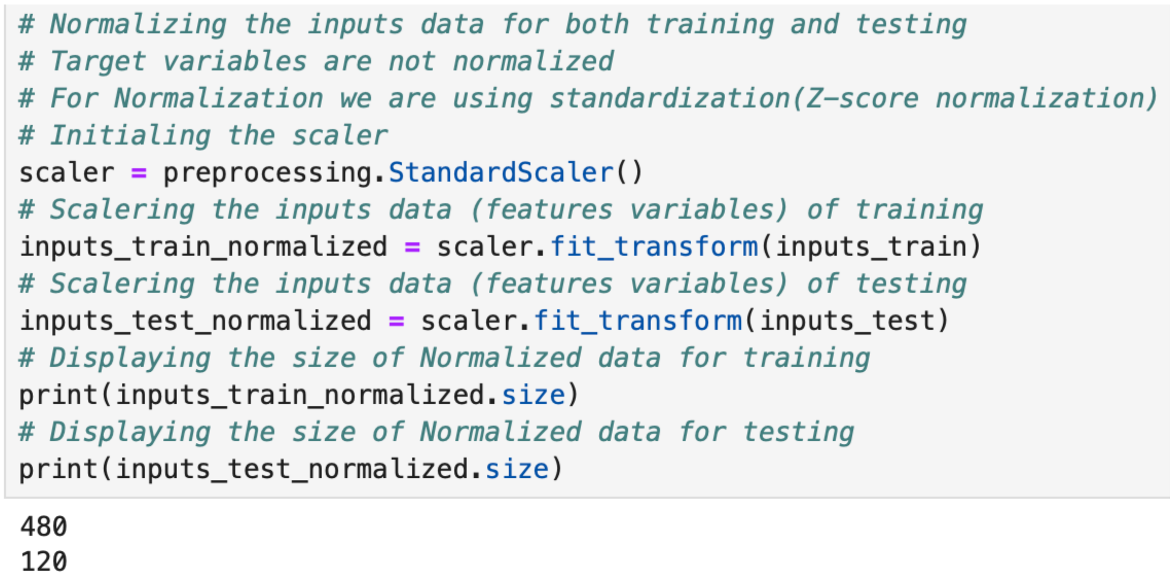
```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2]
```

7. Splitting dataset into training dataset and testing dataset.



```
# Spliting dataset for training(80%) and testing(20%)
inputs_train, inputs_test, outputs_encoded_train, outputs_encoded_test = train_test_split(input
```

8. Normalizing the data.



```
# Normalizing the inputs data for both training and testing
# Target variables are not normalized
# For Normalization we are using standardization(Z-score normalization)
# Initialing the scaler
scaler = preprocessing.StandardScaler()
# Scalering the inputs data (features variables) of training
inputs_train_normalized = scaler.fit_transform(inputs_train)
# Scalering the inputs data (features variables) of testing
inputs_test_normalized = scaler.fit_transform(inputs_test)
# Displaying the size of Normalized data for training
print(inputs_train_normalized.size)
# Displaying the size of Normalized data for testing
print(inputs_test_normalized.size)
```

```
480
120
```

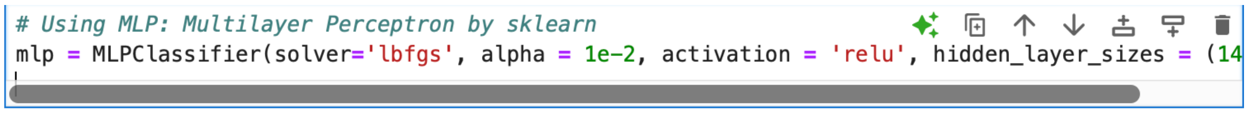
These are the standard steps that need to be performed in all MLP models. After completing these, a specific method must be applied for each individual model.

1) MLP by MLPClassifier:

MLPClassifier is a built-in function from scikit-learn that is used for designing neural networks. It is directly imported from the scikit-learn package and accepts the following arguments:

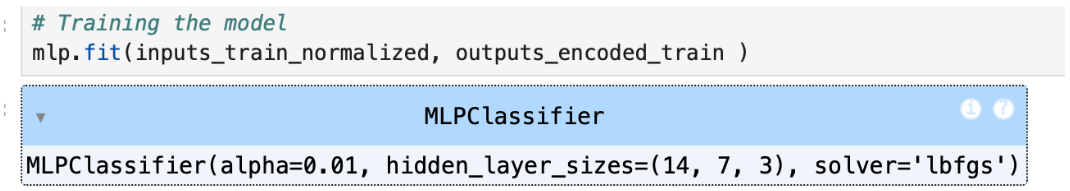
solver : lbfgs (Optimizer algorithms)
alpha : le-2 (Learning rate)
activation: relu (Activation functions)
hidden_layer_size : (14,7,3) (Number of hidden layer)
max_iterate : not mentioned (Number of loops) (All ready controlled by LBFGS optimizer algorithms)

- Designing the neural network



```
# Using MLP: Multilayer Perceptron by sklearn
mlp = MLPClassifier(solver='lbfgs', alpha = 1e-2, activation = 'relu', hidden_layer_sizes = (14
```

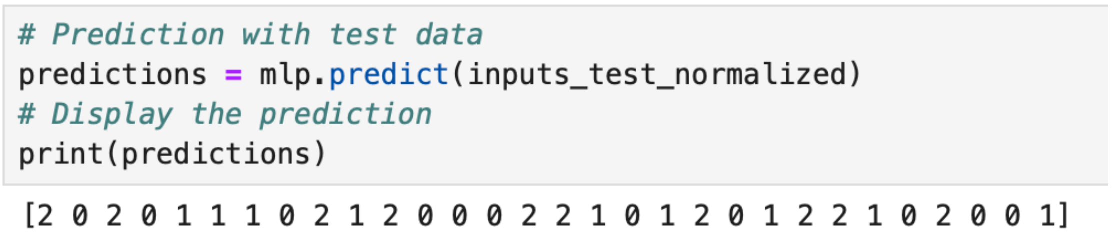
- Training the model with training datasets with built in method called .fit which takes the two arguments input and output



```
# Training the model
mlp.fit(inputs_train_normalized, outputs_encoded_train )
```

```
MLPClassifier
MLPClassifier(alpha=0.01, hidden_layer_sizes=(14, 7, 3), solver='lbfgs')
```

- Prediction is made using the test datasets with built in method called .predict which takes the one argument input

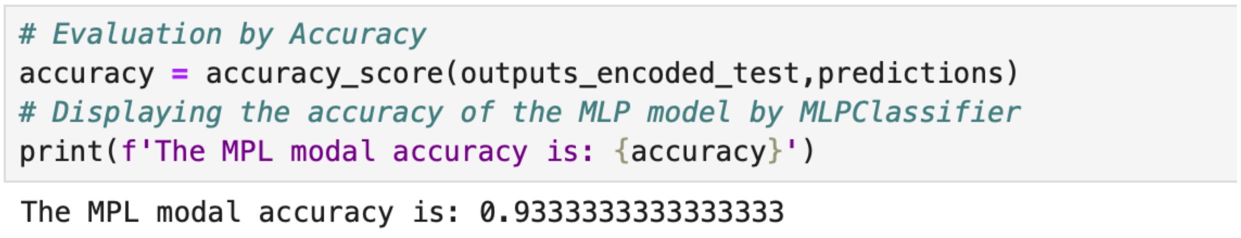


```
# Prediction with test data
predictions = mlp.predict(inputs_test_normalized)
# Display the prediction
print(predictions)
```

```
[2 0 2 0 1 1 1 0 2 1 2 0 0 0 2 2 1 0 1 2 0 1 2 2 1 0 2 0 0 1]
```

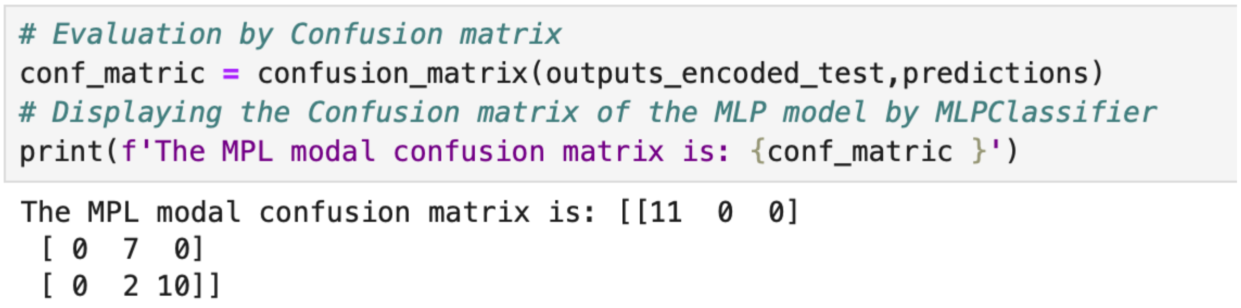
- Predictions have been made, and now it’s time to evaluate the model's performance. There are various methods to assess performance, and among them, we will utilize metric evaluation, which includes:

• Accuracy
• Confusion Matrix
• Precision, Recall, and F1-Score



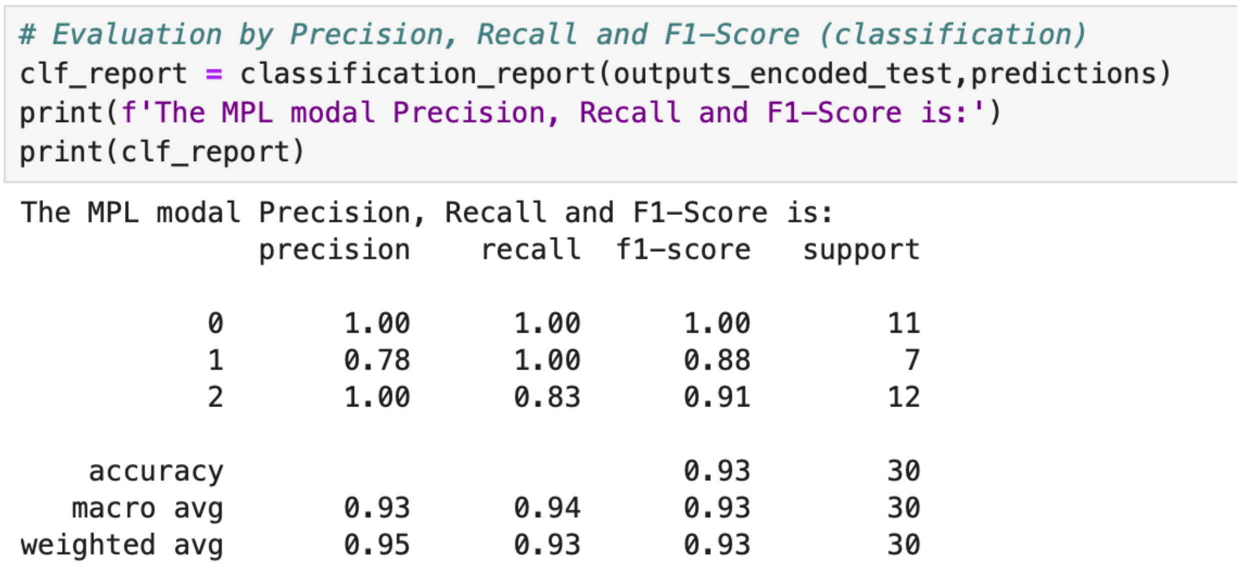
```
# Evaluation by Accuracy
accuracy = accuracy_score(outputs_encoded_test,predictions)
# Displaying the accuracy of the MLP model by MLPClassifier
print(f'The MPL modal accuracy is: {accuracy}')
```

```
The MPL modal accuracy is: 0.9333333333333333
```



```
# Evaluation by Confusion matrix
conf_matric = confusion_matrix(outputs_encoded_test,predictions)
# Displaying the Confusion matrix of the MLP model by MLPClassifier
print(f'The MPL modal confusion matrix is: {conf_matric }')
```

```
The MPL modal confusion matrix is: [[11  0  0]
 [ 0  7  0]
 [ 0  2 10]]
```



```
# Evaluation by Precision, Recall and F1-Score (classification)
clf_report = classification_report(outputs_encoded_test,predictions)
print(f'The MPL modal Precision, Recall and F1-Score is:')
print(clf_report)
```

```
The MPL modal Precision, Recall and F1-Score is:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       0.78      1.00      0.88         7
           2       1.00      0.83      0.91        12

    accuracy                           0.93        30
   macro avg       0.93      0.94      0.93        30
weighted avg       0.95      0.93      0.93        30
```

Drawing conclusion:
The model's accuracy is 93%, and it tends to confuse class 1 with class 2 the most.

2) MLP implemented with PyTorch:

PyTorch is another built-in function that can be imported from the torch package.

- Convert dataset into pytorch tensors

```
 # Convert the dataset to PyTorch tensors
c_inputs_train_normalized = torch.tensor(inputs_train_normalized, dtype=torch.float32)
c_inputs_test_normalized = torch.tensor(inputs_test_normalized, dtype=torch.float32)
c_outputs_encoded_train = torch.tensor(outputs_encoded_train, dtype=torch.long)
c_outputs_encoded_test = torch.tensor(outputs_encoded_test, dtype=torch.long)
```

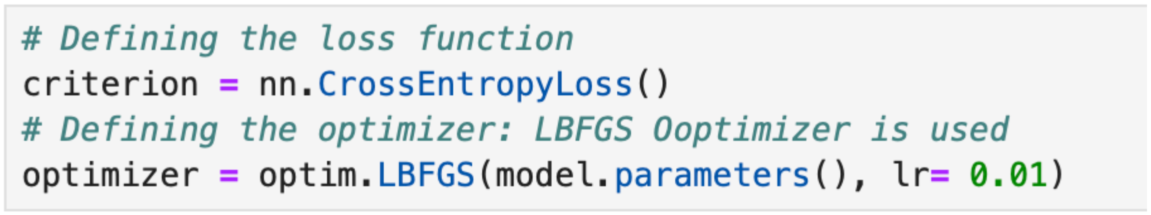
- Designing the neural network

```
# Designing MLP model
# Constructor : initializing
class MLP(nn.Module):
    def __init__(self, input_size, hidden_layer1, hidden_layer2, hidden_layer3, output_size):
        super(MLP, self).__init__()

        # Designing the layer
        self.input_hidden1_layer = nn.Linear(input_size, hidden_layer1)
        self.hidden1_hidden2_layer = nn.Linear(hidden_layer1, hidden_layer2)
        self.hidden2_hidden3_layer = nn.Linear(hidden_layer2, hidden_layer3)
        self.hidden3_output_layer = nn.Linear(hidden_layer3, output_size)
        # Defining the Activation Function: using rectified linear unit
        self.activation_function = nn.ReLU()

    # Passing the inputs into forward direction and computing
    def feedforwarding(self, inputs):
        computed_first_conn = self.input_hidden1_layer(inputs)
        computed_first_conn_with_activation = self.activation_function(computed_first_conn)
        computed_second_conn = self.hidden1_hidden2_layer(computed_first_conn_with_activation)
        computed_second_conn_with_activation = self.activation_function(computed_second_conn)
        computed_third_conn = self.hidden2_hidden3_layer(computed_second_conn_with_activation)
        computed_third_conn_with_activation = self.activation_function(computed_third_conn)
        computed_outputs = self.hidden3_output_layer(computed_third_conn_with_activation)
        return computed_outputs
```

- Constructing the neural network

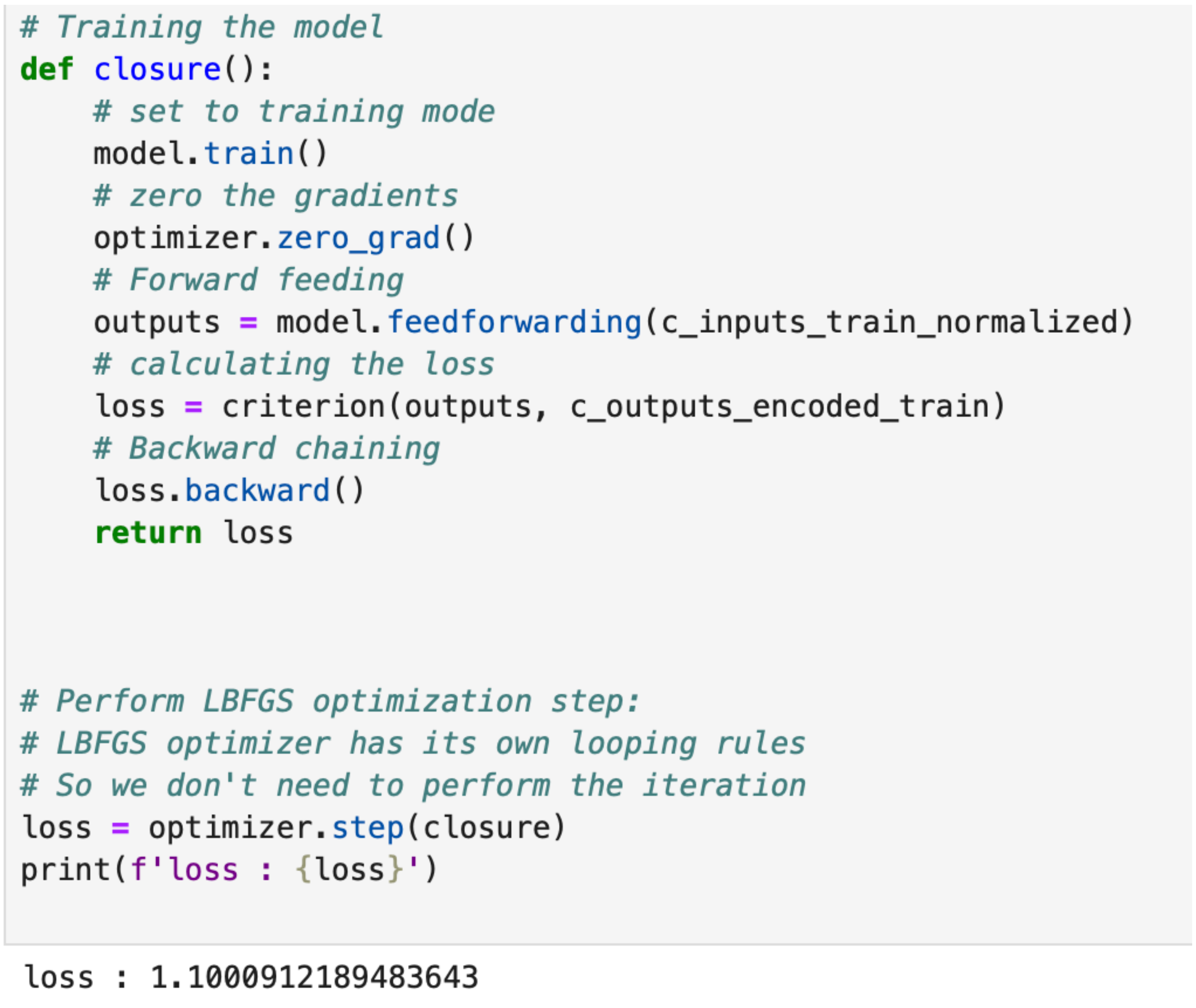
```
# For this model, initialing
# inputs size will number of features
# Three hidden layer with 14,7,3 nodes
# output_size wii be one
# For evaluation purpose with other model this values are used
input_size = c_inputs_train_normalized.shape[1] # counting the number of features
hidden_layer1 = 14
hidden_layer2 = 7
hidden_layer3 = 3
output_size = 3
model = MLP(input_size, hidden_layer1, hidden_layer2, hidden_layer3, output_size)
```

- Defining the loss function and optimizer with built in method .CrossEntropyLoss() and .optim respectively.



```
# Defining the loss function
criterion = nn.CrossEntropyLoss()
# Defining the optimizer: LBFGS Ooptimizer is used
optimizer = optim.LBFGS(model.parameters(), lr= 0.01)
```

- Training the model with training datasets



```
# Training the model
def closure():
    # set to training mode
    model.train()
    # zero the gradients
    optimizer.zero_grad()
    # Forward feeding
    outputs = model.feedforwarding(c_inputs_train_normalized)
    # calculating the loss
    loss = criterion(outputs, c_outputs_encoded_train)
    # Backward chaining
    loss.backward()
    return loss



# Perform LBFGS optimization step:
# LBFGS optimizer has its own looping rules
# So we don't need to perform the iteration
loss = optimizer.step(closure)
print(f'loss : {loss}')
```

```
loss : 1.1000912189483643
```

- Predictions are generated using the test datasets, after which we will evaluate and determine the model's accuracy.

```
# Testing the model with test dataset
# set to testing mode
model.eval()
with torch.no_grad():
    outputs = model.feedforwarding(c_inputs_test_normalized)
    _, predicted = torch.max(outputs,1)
    accuracy = (predicted == c_outputs_encoded_test).sum().item() / c_outputs_encoded_test.size
    print(f'Accuracy on test data: {accuracy:.4f}')
```

```
Accuracy on test data: 0.2333
```

Drawing conclusion:

The model's accuracy is 23% and loss is 1.1.

3) MLP built using NumPy (custom implementation)

This method is developed using NumPy functions and methods. All the mathematical operations and processes are implemented entirely using NumPy.

- Defining the activation function. Sigmoid is the activation function that we are implementing. The mathematic notation is as follows:

Sigmoid Function:

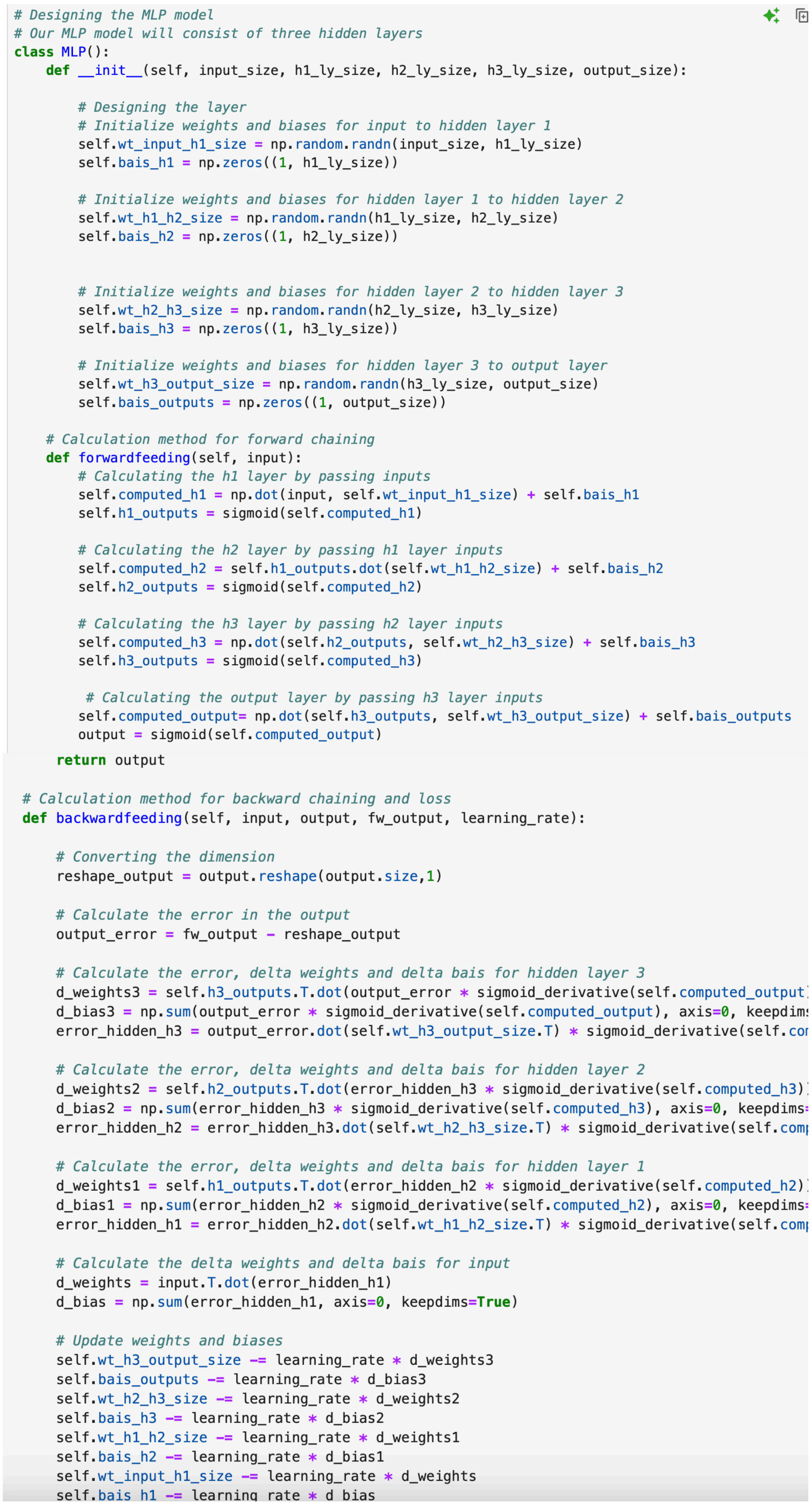
$$S(x) \doteq \frac{1}{1+e^{-x}}$$

Sigmoid Derivative:

$$S(x) * (1 - S(x))$$

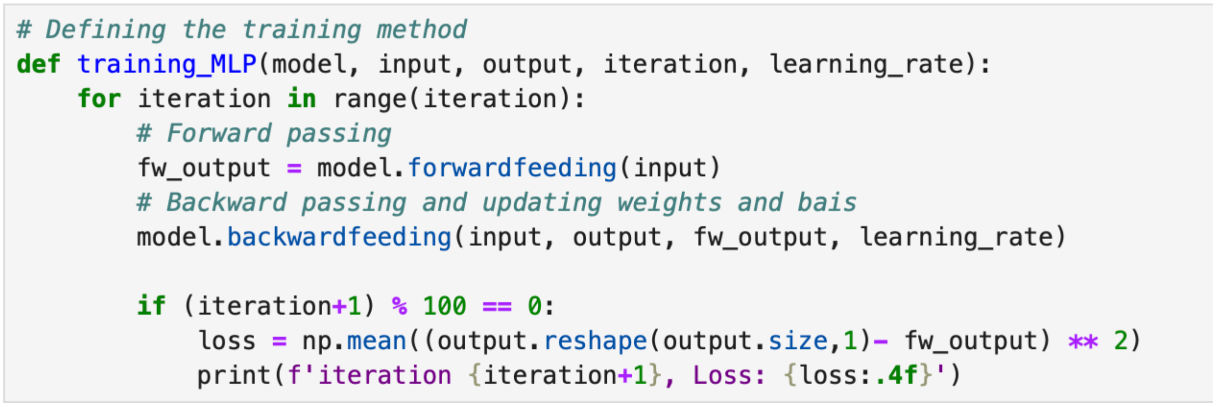
```
# Define the Activation function: we are using Sigmoid

# Define the Sigmoid activation function
def sigmoid(inputs):
    inputs = np.clip(inputs, -500, 500)  # Clipping the input range
    return 1 / (1 + np.exp(-inputs))

# Define the derivative of Sigmoid for backpropagation
def sigmoid_derivative(inputs):
    inputs = np.clip(inputs, -500, 500)  # Clipping the input range
    return inputs * (1 - inputs)
```

- Designing the MLP model.



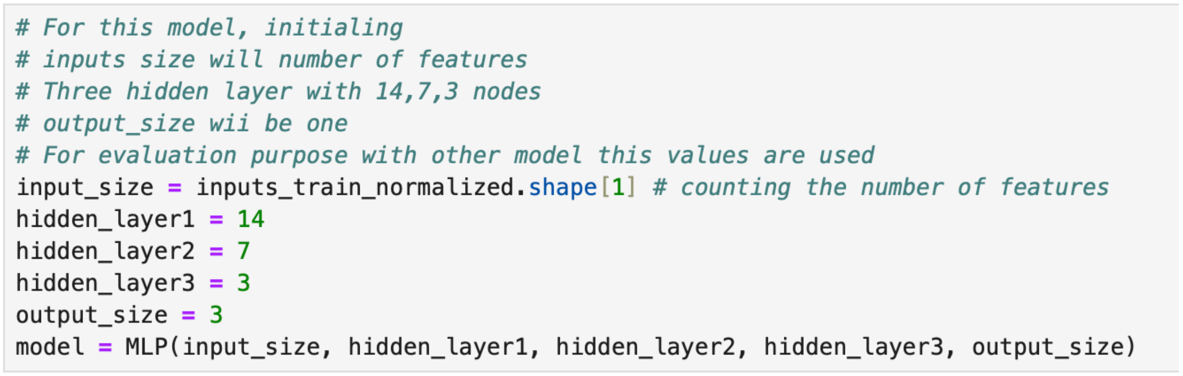
```python
# Designing the MLP model
# Our MLP model will consist of three hidden layers
class MLP():
    def __init__(self, input_size, h1_ly_size, h2_ly_size, h3_ly_size, output_size):

        # Designing the layer
        # Initialize weights and biases for input to hidden layer 1
        self.wt_input_h1_size = np.random.randn(input_size, h1_ly_size)
        self.bais_h1 = np.zeros((1, h1_ly_size))

        # Initialize weights and biases for hidden layer 1 to hidden layer 2
        self.wt_h1_h2_size = np.random.randn(h1_ly_size, h2_ly_size)
        self.bais_h2 = np.zeros((1, h2_ly_size))


        # Initialize weights and biases for hidden layer 2 to hidden layer 3
        self.wt_h2_h3_size = np.random.randn(h2_ly_size, h3_ly_size)
        self.bais_h3 = np.zeros((1, h3_ly_size))

        # Initialize weights and biases for hidden layer 3 to output layer
        self.wt_h3_output_size = np.random.randn(h3_ly_size, output_size)
        self.bais_outputs = np.zeros((1, output_size))

    # Calculation method for forward chaining
    def forwardfeeding(self, input):
        # Calculating the h1 layer by passing inputs
        self.computed_h1 = np.dot(input, self.wt_input_h1_size) + self.bais_h1
        self.h1_outputs = sigmoid(self.computed_h1)

        # Calculating the h2 layer by passing h1 layer inputs
        self.computed_h2 = self.h1_outputs.dot(self.wt_h1_h2_size) + self.bais_h2
        self.h2_outputs = sigmoid(self.computed_h2)

        # Calculating the h3 layer by passing h2 layer inputs
        self.computed_h3 = np.dot(self.h2_outputs, self.wt_h2_h3_size) + self.bais_h3
        self.h3_outputs = sigmoid(self.computed_h3)

         # Calculating the output layer by passing h3 layer inputs
        self.computed_output= np.dot(self.h3_outputs, self.wt_h3_output_size) + self.bais_outputs
        output = sigmoid(self.computed_output)
        return output

    # Calculation method for backward chaining and loss
    def backwardfeeding(self, input, output, fw_output, learning_rate):

        # Converting the dimension
        reshape_output = output.reshape(output.size,1)

        # Calculate the error in the output
        output_error = fw_output - reshape_output

        # Calculate the error, delta weights and delta bais for hidden layer 3
        d_weights3 = self.h3_outputs.T.dot(output_error * sigmoid_derivative(self.computed_output
        d_bias3 = np.sum(output_error * sigmoid_derivative(self.computed_output), axis=0, keepdims
        error_hidden_h3 = output_error.dot(self.wt_h3_output_size.T) * sigmoid_derivative(self.con

        # Calculate the error, delta weights and delta bais for hidden layer 2
        d_weights2 = self.h2_outputs.T.dot(error_hidden_h3 * sigmoid_derivative(self.computed_h3))
        d_bias2 = np.sum(error_hidden_h3 * sigmoid_derivative(self.computed_h3), axis=0, keepdims
        error_hidden_h2 = error_hidden_h3.dot(self.wt_h2_h3_size.T) * sigmoid_derivative(self.comp

        # Calculate the error, delta weights and delta bais for hidden layer 1
        d_weights1 = self.h1_outputs.T.dot(error_hidden_h2 * sigmoid_derivative(self.computed_h2))
        d_bias1 = np.sum(error_hidden_h2 * sigmoid_derivative(self.computed_h2), axis=0, keepdims
        error_hidden_h1 = error_hidden_h2.dot(self.wt_h1_h2_size.T) * sigmoid_derivative(self.comp

        # Calculate the delta weights and delta bais for input
        d_weights = input.T.dot(error_hidden_h1)
        d_bias = np.sum(error_hidden_h1, axis=0, keepdims=True)

        # Update weights and biases
        self.wt_h3_output_size -= learning_rate * d_weights3
        self.bais_outputs -= learning_rate * d_bias3
        self.wt_h2_h3_size -= learning_rate * d_weights2
        self.bais_h3 -= learning_rate * d_bias2
        self.wt_h1_h2_size -= learning_rate * d_weights1
        self.bais_h2 -= learning_rate * d_bias1
        self.wt_input_h1_size -= learning_rate * d_weights
        self.bais_h1 -= learning_rate * d_bias
```

- Defining the training method



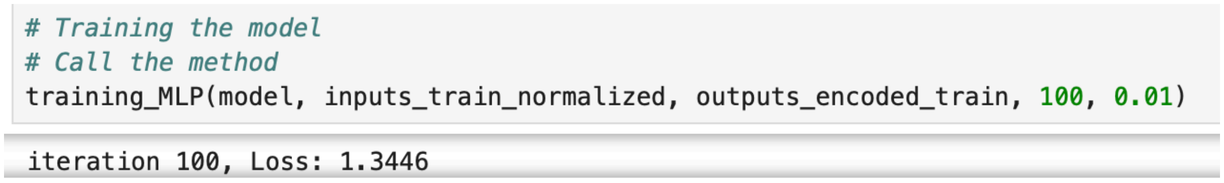
```
# Defining the training method
def training_MLP(model, input, output, iteration, learning_rate):
    for iteration in range(iteration):
        # Forward passing
        fw_output = model.forwardfeeding(input)
        # Backward passing and updating weights and bais
        model.backwardfeeding(input, output, fw_output, learning_rate)

        if (iteration+1) % 100 == 0:
            loss = np.mean((output.reshape(output.size,1)- fw_output) ** 2)
            print(f'iteration {iteration+1}, Loss: {loss:.4f}')
```

- Initialing the all the values for the neural network.



```
# For this model, initialing
# inputs size will number of features
# Three hidden layer with 14,7,3 nodes
# output_size wii be one
# For evaluation purpose with other model this values are used
input_size = inputs_train_normalized.shape[1] # counting the number of features
hidden_layer1 = 14
hidden_layer2 = 7
hidden_layer3 = 3
output_size = 3
model = MLP(input_size, hidden_layer1, hidden_layer2, hidden_layer3, output_size)
```

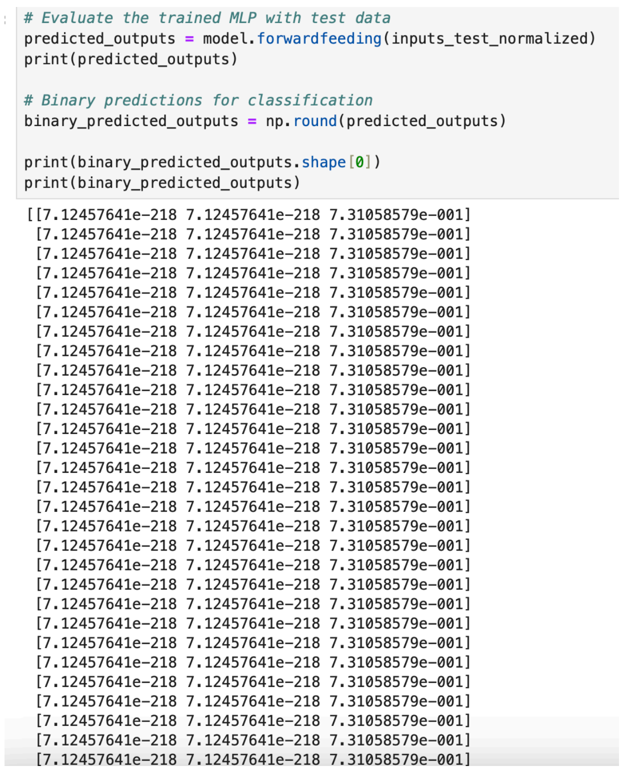
- Training the model by calling the train method and passing the arguments



```
# Training the model
# Call the method
training_MLP(model, inputs_train_normalized, outputs_encoded_train, 100, 0.01)
```

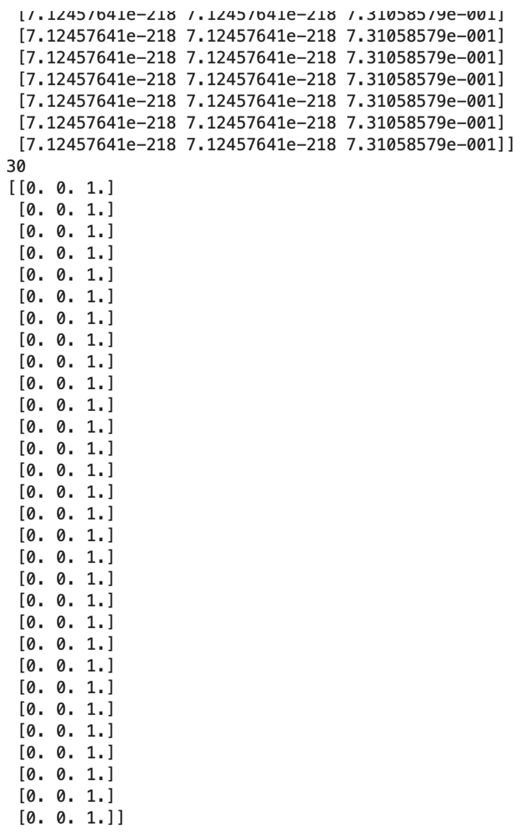
```
iteration 100, Loss: 1.3446
```

- Testing the model with test dataset (prediction) and converting into binary. It needs to be converted into binary for evaluation of the model.



```
# Evaluate the trained MLP with test data
predicted_outputs = model.forwardfeeding(inputs_test_normalized)
print(predicted_outputs)

# Binary predictions for classification
binary_predicted_outputs = np.round(predicted_outputs)

print(binary_predicted_outputs.shape[0])
print(binary_predicted_outputs)
```

```
[[7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
```



```
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]
 [7.12457641e-218 7.12457641e-218 7.31058579e-001]]
30
[[0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]
 [0. 0. 1.]]
```

- Checking the accuracy of the model.

```
# Evaluating the model
# Compute accuracy (percentage of correct predictions)
accuracy = np.mean(binary_predicted_outputs == outputs_encoded_test.reshape(outputs_encoded_tes
print(f"Accuracy: {accuracy:.2f}")
```

```
Accuracy: 0.30
```

Drawing the conclusion:
The model's accuracy is 30% and loss is 1.3.

Comparing the three models

While designing all three models, we used the same architecture: three hidden layers. The first hidden layer contains 14 nodes, the second layer has 7 nodes, and the third layer has 3 nodes. In both the MLPClassifier and PyTorch models, the ReLU activation function was applied, while in the NumPy model, the sigmoid activation function was used.

An activation function is a mathematical function that introduces non-linearity into the model, enabling it to learn complex patterns by capturing non-linear relationships. Using an activation function is essential in every neural network for better performance.

Another key aspect is the optimizer algorithm, which modifies the weights and biases of the neural network, improving predictions. In both the MLPClassifier and PyTorch models, we used the LBFGS (Limited-memory Broyden–Fletcher–Goldfarb–Shanno) optimizer. However, in the NumPy model, no built-in optimizer was used since the steps were manually implemented. The learning rate for all models was set to 0.01.

After making predictions, the accuracy of each model was evaluated as follows:
- MLPClassifier: 93%
- PyTorch: 23%
- NumPy: 30%

This shows that the MLPClassifier outperformed the other models.

Most of the arguments are similar across the models, but the MLPClassifier predicted more accurately because it is a built-in function where all methods are pre-defined. You only need to call the function with the appropriate arguments for prediction. In contrast, while PyTorch is also a built-in library, it requires the definition of certain methods, with other elements left to the user to implement. In NumPy, everything is manually coded, with no pre-built methods.

The lower accuracy in PyTorch and NumPy could be due to the higher number of hidden layers for a dataset with lower computational complexity, potentially leading to overfitting. However, in the MLPClassifier, the accuracy remains high despite using the same hidden layers, possibly because the built-in function incorporates control mechanisms to prevent overfitting.