CSCI 5592 | Deep Learning and Neural Nets | Lab 01 Report

Submitted By: Anusha Basavaraja
(anba9017@colorado.edu)

Part 1 of Lab 01:

Methods:

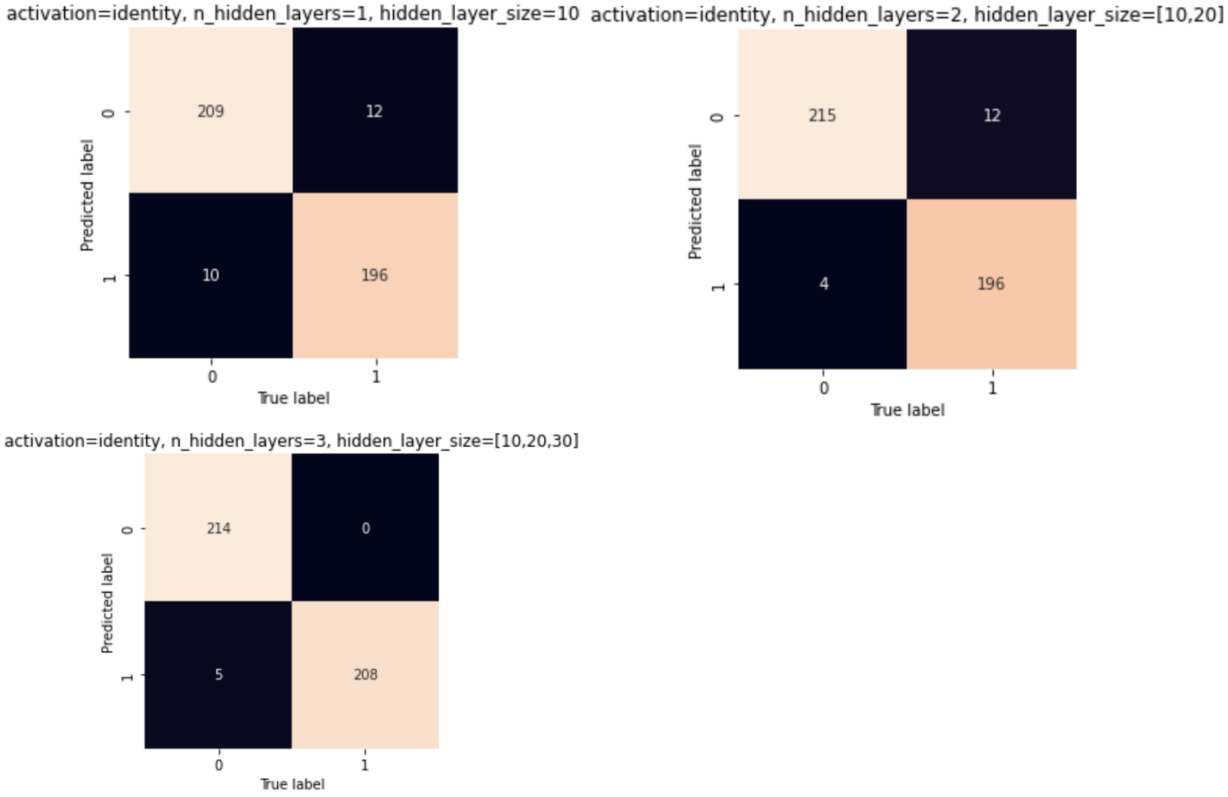
I am using a medical dataset from Kaggle called Anemia Dataset available at this link (https://www.kaggle.com/datasets/biswaranjanrao/anemia-dataset). This is a simple dataset consisting of 1421 instances in it. It has a total of 6 attributes – Gender, Hemoglobin, MCH (mean corpuscular hemoglobin), MCHC (mean corpuscular hemoglobin concentration), MCV (mean corpuscular volume) and Result ('0' represents 'not anemic' and '1' represents 'anemic'). It has 801 instances with Result = 0 (not anemic) and 620 instances with Result = 1(anemic). So, it looks like the data is not very much imbalanced considering the binary classification scenario.

I am using 4 different activation functions (logistic/sigmoid, tanh, relu and identity) available in scikit learn package with varying no. of hidden layers (1/2/3) which results in different combinations. I have kept the following hyperparameters/parameters constant which includes: learning rate = 0.001 (default value available in the sklearn), no. of epochs = 25 (since the size of the dataset is really less, I used less no. of iterations as keeping it too big will not allow me to learn/understand the change in the patterns thereby always resulting in good accuracy scores), no. of neurons per hidden layer – at hidden layer 1 = 10, at hidden layer 2 = 20, at hidden layer 3 = 30 (using less no. of neurons at each stage because of the small size of the data set). I haven't explicitly altered/changed any other values apart from the above-mentioned ones as I am using rest of the default hyperparameters/parameters in my experiment from the package.

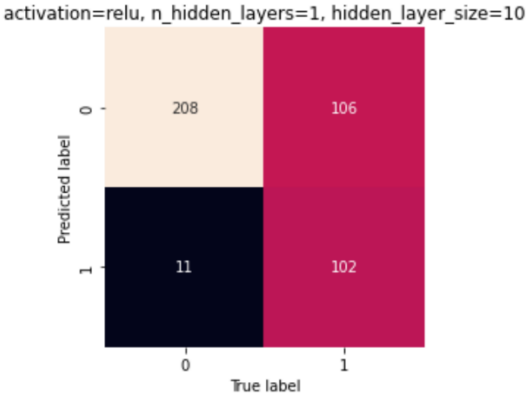
Results:

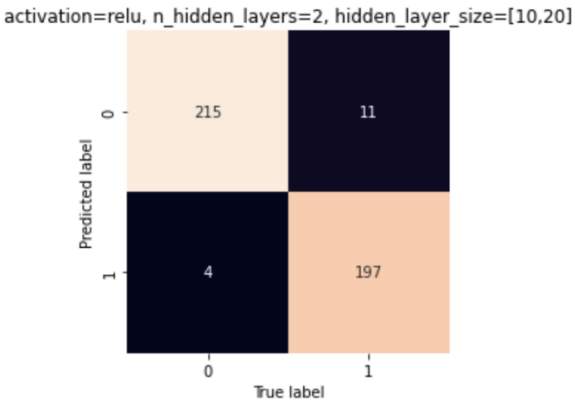
Model No.	*Activation function*	*No. of Hidden Layers*	*No. of neurons per Hidden Layer*	*Model Accuracy*	*Model Precision*	*Model Recall*	*Model F1-score*
1.	Logistic/ sigmoid	1	10	51.29%	0.26	0.51	0.35
2.		2	10,20	51.29%	0.26	0.51	0.35
3.		3	10,20,30	51.29%	0.26	0.51	0.35
4.	tanh	1	10	73.77%	0.79	0.74	0.72
5.		2	10,20	94.61%	0.91	0.99	0.95
6.		3	10,20,30	98.13%	0.98	0.98	0.98
7.	relu	1	10	72.60%	0.78	0.73	0.71
8.		2	10,20	96.49%	0.97	0.96	0.96
9.		3	10,20,30	97.89%	0.98	0.98	0.98
10.	identity	1	10	94.85%	0.95	0.95	0.95
11.		2	10,20	96.25%	0.96	0.96	0.96
12.		3	10,20,30	98.83%	0.99	0.99	0.99

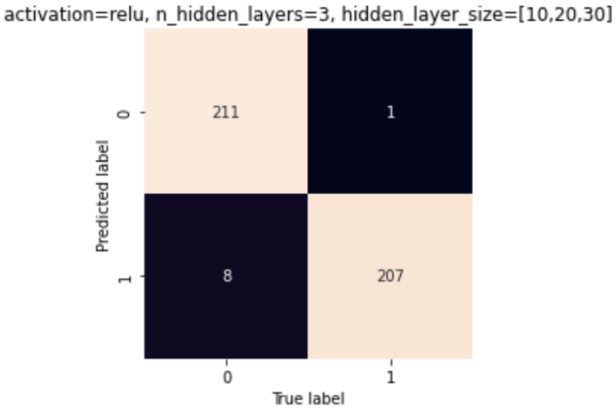
Confusion matrix – from 'Identity' activation function with 1, 2, 3 hidden layers:



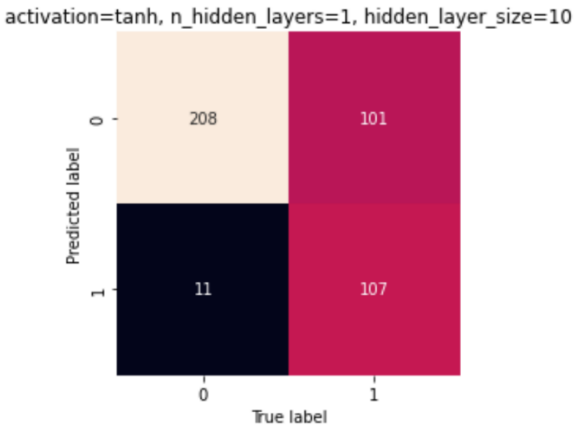
Confusion matrix – from 'ReLU' activation function with 1, 2, 3 hidden layers:

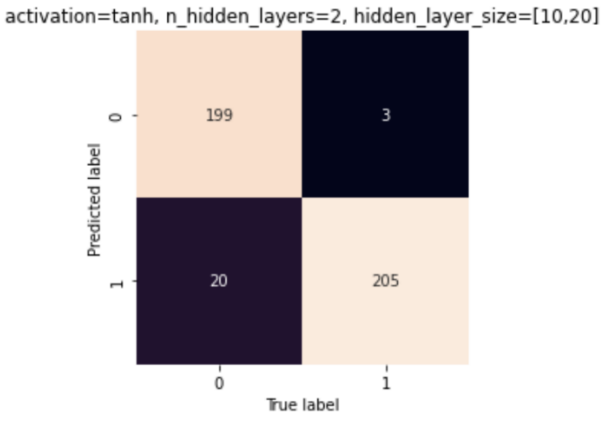




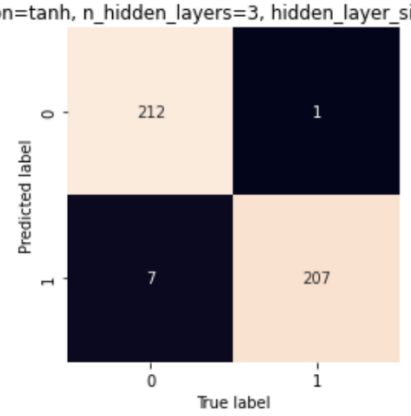


Confusion matrix – from 'tanh' activation function with 1, 2, 3 hidden layers:



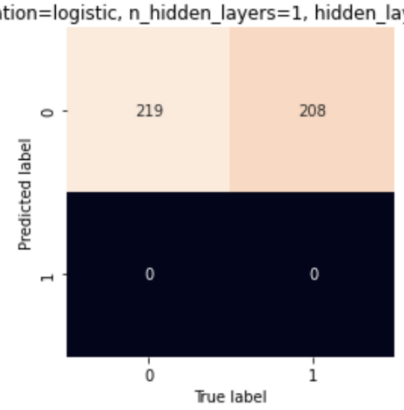


activation=tanh, n_hidden_layers=3, hidden_layer_size=[10,20,30]

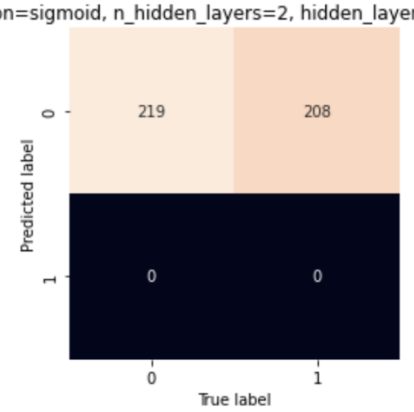


<u>Confusion matrix – from 'logistic/sigmoid' activation function with 1, 2, 3 hidden layers</u>:

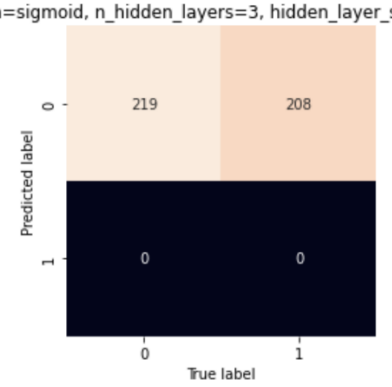
activation=logistic, n_hidden_layers=1, hidden_layer_size=10



activation=sigmoid, n_hidden_layers=2, hidden_layer_size=[10,20]



activation=sigmoid, n_hidden_layers=3, hidden_layer_size=[10,20,30]



Analysis:

1. Did a certain number of hidden layers lead to consistently better results?

Considering the various evaluation metrics observed from the results during the execution of different models, looks like the experiments with most no. of hidden layers performed really well compared to the ones with least no. of hidden layers. This is because it looks like the dataset required comparatively complex model/s than the simplest ones to understand the patterns within the dataset to learn for the classification task.

2. Did a certain type of activation function lead to consistently better results?

Yes, the 'Identity' function outdid every other activation function used thereby resulting in better results when comparing the respective evaluation metrices. This might be because 'Identity' function is a simple linear function. It looks like the dataset used might require simple linear transformation in classifying the instances more accurately compared to the more complex non-linear transformations.

Considering the fact that NN is about using the non-linear activation functions, then I should mention that 'tanh' and 'ReLU both performed really well when compared to 'Sigmoid/Logistic' function (see the 1st table in the Results section above). Just to make sure that there is improvement in the model performance using 'Sigmoid' function, I conducted additional experiments with additional number of neurons per layer. Then the models with sigmoid activation function and substantially greater number of neurons in each of the hidden layers performed well showing the accuracy up to 99%. Here is a table that depicts the improvement in the performance of the model using 'sigmoid' activation function with variable no. of neurons per layer.

Model No.	*Activation function*	*No. of Hidden Layers*	*No. of neurons per Hidden Layer*	*Model Accuracy*	*Model Precision*	*Model Recall*	*Model F1-score*
1.	sigmoid/ logistic	1	100	77.52%	0.84	0.78	0.76
2.		1	250	92.97%	0.97	0.93	0.93
3.		2	50, 50	90.16%	0.91	0.90	0.90
4.		2	100, 100	92.74%	0.93	0.93	0.93
5.		2	250, 250	98.83%	0.99	0.99	0.99

Looking at this table, it seems like the use of sigmoid activation function also results in better performance if and only if we use more number of neurons per layer

and by adding additional hidden layers. This is because, when we use 'tanh' or 'relu', the model does not require more complexity in understanding the patterns in the dataset to learn whereas when we use 'sigmoid' function, it looks like the model is not complex enough thereby resulting in underfitting issue.

3. *What insights are gained by looking at the different evaluation metrics?*

The evaluation metrics allows us to understand the overall model performance. From the 1st result table, we can notice that apart from 'sigmoid', rest of all the models have good accuracy, precision, recall and f1-scores as we add more hidden layers which means that these metrices are directly proportional to the correctly identified instances from the test set. This becomes clear when we look at the confusion matrix. The models with good evaluation metrices have greater no. of True Positives and True Negatives compared to False Positives and False Negatives.

Bonus Analysis:

1) From my experiment, it is clear that the no. of neurons per hidden layer and no. of hidden layers assigned for a model plays very important role in the model performance.
- Higher the no. of neurons per layer, higher will be the accuracy and even though we add more no. of layers, there will not be much substantial improvement observed in the model performance.
- If we set fewer neurons per layer, then we need more hidden layers to observe substantial/greater improvement in the model performance.

2) When experimenting with the no. of neurons and no. of hidden layers in a model, it is important to find that threshold range which does not lead to underfitting or overfitting of the model as too less leads to underfitting (not complex enough to learn) and too much leads to overfitting (too complex to learn).

Part 2 of Lab 01:

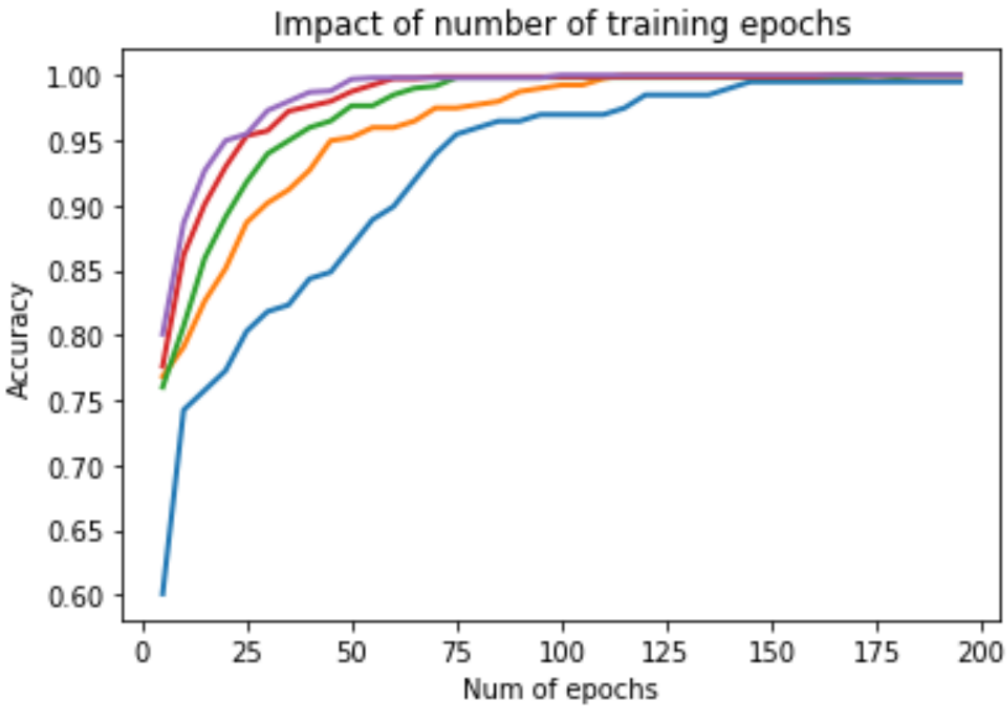
Methods:

I am using the same medical dataset from Kaggle called Anemia Dataset from part 1 available at this link (https://www.kaggle.com/datasets/biswaranjanrao/anemia-dataset). This is a simple dataset consisting of 1421 instances in it. It has a total of 6 attributes – Gender, Hemoglobin, MCH (mean corpuscular hemoglobin), MCHC (mean corpuscular

hemoglobin concentration), MCV (mean corpuscular volume) and Result ('0' represents 'not anemic' and '1' represents 'anemic'). It has 801 instances with Result = 0 (not anemic) and 620 instances with Result = 1(anemic). So, it looks like the data is not very much imbalanced considering the binary classification scenario.

The original dataset has 70%-30% train-test split. Out of 70% training dataset, I am further considering 20%, 40%, 60%, 80% and 100% of training data for experimenting with size of training dataset as suggested. During all these 5 scenarios/approaches, I am keeping the values of no. of hidden layers (3), no. of neurons per layer (10, 25, 50) and activation function ('tanh') constant. Rest of the hyperparameters/parameters used are the same default values from the sklearn package. I am starting the model learning form no. of epochs = 5 to 200 with the gradual increments of 5 epochs and plotting the resultant learning curves (No. of epochs vs Accuracy) to understand the impact of training data size and duration on the model performance.

Results:



Analysis:

From the graph, we can observe that, the training duration (no. of epochs) and the size of the training dataset are very important factors in understanding the model performance. When we use less amount of training data (20% of original 70% training data in blue color), the model needs to trained for really longer times to achieve better model results (20% - it almost took about 150 epochs to give good consistent accurate results). On the other hand, when we consider more amount of training data (100% of original training data in purple color), the model requires very less training to achieve better results (100% - it just took about 50 epochs to give good consistent accurate results). Similarly, 40% of original training data (yellow) took about 115 epochs, 60% of original training data (green) took about 75 epochs and 80% of the original training data (red) took about 60 epochs to achieve consistent accurate results whose accuracy is almost close to 1.00 or 100% after those no. of epochs or training duration.

1) *What is the influence of the amount of training data?*

The size/amount of the training data plays substantially important role on the model performance i.e., the size of training data is directly proportional to the model performance (more the size of training data, greater will be the model performance) and inversely proportional to the training duration or no. of epoch required for better performance of the model (more the size of training data, less no. of epochs required to achieve good results).

2) *What is the influence of the training duration?*

If we keep the results/expected model performance constant, the training duration is inversely proportional to the amount of training data used (i.e., less no. of epochs required if we use more amount of training data and more no. of epochs required if we use very less amount to training data to achieve good results). Considering the other way around, keeping the size of training data and values of hyperparameters constant, then the training duration is directly proportional to the model performance (i.e., we need more no. of epochs to minimize the training loss to achieve better results).

Part 1 of Lab 01:

```
import pandas as pd

data = pd.read_csv('anemia.csv')
len_data = len(data)
len_data
```

```
data.head(10)
```

```
# Differentiating the features from the target column
# X -> data input to the model, y -> target/label

X = data.iloc[ : , 0:5]
y = data.iloc[ : , 5:6]
```

```
X.head(10)
```

```
y.head(10)
```

```
# Create the train/test split and preprocess
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state=1)

# Standardize data
from sklearn.preprocessing import StandardScaler
stdsc = StandardScaler()
stdsc.fit(X_train)
X_train_std = stdsc.transform(X_train)
X_test_std = stdsc.transform(X_test)
```

```
# Setting hyperparameters to Train NN model -> Multi Layer Perceptron (MLP)
# Change the hyperparameters to experiment with the model

n_neurons_10 = 10
n_neurons_20 = 20
n_neurons_30= 30
n_neurons_100 = 100
n_neurons_50 = 50

# Activation functions -> 'logistic', 'tanh', 'relu', 'identity

activation_function_1 = 'logistic'
activation_function_2 = 'tanh'
activation_function_3 = 'relu'
```

```
activation_function_4 = 'identity'
```

```
# Training and Testing the NN model  -> Multi Layer Perceptron (MLP)

from sklearn.neural_network import MLPClassifier
from sklearn import metrics

# Choose the hyper parameters to experiment with different models
# Activations functions -> 1. logitic/sigmoid, 2. tanh, 3. relu, 4. identity
# Number of hidden Layers -> 1/2/3
# No. of neurons per layer = 10/20/30/50/100

mlp = MLPClassifier(activation=activation_function_4, hidden_layer_sizes=(n_neurons_100, n_ne
mlp.fit(X_train_std, y_train)

print('Activation function: {}'.format(mlp.activation))
print("Accuracy score on the test set: {}".format(mlp.score(X_test_std, y_test)))

y_pred = mlp.predict(X_test_std)
print(metrics.classification_report(y_test, y_pred))
```

```
#Confusion Matrix

import seaborn as sns
import matplotlib.pyplot as plt

mat = metrics.confusion_matrix(y_test, y_pred)
plt.title('activation=identity, n_hidden_layers=2, hidden_layer_size=[100,100]')
sns.heatmap(mat.T, square = True, annot = True, fmt = 'd', cbar = False)
plt.xlabel("True label")
plt.ylabel("Predicted label")
```

Colab paid products - Cancel contracts here

✓ 0s completed at 12:44 PM

Part 2 of Lab 01:

```
import pandas as pd

data = pd.read_csv('anemia.csv')
len_data = len(data)
print("Total instances in original training dataset:", len_data)
X = data.iloc[ : , 0:5]
y = data.iloc[ : , 5:6]
```

```
Total instances in original training dataset: 1421
```

```
# Create the train/test split and preprocess
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,
                                                    random_state=1)

# Standardize data
from sklearn.preprocessing import StandardScaler
stdsc = StandardScaler()
stdsc.fit(X_train)
X_train_std = stdsc.transform(X_train)
X_test_std = stdsc.transform(X_test)


# Considering only 20%, 40%, 60%, 80% and 100% from original 70% of
#training data for experimenting with the models.

X_train_std_20 = X_train_std[0 : int(0.20 * len(X_train_std))]
y_train_20 = y_train[0 : int(0.20 * len(X_train_std))]

X_train_std_40 = X_train_std[0 : int(0.40 * len(X_train_std))]
y_train_40 = y_train[0 : int(0.40 * len(X_train_std))]

X_train_std_60 = X_train_std[0 : int(0.60 * len(X_train_std))]
y_train_60 = y_train[0 : int(0.60 * len(X_train_std))]

X_train_std_80 = X_train_std[0 : int(0.80 * len(X_train_std))]
y_train_80 = y_train[0 : int(0.80 * len(X_train_std))]


# Prining out the length of training data sizes
'''
print("len_20% of training: {} and test: {}". format(len(X_train_std_20), len(y_train_20)))
print("len_40% of training: {} and test: {}". format(len(X_train_std_40), len(y_train_40)))
print("len_60% of training: {} and test: {}". format(len(X_train_std_60), len(y_train_60)))
print("len_80% of training: {} and test: {}". format(len(X_train_std_80), len(y_train_80)))
print("len_100% of training: {} and test: {}". format(len(X_train_std), len(y_train)))
'''
```

```
# Import all packages required
from sklearn.neural_network import MLPClassifier
from sklearn import metrics
import seaborn as sns
import matplotlib.pyplot as plt


n_start_itr = 5
n_end_itr = 200
n_itr_inc = 5

def train_size_expt(X_train_data, y_train_data, size_training_data):
  epoch_counts = []
  accuracy_observations = []
  for epochs in range(n_start_itr, n_end_itr, n_itr_inc):
    mlp = MLPClassifier(activation='tanh', hidden_layer_sizes=(10, 25, 50),
                        max_iter=epochs, random_state=1)
    mlp.fit(X_train_data, y_train_data)
    accuracy_score = mlp.score(X_train_data, y_train_data)
    accuracy_observations.append(accuracy_score)
    epoch_counts.append(epochs)

  return accuracy_observations, epoch_counts, size_training_data


result = []
epoch_steps = []

# Using 20% of original 70% training data
result_20 = train_size_expt(X_train_std_20, y_train_20, 20)
result.append(result_20[0])
epoch_steps.append(result_20[1])

# Using 40% of original 70% training data
result_40 = train_size_expt(X_train_std_40, y_train_40, 40)
result.append(result_40[0])
epoch_steps.append(result_40[1])

# Using 60% of original 70% training data
result_60 = train_size_expt(X_train_std_60, y_train_60, 60)
result.append(result_60[0])
epoch_steps.append(result_60[1])

# Using 80% of original 70% training data
result_80 = train_size_expt(X_train_std_80, y_train_80, 80)
result.append(result_80[0])
epoch_steps.append(result_80[1])

# Using 100% of original 70% training data
result_100 = train_size_expt(X_train_std, y_train, 100)
result.append(result_100[0])
```

```
    epoch_steps.append(result_100[1])

    plt.xlabel("Num of epochs")
    plt.ylabel("Accuracy")
    plt.title("Impact of number of training epochs")
    for i in range(0, len(result)):
      plt.plot(epoch_steps[i], result[i], linewidth = 2)

    plt.show()
```


✓ 2m 5s completed at 11:09 AM