Phishing Email Detection Report

Creating the Model

My first step in creating my MLP model was importing the necessary modules. Pytorch (torch in the imports) will be used in order to create the Multilayer Perceptron. We also must import “torch.nn” which is the neural network portion of pytorch and allows us to use neural network components. The “torch.optim” import gives us tools that we can use to help optimize the neural network for computational efficiency. Import torch.nn.functional is used to create activation functions, and torch.utils.data is necessary to make the data usable for our tensors. Lastly, we import the same sklearn modules from Assignment 2 in order to create the Naive Bayes model.

mlp_483new.py > ...

```python
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score, precision_score
from sklearn.naive_bayes import MultinomialNB
```

The next step is preprocessing the data. I used pandas to import the “emails.csv” dataset and then TfidfVectorizer to vectorize the data into numerical data. This is necessary because neural networks and naive bayes require numerical data. I chose to use TfidfVectorizer because it limits the impact of words that have little meaning (i.e. the, or, etc.). The max_features is limited to the 1000 most important terms to improve computational efficiency. I then split the data into a training and test set with an 80/20 split.

```python
#Load and preprocess dataset
import pandas as pd

#Load dataset
emails = pd.read_csv('emails.csv')
#emails = emails.sample(frac=0.01, random_state=42)

#Vectorize text data
vectorizer = TfidfVectorizer(max_features=1000)
X = vectorizer.fit_transform(emails['body']).toarray()
y = emails['label'].values

#Create train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
```

Because pytorch models require tensor inputs, I had to convert the training and test sets to tensors, with features being assigned the dtype of float32 and labels being assigned the dtype of long. I then used TensorDataset to wrap the related tensors together, this reduces indexing errors and allows us to retrieve the feature and label simultaneously, increasing the program's efficiency. By using TensorDataset I can also then use the wrapped tensors as input for DataLoader(). DataLoader() automatically batches and shuffles data, making it as optimized as possible to be used with the Minibatch Gradient Descent training method later in the program. Lastly, I implemented the MLP model. The _init_ function creates 2 layers, the hidden and output layer. The hidden layer contains 128 neurons, and the 1000 vectorized features would be mapped to these neurons for predictions. The output layer contains 2 neurons, this is for binary classification (i.e spam or not). The forward function is a necessary part of any pytorch program as it defines how data will flow through the neural network. Using F to create the activation function and ReLU as the activation function to be used, we then pass data through the hidden and output layers. I chose ReLU as the activation function because it excels at introducing non-linearity so that functions such as spam classification are both accurate and efficient in neural networks.

```
#Convert train and test sets to tensors so they can be used with MLP
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)

#Create DataLoader objects
batch_size = 64
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

#Implement MLP model
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

My next step was creating the functions to train and test the model. I used minibatch gradient descent as my training method because of it's efficiency. Within the training function, zero_grad() erases previously calculated gradients, it then propagates backwards to calculate new gradients using loss.backward(), lastly, optimizer.step() updates model parameters using the adam optimizer. I chose the adam optimizer because it works well with data that has features that vary in how much they impact predictions, such as the spam email dataset. The training function then prints the Epoch and Loss which can be seen in the result image below. The test function is set in eval mode by using model.eval(), this ensures consistent results. I then created two arrays for the true and predicted labels. The function then loops through the test dataset within the predefined minibatches, the data contains the features of the current batch while the target contains the true labels for the current batch. The torch.no_grad() function is used to make sure pytorch doesn't take up any computational space by calculating gradients or updating parameters. The data is then fed to the model, torch.max() finds the class that a feature has the highest probability of landing in. Then the calculated predictions and true labels are appended to the aforementioned arrays.

```python
#Create function to train MLP model
def train_model(model, loader, criterion, optimizer, num_epochs):
    model.train()
    for epoch in range(num_epochs):
        for batch_idx, (data, target) in enumerate(loader):
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item()}")

#Create function to test MLP model
def test_model(model, loader):
    model.eval()
    y_true = []
    y_pred = []
    with torch.no_grad():
        for data, target in loader:
            output = model(data)
            _, preds = torch.max(output, 1)
            y_true.extend(target.numpy())
            y_pred.extend(preds.numpy())
    return y_true, y_pred
```

The next step in the classification process is defining the hyperparameters and then running the data through the model. Input_size was set to 1000 so it matches the max_features parameter set earlier. I set hidden_size to 128 as it was a good middle ground between computational efficiency and accuracy. Only two classes are needed as the data is classified as either spam or not spam. I set the learning_rate to 0.001 because it is the default for the Adam optimizer. Lastly, I set num_epochs to 10 because it seemed as if increasing or decreasing this number either resulted in underfitting or decreased efficiency. As for the criterion, this is what is used to calculate the error between true and predicted labels, CrossEntropyLoss() was designed for classification which led me to choose it. I then train and test the MLP model using the data and calculate the appropriate metrics.

```
#Define Hyperparameters
input_size = 1000
hidden_size = 128
num_classes = 2
learning_rate = 0.001
num_epochs = 10

mlp_model = MLP(input_size, hidden_size, num_classes)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(mlp_model.parameters(), lr=learning_rate)

#Train MLP model
print("Training MLP...")
train_model(mlp_model, train_loader, criterion, optimizer, num_epochs)

#Test MLP model
print("Testing MLP...")
y_true_mlp, y_pred_mlp = test_model(mlp_model, test_loader)

#Evaluate MLP model
mlp_conf_matrix = confusion_matrix(y_true_mlp, y_pred_mlp)
mlp_accuracy = accuracy_score(y_true_mlp, y_pred_mlp)
mlp_precision = precision_score(y_true_mlp, y_pred_mlp)

print("MLP Confusion Matrix:\n", mlp_conf_matrix)
print(f"MLP Accuracy: {mlp_accuracy}")
print(f"MLP Precision: {mlp_precision}")
```

The last piece of my classification program was creating the NaiveBayes classifier. I then calculated the metrics for this classifier and compared them to the MLP.

```
#Create and train Naive Bayes Model
print("Training Naive Bayes...")
nb_model = MultinomialNB()
nb_model.fit(X_train, y_train)

#Test Naive Bayes Model
y_pred_nb = nb_model.predict(X_test)

#Evaluate Naive Bayes
nb_conf_matrix = confusion_matrix(y_test, y_pred_nb)
nb_accuracy = accuracy_score(y_test, y_pred_nb)
nb_precision = precision_score(y_test, y_pred_nb)

print("Naive Bayes Confusion Matrix:\n", nb_conf_matrix)
print(f"Naive Bayes Accuracy: {nb_accuracy}")
print(f"Naive Bayes Precision: {nb_precision}")

#Compare performance
print("\nComparison of Models:")
print(f"MLP Accuracy: {mlp_accuracy} vs Naive Bayes Accuracy: {nb_accuracy}")
print(f"MLP Precision: {mlp_precision} vs Naive Bayes Precision: {nb_precision}")
```

In total this program had 3 layers (input layer, hidden layer, and output layer), and used a total of 1,130 neurons (1000 for input layer, 128 for hidden layer, and 2 for output layer.

Results

As you can see by the picture below, the MLP model outperformed the Naive Bayes model in both accuracy and precision. Because the main focus of this program is to detect phishing emails as accurately as possible, we should choose the model that minimizes the chances of missing phishing emails. We can use the function TP/(TP+FN) to calculate the model that performs the best in this area based on the data we got from the confusion matrices. TP stands for “True Positives”, which is the top left quadrant of the confusion matrices, and FN stands for “False Negatives” which is the bottom left quadrant of the confusion matrices.For the MLP this function gives us 3415/(3415 + 19) = 0.9945, while the Naive Bayes classifier gives us 3394/(3394 + 23) = 0.9065. Thus using this metric along with the accuracy and precision score as well as the runtime, the MLP appears to be the best model for detecting phishing emails.

```
(.venv) zacahryy@ZLAPTOP:~/Desktop/Classes/PhishingEmailDetection$ python3 mlp_483new.py
Training MLP...
Epoch 1/10, Loss: 0.03551078587770462
Epoch 2/10, Loss: 0.01616753451526165
Epoch 3/10, Loss: 0.11043664813041687
Epoch 4/10, Loss: 0.0016422400949522853
Epoch 5/10, Loss: 0.003857641015201807
Epoch 6/10, Loss: 0.0009172785212285817
Epoch 7/10, Loss: 0.005849200300872326
Epoch 8/10, Loss: 0.0012492547975853086
Epoch 9/10, Loss: 0.0015928864013403654
Epoch 10/10, Loss: 0.005474794190376997
Testing MLP...
MLP Confusion Matrix:
 [[3415   47]
 [  19 4350]]
MLP Accuracy: 0.9915719576043928
MLP Precision: 0.9893108937912213
Training Naive Bayes...
Naive Bayes Confusion Matrix:
 [[3394   68]
 [ 355 4014]]
Naive Bayes Accuracy: 0.9459839101008811
Naive Bayes Precision: 0.9833414992650662

Comparison of Models:
MLP Accuracy: 0.9915719576043928 vs Naive Bayes Accuracy: 0.9459839101008811
MLP Precision: 0.9893108937912213 vs Naive Bayes Precision: 0.9833414992650662
```