Small Data Project on Classification

MTH2137 - Machine Learning

Sam Wisnoski

Project Description: In this project, you will contemplate a potential application for classification, you will implement and improve upon a machine learning algorithm on a data set in support of this application, and you will evaluate the models performance

For this assignment, I chose to use the kaggle dataset "Poetry Foundation Poems" published by John Titor. This dataset houses a collection of 13,000+ poems from 3,000+ authors from the website www.poetryfoundation.org. For each poem included in the dataset, we are given the poem title, the poem content, the authors name, and tags relating to what the poem is about. For the means of this project, I will only be looking at the contents of the poem and the author of the poem (although I will still include the poem in my dataset as it makes validation easier).

We are going to attempt to use a machine learning algorithm - specifically, a nueral network - to predict the author of a poem based on the poem's contents. This application can be used for many things: it can help identify poems with unknown authors, or it can help aspiring poets who want to try and mimic a certain author's style. In order to have this system be usable, we are going to try to aim for at least 70% accuracy, which is usually the lowest acceptable accuracy for a useful ML model.

Here are the steps I took in order to reach this goal:

1. Loading the data and filtering it into poets with more than 45 poems
2. Spliting each poem into a list of words and removing common words from the poem
3. Converting each unique author and each unqiue word to a mathematical index
4. Padding the imputs (words in a poem) to all be the same length
5. Running a Linear Regression model to assess base accuracy
6. Running a simple MLP nueral network over my data to assess base accuracy
7. Looping over different wordcounts to attempt to improve accuracy
8. Looping over different numbers of hidden layers to attepmt to improve accuracy
9. Using TF-IDF on my data, which identifies term relevance in a set of words, and evaluating my data again using a simple MLP neural network.
10. Final analysis and reflection of the project

Now, let's walk through each step individually.

1. Load Data and Begin Filtering

Our first step is to load the data into our colab enviornment. This is done by using the kagglehub module to download the kaggle dataset, and the google.colab module to save the dataset into our drive. Then, we can access the data directly from our drive.

Our second step is to narrow the data to just the poet, poem, and title. Although using tags might help with classification, it is ultimately not the goal of the project. The goal of the project is to identify the author soley based on the content of the poem.

Lastly, we want to drastically lower the number of poets in our dataset. Machine learning can be extremely difficult with less than 30 entries for a specific class. In order to make sure we have enough data, we will only be focusing on poets that have more than 45 poems. This narrows our dataset to a total of 309 poems and five poets: William Shakespeare, Emily Dickinson, William Wordsworth, Rae Armantrout, and Alfred, Lord Tennyson. This filtered dataset is then saved to our drive and printed below.

```
import kagglehub
import os
import pandas as pd
from google.colab import drive


drive.mount('/content/drive')
path = kagglehub.dataset_download("tgdivy/poetry-foundation-poems")

poet_data_path = os.path.join(path, os.listdir(path)[0])
poetry_dataframe = pd.read_csv(poet_data_path)
poetry_dataframe = poetry_dataframe[["Title", "Poem", "Poet"]]

poet_counts = poetry_dataframe['Poet'].value_counts()
poets_with_more_than_45 = poet_counts[poet_counts >= 45].index
```

```
filtered_poetry_dataframe = poetry_dataframe[poetry_dataframe['Poet'].isin(poets_with_more_than_45)]

drive_path = '/content/drive/MyDrive/filtered_poetry_foundation_poems.csv'
filtered_poetry_dataframe.to_csv(drive_path, index=False)

filtered_poetry_dataframe
```

```
Mounted at /content/drive
Warning: Looks like you're using an outdated `kagglehub` version, please consider updating (latest version: 0.3.3)
Downloading from https://www.kaggle.com/api/v1/datasets/download/tgdivy/poetry-foundation-poems?dataset_version_number=1...
100%|██████████| 8.88M/8.88M [00:00<00:00, 11.5MB/s]Extracting files...
```

	Title	Poem	Poet
880	\r\r\n The Phoenix and the ...	\r\r\nLet the bird of loudest lay On the sole ...	William Shakespeare
913	\r\r\n This World is not Co...	\r\r\nThis World is not Conclusion.\r\r\nA Spe...	Emily Dickinson
1774	\r\r\n To fight aloud is ve...	\r\r\nTo fight aloud, is very brave - \r\r\nBu...	Emily Dickinson
1874	\r\r\n I like to see it lap...	\r\r\nI like to see it lap the Miles -\r\r\nAn...	Emily Dickinson
2002	\r\r\n Sonnet 123: No, Time...	\r\r\nNo, Time, thou shalt not boast that I do...	William Shakespeare
...	...	...	...
13420	\r\r\n Song: "Hark, hark! t...	\r\r\n\r\r\n\r\r\n\r\r\n(from Cymbeline)\r\r\n...	William Shakespeare
13421	\r\r\n Song: "O Mistress mi...	\r\r\n\r\r\n\r\r\n\r\r\n(from Twelfth Night)\r...	William Shakespeare
13422	\r\r\n Song: "Orpheus with ...	\r\r\n\r\r\n\r\r\n\r\r\n(from Henry VIII)\r\r\...	William Shakespeare
13423	\r\r\n Song: "Take, oh take...	\r\r\n\r\r\n\r\r\n\r\r\n(from Measure for Meas...	William Shakespeare
13424	\r\r\n Song: "Under the gre...	\r\r\n\r\r\n\r\r\n\r\r\n(from As You Like It)\...	William Shakespeare

309 rows × 3 columns

2. Splitting Poems into Words, Removing Common Words, and Adding Wordcount

The next step for our data is to start creating the actual list of words that will be fed into our algorithm.

We can use the Natural Language Toolkit (nltk) to import a list of stopwords, which we will then remove from all of the poems. A stopword is a word such as "as", "a", "the", or "and", that tells us nothing about the poet's writing style or contents of the poem. Removing these words will make the data less noisy and easier to classify. In the same step we remove these words, we also create a new column of data that contains a list of all the words in each poem. This list (after some modification) is what our model will be trained on and use for classification.

Lastly, we will use a function to add a column for wordcount - the wordcount will ultimately not be used in classification, but like the title, will help us understand our dataset more.

The updated dataset is again printed below.

```
import nltk
from nltk.corpus import stopwords

nltk.download('stopwords')
stop_words = set(stopwords.words('english'))

filtered_poetry_dataframe.loc[:, 'Words'] = filtered_poetry_dataframe['Poem'].apply(
    lambda x: [word for word in x.split() if word.lower() not in stop_words]
)

def count_words(poem):
    '''
    Function to count words in a poem after removing stop words
    Input: String of words
    Output: Integer of word count
    '''
    return len([word for word in poem.split() if word.lower() not in stop_words])

filtered_poetry_dataframe.loc[:, 'WordCount'] = filtered_poetry_dataframe['Poem'].apply(count_words)
filtered_poetry_dataframe = filtered_poetry_dataframe.reset_index(drop=True)

filtered_poetry_dataframe
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
<ipython-input-3-c50f623aca05>:7: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-cc
  filtered_poetry_dataframe.loc[:, 'Words'] = filtered_poetry_dataframe['Poem'].apply(
<ipython-input-3-c50f623aca05>:19: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-cc
  filtered_poetry_dataframe.loc[:, 'WordCount'] = filtered_poetry_dataframe['Poem'].apply(count_words)
```

	Title	Poem	Poet	Words	WordCount
0	\r\r\n The Phoenix and the ...	\r\r\nLet the bird of loudest lay On the sole ...	William Shakespeare	[Let, bird, loudest, lay, sole, Arabian, tree,...	212
1	\r\r\n This World is not Co...	\r\r\nThis World is not Conclusion.\r\r\nA Spe...	Emily Dickinson	[World, Conclusion., Species, stands, beyond, ...	72
2	\r\r\n To fight aloud is ve...	\r\r\nTo fight aloud, is very brave - \r\r\nBu...	Emily Dickinson	[fight, aloud,, brave, -, gallanter,, know, ch...	43
3	\r\r\n I like to see it lap...	\r\r\nI like to see it lap the Miles -\r\r\nAn...	Emily Dickinson	[like, see, lap, Miles, -, lick, Valleys, -, s...	54
4	\r\r\n Sonnet 123: No, Time...	\r\r\nNo, Time, thou shalt not boast that I do...	William Shakespeare	[No,, Time,, thou, shalt, boast, change:Thy, p...	57
...	...	...	...	...	...
304	\r\r\n Song: “Hark, hark! t...	\r\r\n\r\r\n\r\r\n\r\r\n(from Cymbeline)\r\r\n...	William Shakespeare	[(from, Cymbeline), Hark,, hark!, lark, heaven...	32
305	\r\r\n Song: “O Mistress mi...	\r\r\n\r\r\n\r\r\n\r\r\n(from Twelfth Night)\r...	William Shakespeare	[(from, Twelfth, Night), Mistress, mine, roami...	49
306	\r\r\n Song: “Orpheus with	\r\r\n\r\r\n\r\r\n\r\r\n(from Henry VIII)\r\r\...	William	[(from, Henry, VIII), Orpheus, lute, made,	44

3. Convert Each Word and Author to an Index

In order to feed the newly created list of words into our machine learning algorithm, we need to translate the list of words into a list of numbers. The best way to do this would be to assign each unique word to an index. For example, the word "tree" might be assigned index number 1. Anywhere the word "tree" is included in a list, a 1 will be included instead. The same process will occur for each unique word.

We then do the same for authors, using a dictionary instead of a function. We can use a dictionay here since each poem was written by a single author, while each poem can contain hundreds of different words.

After indexing the poems and authors, the updated dataset is displayed below.

```python
words = set(word for word_list in filtered_poetry_dataframe['Words'] for word in word_list)

word_to_index = {word: idx for idx, word in enumerate(words)}

def words_to_word_indices(words_list):
    '''
    Function to convert a list of words (from the "Words" column) to a list of word indices
    Input: List of words
    Output: List of word indices
    '''
    return [word_to_index[word] for word in words_list]

filtered_poetry_dataframe['poem_word_indices'] = filtered_poetry_dataframe['Words'].apply(words_to_word_indices)

author_to_index = {author: idx for idx, author in enumerate(set(filtered_poetry_dataframe['Poet']))}
filtered_poetry_dataframe['author_index'] = filtered_poetry_dataframe['Poet'].map(author_to_index)

filtered_poetry_dataframe
```

	Title	Poem	Poet	Words	WordCount	poem_word_indices	author_index
0	\r\r\n The Phoenix and the ...	\r\r\nLet the bird of loudest lay On the sole ...	William Shakespeare	[Let, bird, loudest, lay, sole, Arabian, tree,...	212	[7436, 14158, 15099, 4722, 9151, 6638, 8418, 1...	4
1	\r\r\n This World is not Co...	\r\r\nThis World is not Conclusion.\r\r\nA Spe...	Emily Dickinson	[World, Conclusion., Species, stands, beyond, ...	72	[7197, 18035, 17228, 11592, 10211, 449, 12541,...	3
2	\r\r\n To fight aloud is ve...	\r\r\nTo fight aloud, is very brave - \r\r\nBu...	Emily Dickinson	[fight, aloud,, brave, -, gallanter,, know, ch...	43	[17861, 991, 17227, 449, 15100, 12444, 7733, 6...	3
3	\r\r\n I like to see it lap...	\r\r\nI like to see it lap the Miles - \r\r\nAn...	Emily Dickinson	[like, see, lap, Miles, -, lick, Valleys, -, s...	54	[1684, 13036, 9023, 9272, 449, 3910, 7025, 449...	3
4	\r\r\n Sonnet 123: No, Time...	\r\r\nNo, Time, thou shalt not boast that I do...	William Shakespeare	[No,, Time,, thou, shalt, boast, change:Thy, p...	57	[925, 5504, 3152, 3285, 13628, 16707, 8823, 98...	4
...	...	...	...	...	...	...	...
304	\r\r\n Song: "Hark, hark! t...	\r\r\n\r\r\n\r\r\n\r\r\n(from Cymbeline)\r\r\n...	William Shakespeare	[(from, Cymbeline), Hark,, hark!, lark, heaven...	32	[8140, 5133, 15961, 15983, 12532, 16178, 3058,...	4

4. Padding the Input to Equal Lengths

In order for our nueral network to run correctly, we need to have each input be the same length. However, currently, the length of each list of indices is simply however many words the poem is. This could be 20 words, or it could be 20,000.

First, we find the maximum number of words based on the wordcount column we created earlier. Then, we pad each list to the maximun word count, adding zeros to each - this is called zero padding. Zero padding ensures all poems have the same number of word indices without introducing any meaningful values.

However, we do not wish all poems to be the maximum length, or most of the word lists will be 90% zeros. To make the poems a meaningful length, we then only look at the first 200 words/indices of each list.

The updated dataset is then displayed below.

With these lists, each containing 200 indices corresponding to (up to) the first 200 words in the poem, we are now ready to create and run the first iteration of our model.

```
max_index = filtered_poetry_dataframe['WordCount'].max()

filtered_poetry_dataframe['poem_indices_padded'] = filtered_poetry_dataframe['poem_word_indices'].apply(lambda x: x + [0] * (max_index - ler

def pad_to_200(x):
    '''
    Function to pad a list of word indices to 200
    Input: List of word indices
    Output: List of word indices padded to 200
    '''
    return x[:200] + [0] * (200 - len(x)) if len(x) < 200 else x[:200]

filtered_poetry_dataframe['poem_indices_first_200'] = filtered_poetry_dataframe['poem_word_indices'].apply(pad_to_200)

filtered_poetry_dataframe
```

	Title	Poem	Poet	Words	WordCount	poem_word_indices	author_index	poem_indices_padded	poem_i
0	\r\r\n The Phoenix and the ...	\r\r\nLet the bird of loudest lay On the sole ...	William Shakespeare	[Let, bird, loudest, lay, sole, Arabian, tree,...	212	[7436, 14158, 15099, 4722, 9151, 6638, 8418, 1...	4	[7436, 14158, 15099, 4722, 9151, 6638, 8418, 1...	[7436, 9
1	\r\r\n This World is not Co...	\r\r\nThis World is not Conclusion.\r\r\nA Spe...	Emily Dickinson	[World, Conclusion., Species, stands, beyond, ...	72	[7197, 18035, 17228, 11592, 10211, 449, 12541,...	3	[7197, 18035, 17228, 11592, 10211, 449, 12541,...	[
2	\r\r\n To fight aloud is ve...	\r\r\nTo fight aloud, is very brave - \r\r\nBu...	Emily Dickinson	[fight, aloud,, brave, -, gallanter,, know, ch...	43	[17861, 991, 17227, 449, 15100, 12444, 7733, 6...	3	[17861, 991, 17227, 449, 15100, 12444, 7733, 6...	[178 151
3	\r\r\n I like to see it lap...	\r\r\nI like to see it lap the Miles -\r\r\nAn...	Emily Dickinson	[like, see, lap, Miles, -, lick, Valleys,	54	[1684, 13036, 9023, 9272, 449, 3910, 7025, 449...	3	[1684, 13036, 9023, 9272, 449, 3910, 7025, 449...	[1684 44

5a. Create a Linear Regression Model

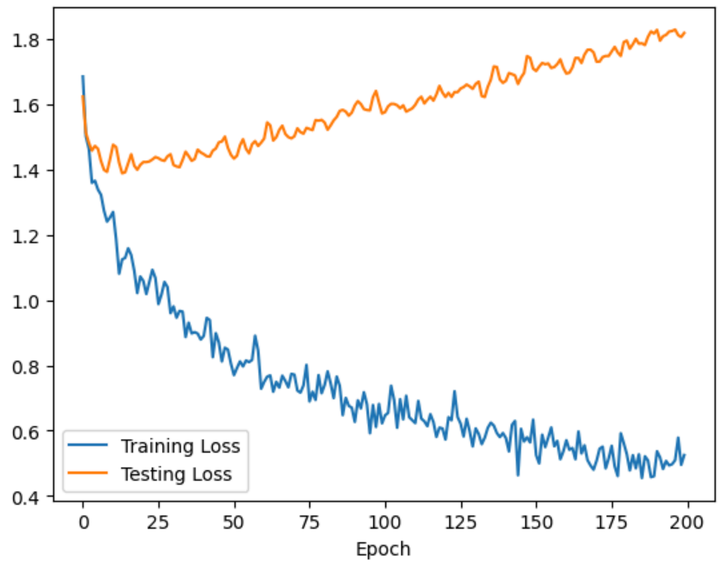
Now that we have our data sorted and normalized, we can begin creating out machine learning model. We begin by implementing a logistic regression classifier using PyTorch to categorize poems by their authors. The LogisticRegression class defines the model, which consists of a single linear layer that transforms input features (the word indices of the poems) into output classes (the authors). The poem data is converted into NumPy arrays, normalized, and then transformed into PyTorch tensors, in order for the data to be correctly processed by PyTorch.

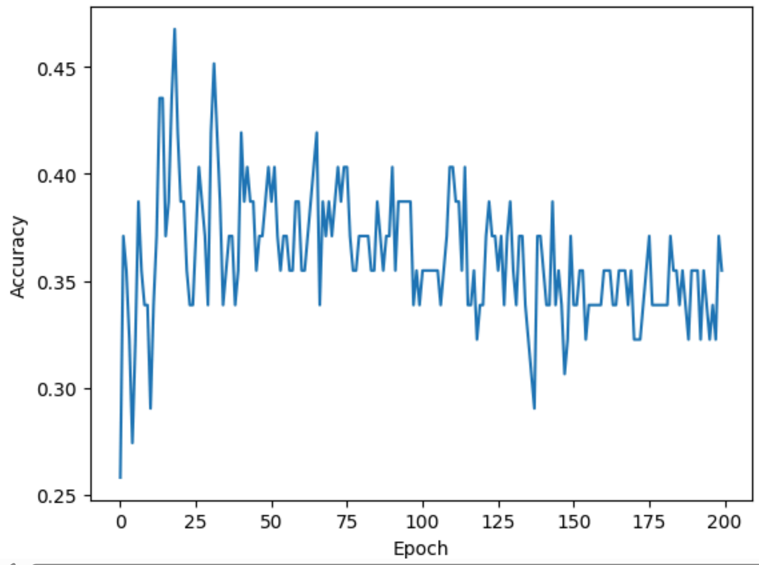
To handle potential class imbalance, weights for each class are calculated and a WeightedRandomSampler is created for balanced training batches. The model is trained over a specified number of epochs using the Adam optimizer and cross-entropy loss. During training, the model makes predictions, computes losses, and updates weights through backpropagation.

After training, the code generates plots of training and testing losses and accuracy over the epochs, allowing us to assess the models accuracy.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from sklearn.model_selection import train_test_split
import numpy as np
from torch.utils.data import DataLoader, WeightedRandomSampler
import matplotlib.pyplot as plt
import seaborn as sns

class LogisticRegression(nn.Module):
    """ A model that implements a logistic regression classifier. """
    def __init__(self, input_size, num_classes):
        super(LogisticRegression, self).__init__()
        # initialize the model weights
        self.linear = nn.Linear(input_size, num_classes)

    def forward(self, x):
        """ Implement the forward pass of the model. """
        out = self.linear(x)
        return out


# Check for GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Initialize the model
model = LogisticRegression(200, 5).to(device)

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(
    filtered_poetry_dataframe['poem_indices_first_200'].tolist(),
    filtered_poetry_dataframe['author_index'].tolist(),
    test_size=0.2,
    random_state=42
)
```

```
# Convert to NumPy arrays
X_train = np.array(X_train, dtype=np.float32)
y_train = np.array(y_train)
X_test = np.array(X_test, dtype=np.float32)
y_test = np.array(y_test)

# Normalize inputs (optional)
X_train = (X_train - np.mean(X_train)) / np.std(X_train)
X_test = (X_test - np.mean(X_test)) / np.std(X_test)

# Convert to PyTorch tensors and move to the device
X_train = torch.from_numpy(X_train).to(device)
y_train = torch.from_numpy(y_train).to(device)
X_test = torch.from_numpy(X_test).to(device)
y_test = torch.from_numpy(y_test).to(device)

# Calculate class weights for weighted sampling
class_sample_count = np.array([len(np.where(y_train.cpu() == t)[0]) for t in np.unique(y_train.cpu())])
weights = 1. / (class_sample_count + 1e-5)  # Avoid division by zero
sample_weights = np.array([weights[t] for t in y_train.cpu()])
sample_weights = torch.from_numpy(sample_weights).double()

# Create a WeightedRandomSampler for the training set
weighted_sampler = WeightedRandomSampler(sample_weights, len(sample_weights))

# Create DataLoader for the training and testing set
train_loader = DataLoader(dataset=list(zip(X_train, y_train)), sampler=weighted_sampler, batch_size=32)
test_loader = DataLoader(dataset=list(zip(X_test, y_test)), batch_size=32, shuffle=True)

# Training setup
n_epochs = 200
learning_rate = 0.001  # Lower learning rate for Adam
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)  # Use Adam optimizer

train_losses = []
test_losses = []
accuracies = []

# Training loop
for epoch in range(n_epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    train_losses.append(running_loss / len(train_loader))

    model.eval()
    with torch.no_grad():
        running_test_loss = 0.0
        correct = 0
        total = 0
        for inputs, labels in test_loader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            running_test_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

        test_losses.append(running_test_loss / len(test_loader))
        accuracies.append(correct / total)

# Plot losses and accuracy
plt.figure()
plt.plot(train_losses, label='Training Loss')
plt.plot(test_losses, label='Testing Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()
plt.figure()
plt.plot(accuracies)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.show()
```





As we can see on the plots, we are not achieving the desired accuracy. Our accuracy ranges from a low of 20% to a high of around 40%. And while our accuracy does increase over epochs, the testing loss also increases. This points to a potential problem of overfitting the data based on the training data.

5b. Evaluate the Linear Regression Model

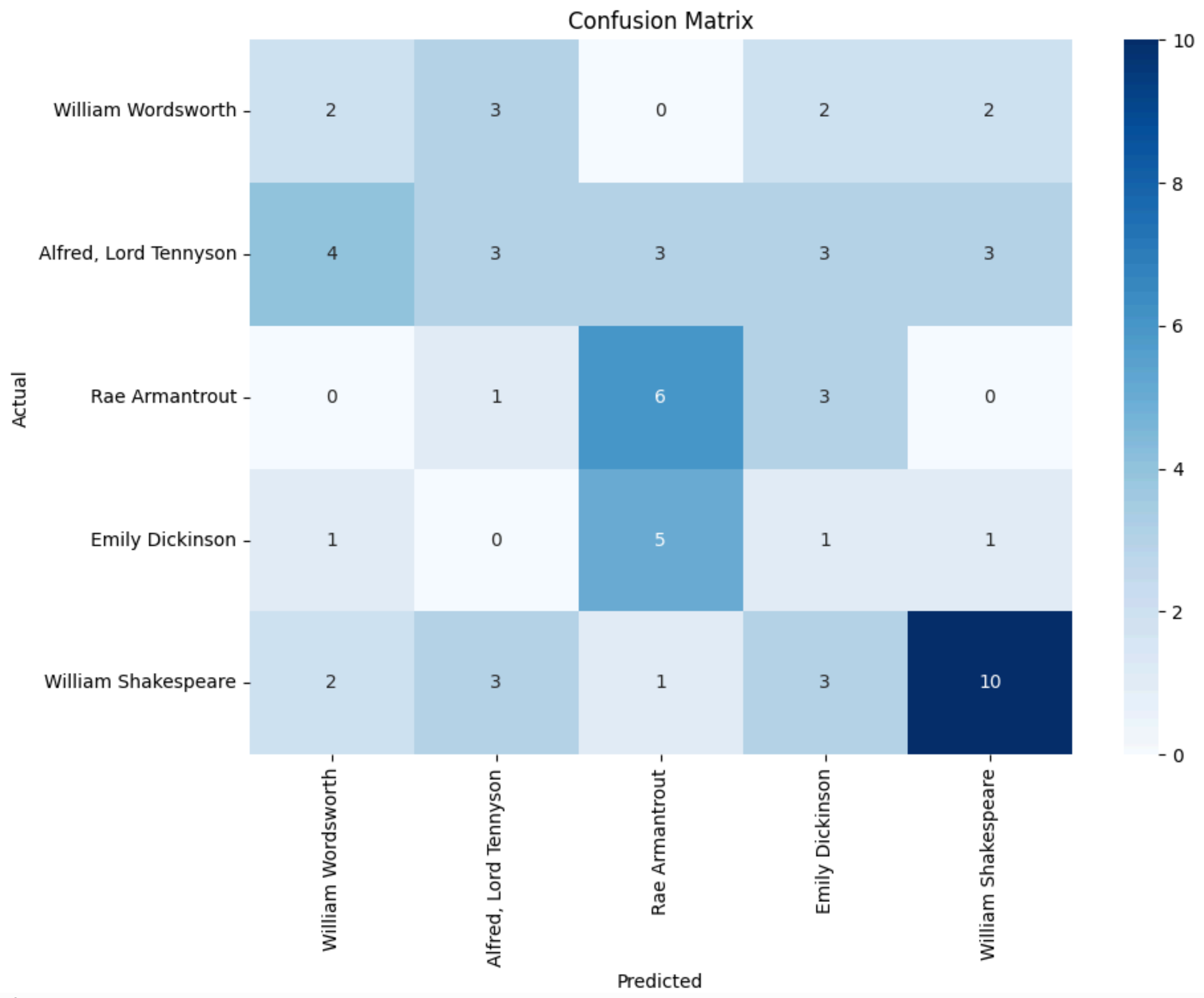
Since the model significantly underperformed in classifying poets, we can generate a confusion matrix and additional statistics in order to . We set the model to evaluation mode, and predictions are made on the test set without calculating gradients to save computational resources.

We compute precision, recall, and accuracy to get a sense of how well the model is performing. We also create a confusion matrix to visualize how the model's predictions compare to the actual labels. This matrix is presented as a heatmap, making it easy to identify patterns in misclassification, such as which authors are frequently confused with each other, which can inform further adjustments to improve the model.

```
from sklearn.metrics import precision_score, recall_score, accuracy_score, confusion_matrix

model.eval()
index_to_author = {idx: author for author, idx in author_to_index.items()}
```

```
# Predictions on test set
with torch.no_grad():
    test_outputs = model(X_test)
    _, predicted = torch.max(test_outputs, 1)

# Convert tensors to numpy arrays for easier calculation
y_test_np = y_test.cpu().numpy()
predicted_np = predicted.cpu().numpy()

# Calculate precision, recall, and accuracy
precision = precision_score(y_test_np, predicted_np, average='weighted')
recall = recall_score(y_test_np, predicted_np, average='weighted')
accuracy = accuracy_score(y_test_np, predicted_np)

print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'Accuracy: {accuracy:.4f}')

# Calculate the confusion matrix
cm = confusion_matrix(y_test_np, predicted_np)

# Create a heatmap to visualize the confusion matrix
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=[index_to_author[i] for i in range(len(index_to_author))],
            yticklabels=[index_to_author[i] for i in range(len(index_to_author))])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

```
Precision: 0.3765
Recall: 0.3548
Accuracy: 0.3548
```



Actual \ Predicted	William Wordsworth	Alfred, Lord Tennyson	Rae Armantrout	Emily Dickinson	William Shakespeare
William Wordsworth	2	3	0	2	2
Alfred, Lord Tennyson	4	3	3	3	3
Rae Armantrout	0	1	6	3	0
Emily Dickinson	1	0	5	1	1
William Shakespeare	2	3	1	3	10

When examining the accuracy of this model, we should compare it to a base accuracy of 0.20, or 20%. If we were given a poem and randomly distributed it between the five poets, we would expect to be right one of every five times. While our accuracy is slightly better than 20%, it is far below our benchmark of 70%.

Unfortunetly, there are no glaring, obvious flaws in the confusion matrix besides it just being generally wrong. Rather than attempting to fix the model by tuning hyper-parameters, we are instead going to try a different type of machine learning - a neural network.

6a. Create a MLP Model

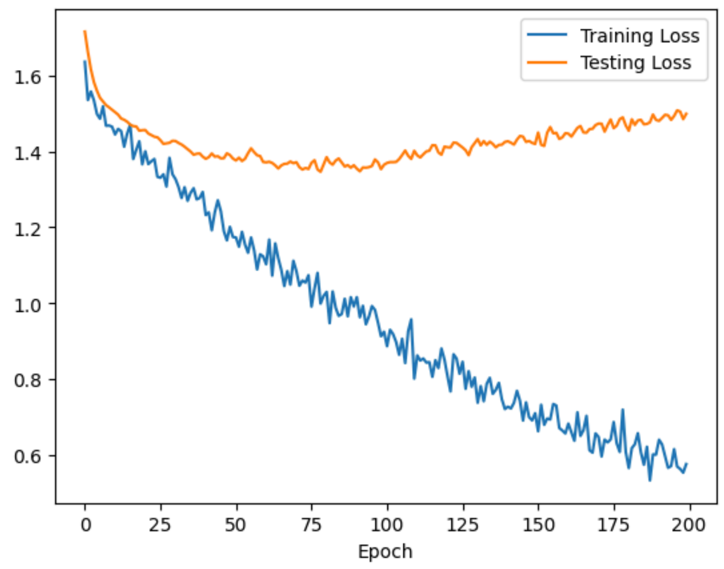
Since Linear Regression did not proivde enough accuracy, we are now going to try a simple Multi-Layer Perceptron (MLP) using PyTorch. Like the Linear Regression mode, the MLP model is defined with an input layer, the poems indices, and an output layer corresponding to the number of authors (classes). However, the MLP also has a hidden layer between the input and output layer that adds additional complexity to the model.

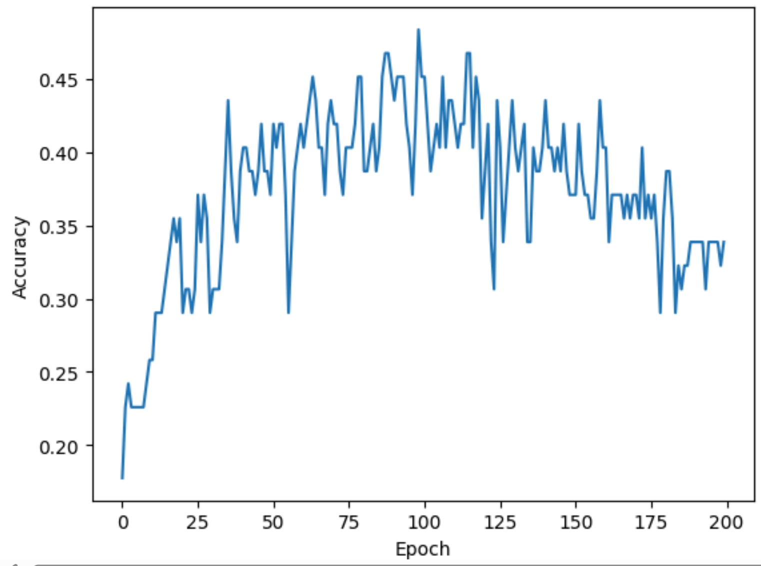
Besides the change to a MLP model, the remaining processing is largely the same to that of a logistic regression. We still use a WeightedRandomSampler, the Adam optimizer, and cross-entropy loss.

To keep comparisions the same between the models, we again generate plots of training and testing losses and accuracy over the epochs.

```
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(MLP, self).__init__()
        self.linear_1 = nn.Linear(input_size, hidden_size)
        self.linear_2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.sigmoid(out)
        out = self.linear_2(out)
        return out

# Check for GPU
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Initialize the model
MLPmodel = MLP(200, 10, 5).to(device)

# Train/test split
X_train, X_test, y_train, y_test = train_test_split(
    filtered_poetry_dataframe['poem_indices_first_200'].tolist(),
    filtered_poetry_dataframe['author_index'].tolist(),
    test_size=0.2,
    random_state=42
)

# Convert to NumPy arrays
X_train = np.array(X_train, dtype=np.float32)
y_train = np.array(y_train)
X_test = np.array(X_test, dtype=np.float32)
y_test = np.array(y_test)

# Normalize inputs (optional)
X_train = (X_train - np.mean(X_train)) / np.std(X_train)
X_test = (X_test - np.mean(X_test)) / np.std(X_test)

# Convert to PyTorch tensors and move to the device
X_train = torch.from_numpy(X_train).to(device)
y_train = torch.from_numpy(y_train).to(device)
X_test = torch.from_numpy(X_test).to(device)
y_test = torch.from_numpy(y_test).to(device)

# Calculate class weights for weighted sampling
class_sample_count = np.array([len(np.where(y_train.cpu() == t)[0]) for t in np.unique(y_train.cpu())])
weights = 1. / (class_sample_count + 1e-5)  # Avoid division by zero
sample_weights = np.array([weights[t] for t in y_train.cpu()])
sample_weights = torch.from_numpy(sample_weights).double()

# Create a WeightedRandomSampler for the training set
weighted_sampler = WeightedRandomSampler(sample_weights, len(sample_weights))

# Create DataLoader for the training and testing set
train_loader = DataLoader(dataset=list(zip(X_train, y_train)), sampler=weighted_sampler, batch_size=32)
```

```
test_loader = DataLoader(dataset=list(zip(X_test, y_test)), batch_size=32, shuffle=True)

# Training setup
n_epochs = 200
learning_rate = 0.001  # Lower learning rate for Adam
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(MLPmodel.parameters(), lr=learning_rate)  # Use Adam optimizer

train_losses = []
test_losses = []
accuracies = []

# Training loop
for epoch in range(n_epochs):
    MLPmodel.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = MLPmodel(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    train_losses.append(running_loss / len(train_loader))

    MLPmodel.eval()
    with torch.no_grad():
        running_test_loss = 0.0
        correct = 0
        total = 0
        for inputs, labels in test_loader:
            outputs = MLPmodel(inputs)
            loss = criterion(outputs, labels)
            running_test_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

        test_losses.append(running_test_loss / len(test_loader))
        accuracies.append(correct / total)

# Plot losses and accuracy
plt.figure()
plt.plot(train_losses, label='Training Loss')
plt.plot(test_losses, label='Testing Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()

plt.figure()
plt.plot(accuracies)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.show()
```





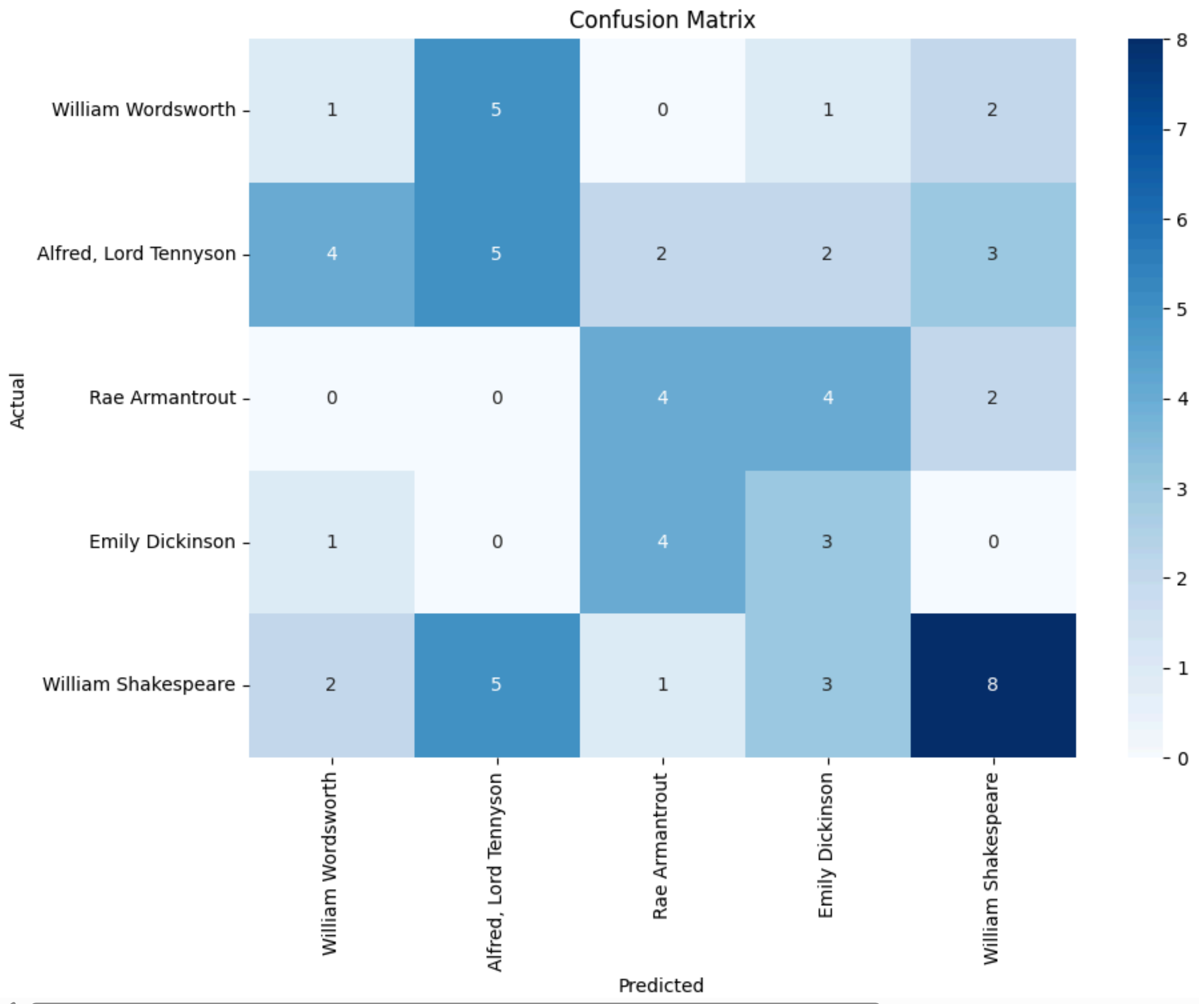
6b. Evaluate the MLP model

We can already see that our accuracy is not ideal - it seems to be about the same as the logistic regrssion model, with an average of ~35%. Again, this is better than the "average" of 20%, but a far cry from our 70% goal.

We will evalute the model in the same way, generating data for the precision, accuracy recall, and creating a confusion matrix for a visual representation.

```
MLPmodel.eval()

# Predictions on test set
with torch.no_grad():
    test_outputs = MLPmodel(X_test)
    _, predicted = torch.max(test_outputs, 1)

# Convert tensors to numpy arrays for easier calculation
y_test_np = y_test.cpu().numpy()
predicted_np = predicted.cpu().numpy()

# Calculate precision, recall, and accuracy
precision = precision_score(y_test_np, predicted_np, average='weighted')
recall = recall_score(y_test_np, predicted_np, average='weighted')
accuracy = accuracy_score(y_test_np, predicted_np)

print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'Accuracy: {accuracy:.4f}')

# Calculate the confusion matrix
cm = confusion_matrix(y_test_np, predicted_np)

# Create a heatmap to visualize the confusion matrix
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=[index_to_author[i] for i in range(len(index_to_author))],
            yticklabels=[index_to_author[i] for i in range(len(index_to_author))])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

```
Precision: 0.3560
Recall: 0.3387
Accuracy: 0.3387
```



Evidently, our confusion matrix doesn't look much prettier than before. Our accuracy is slightly higher, but not noticably so. Let's try tweaking the MLP model to increase accuracy.

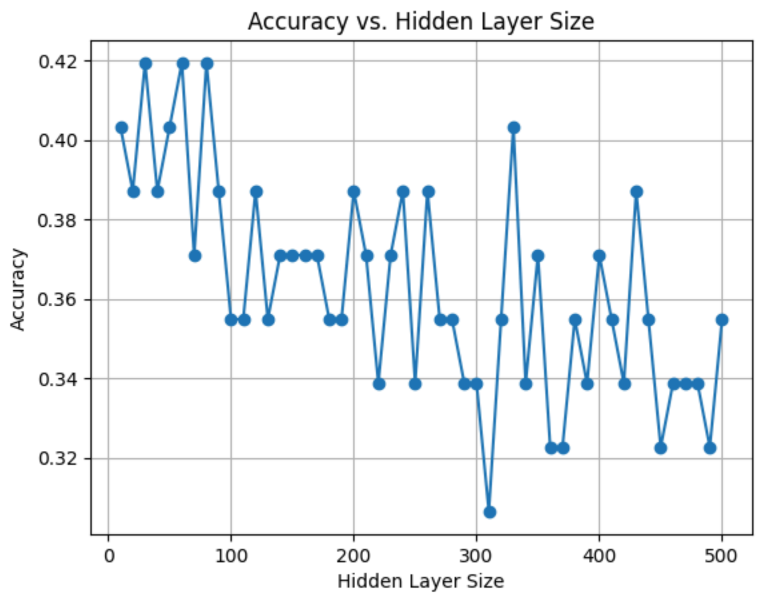
7. Searching Over Hidden Layers to Find Optimal Number of Hidden Layers

In theory, a MLP model should outperform a Linear Regression model. The addition of hidden layers and extra complexity should do better with a complex dataset such as natural language processing. However, perhaps we are using a bad number of hidden layers. In order to improve our MLP, let's test over different numbers of hidden layers.

We can test this by running the model over a simple for loop. Each time the model is run, we switch the hidden layer accuracy. After running the model 50 times, from 10 hidden layers to 500 hidden layers, we plot the number of hidden layers in respect to accuracy.

By systematically varying the hidden layer size in our neural network, we can identify how network complexity influences accuracy. Smaller hidden layers may not capture enough patterns in the data, while excessively large layers could lead to overfitting or inefficiency. The goal is to find the right size that balances learning capacity without unnecessary complexity.

```
# Initialize the list to store accuracies for each hidden layer size
hidden_layer_accuracies = []

# Iterate over different hidden layer sizes
for hidden_size in range(10, 501, 10):

    # Train/test split
    X_train, X_test, y_train, y_test = train_test_split(
        filtered_poetry_dataframe['poem_indices_first_200'].tolist(),
        filtered_poetry_dataframe['author_index'].tolist(),
        test_size=0.2,
        random_state=42
    )

    # Convert to NumPy arrays
    X_train = np.array(X_train, dtype=np.float32)
    y_train = np.array(y_train)
    X_test = np.array(X_test, dtype=np.float32)
    y_test = np.array(y_test)

    # Normalize inputs (optional)
    X_train = (X_train - np.mean(X_train)) / np.std(X_train)
    X_test = (X_test - np.mean(X_test)) / np.std(X_test)

    # Convert to PyTorch tensors and move to the device
    X_train = torch.from_numpy(X_train).to(device)
    y_train = torch.from_numpy(y_train).to(device)
    X_test = torch.from_numpy(X_test).to(device)
    y_test = torch.from_numpy(y_test).to(device)

    # Calculate class weights for weighted sampling
    class_sample_count = np.array([len(np.where(y_train.cpu() == t)[0]) for t in np.unique(y_train.cpu())])
    weights = 1. / (class_sample_count + 1e-5)  # Avoid division by zero
    sample_weights = np.array([weights[t] for t in y_train.cpu()])
    sample_weights = torch.from_numpy(sample_weights).double()

    # Create a WeightedRandomSampler for the training set
    weighted_sampler = WeightedRandomSampler(sample_weights, len(sample_weights))

    # Create DataLoader for the training and testing set
    train_loader = DataLoader(dataset=list(zip(X_train, y_train)), sampler=weighted_sampler, batch_size=32)
    test_loader = DataLoader(dataset=list(zip(X_test, y_test)), batch_size=32, shuffle=True)

    # Initialize the model for the current hidden layer size
    model = MLP(200, hidden_size, 5).to(device)

    # Training setup
    n_epochs = 50  # You can adjust this
    learning_rate = 0.001  # Lower learning rate for Adam
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    # Training loop
    for epoch in range(n_epochs):
        model.train()
        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

    # Evaluation
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    # Calculate accuracy for the current hidden layer size
    accuracy = correct / total
    hidden_layer_accuracies.append(accuracy)

# Plot the accuracies for each hidden layer size
plt.figure()
plt.plot(range(10, 501, 10), hidden_layer_accuracies, marker='o')
plt.xlabel('Hidden Layer Size')
plt.ylabel('Accuracy')
plt.title('Accuracy vs. Hidden Layer Size')
plt.grid(True)
plt.show()
```



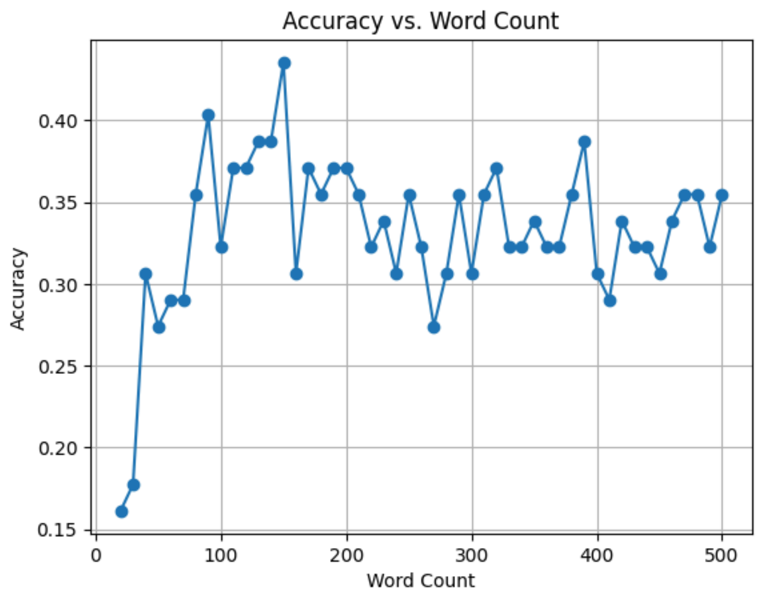
Even variation in hidden layers, accuracy doesn't reach above 50%. Additionally, there is no consistant pattern in the data. Perhaps it is not the model that is the problem, but the data. It could be we are including too few or two many variables for each poem when calculating poem indices.

8. Searching Over Wordcounts to Find Optimal Wordcounts

We now explore the impact of poem length on the accuracy of our model. By iterating over different word counts, we can investigate whether longer poems provide more useful information for classification or if there is a point where additional words contribute little to model performance/deteriorate model performance. Shorter poems may not provide enough context for accurate predictions, while excessively long ones might introduce noise or unnecessary details. Finding the right balance can help optimize the model's effectiveness.

To analyze the effect of word count, we can use a simple for loop, iterating over the varying count accuracies. With each loop, we create a "new" dataset with poem indices matching that loop. With each run, we calculate the accuracy and record it. When finished with all runs, we graph the accuracy against the number of words included in analysis.

```
# Initialize the list to store accuracies for each word count
word_count_accuracies = []

# Iterate over different word counts
for word_count in range(20, 501, 10):

    # Function to pad/truncate the poems to the current word count
    def pad_to_word_count(x):
        return x[:word_count] + [0] * (word_count - len(x)) if len(x) < word_count else x[:word_count]

    # Apply padding/truncation to 'poem_word_indices'
    filtered_poetry_dataframe['poem_indices_current'] = filtered_poetry_dataframe['poem_word_indices'].apply(pad_to_word_count)

    # Train/test split
    X_train, X_test, y_train, y_test = train_test_split(
        filtered_poetry_dataframe['poem_indices_current'].tolist(),
        filtered_poetry_dataframe['author_index'].tolist(),
        test_size=0.2,
        random_state=42
    )

    # Convert to NumPy arrays
    X_train = np.array(X_train, dtype=np.float32)
    y_train = np.array(y_train)
    X_test = np.array(X_test, dtype=np.float32)
    y_test = np.array(y_test)

    # Normalize inputs (optional)
    X_train = (X_train - np.mean(X_train)) / np.std(X_train)
    X_test = (X_test - np.mean(X_test)) / np.std(X_test)

    # Convert to PyTorch tensors and move to the device
    X_train = torch.from_numpy(X_train).to(device)
    y_train = torch.from_numpy(y_train).to(device)
    X_test = torch.from_numpy(X_test).to(device)
    y_test = torch.from_numpy(y_test).to(device)

    # Calculate class weights for weighted sampling
    class_sample_count = np.array([len(np.where(y_train.cpu() == t)[0]) for t in np.unique(y_train.cpu())])
    weights = 1. / (class_sample_count + 1e-5)  # Avoid division by zero
    sample_weights = np.array([weights[t] for t in y_train.cpu()])
    sample_weights = torch.from_numpy(sample_weights).double()

    # Create a WeightedRandomSampler for the training set
    weighted_sampler = WeightedRandomSampler(sample_weights, len(sample_weights))

    # Create DataLoader for the training and testing set
    train_loader = DataLoader(dataset=list(zip(X_train, y_train)), sampler=weighted_sampler, batch_size=32)
    test_loader = DataLoader(dataset=list(zip(X_test, y_test)), batch_size=32, shuffle=True)

    # Initialize the model for the current word count
    model = MLP(word_count, 500, 5).to(device)

    # Training setup
    n_epochs = 50  # You can adjust this
    learning_rate = 0.001  # Lower learning rate for Adam
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    # Training loop
    for epoch in range(n_epochs):
        model.train()
        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

    # Evaluation
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    # Calculate accuracy for the current word count
    accuracy = correct / total
    word_count_accuracies.append(accuracy)

# Plot the accuracies for each word count
plt.figure()
plt.plot(range(20, 501, 10), word_count_accuracies, marker='o')
plt.xlabel('Word Count')
plt.ylabel('Accuracy')
plt.title('Accuracy vs. Word Count')
plt.grid(True)
plt.show()
```



Although accuracy seems to be highest between 100 and 300 words, it seems that overall, word count has a minimal effect on accuracy. If tweaking both our model and our inputs had no effect on the performance of the model, let's try a signfigcantly different set of data.

9a. Implementing TF-IDF and Rerunning the Model

Let's try creating a totally different set of data. Instead of turning the words of the poems into indices, we will use Term Frequency-Inverse Document Frequency on our dataset. TF-IDF allows the model to focus on the most important words across the dataset, ignoring less meaningful terms. By transforming the poems into TF-IDF vectors, we aim to capture key word patterns that are helpful in classifying authors based on their writing styles.

Although we are using a different data set, we will still use the same MLP model to process the data. As with both the Linear and MLP models, after each epoch, we evaluate the model's performance on a test set and monitor both the training and testing loss, along with the classification accuracy.

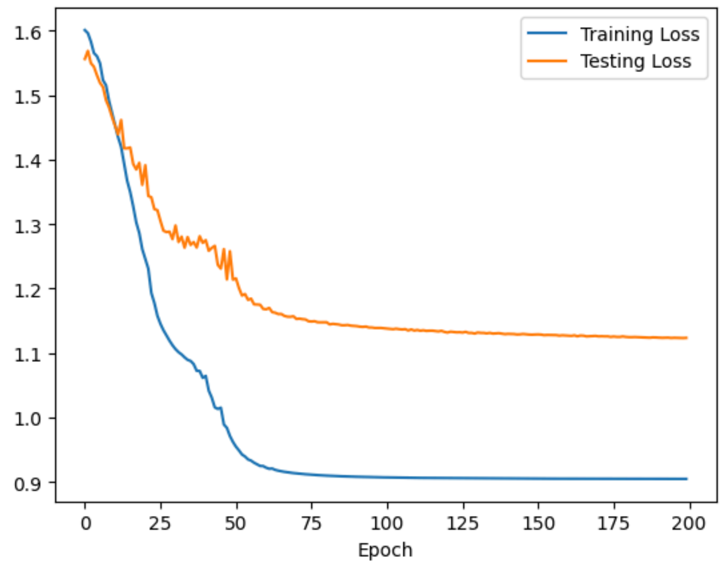
Hopefully, this helps us understand the model's learning curve and identify any potential overfitting.

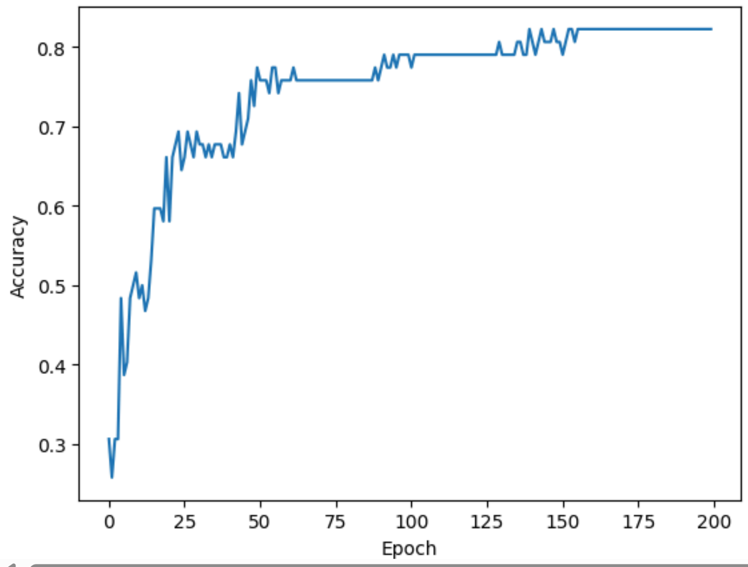
```
from sklearn.feature_extraction.text import TfidfVectorizer

# Create a TF-IDF vectorizer
tfidf_vectorizer = TfidfVectorizer(max_features=400)  # Limit to 400 features
X = tfidf_vectorizer.fit_transform(filtered_poetry_dataframe['Poem']).toarray()
y = np.array(filtered_poetry_dataframe['author_index'])

print(X.shape)
print(X)
```

```
(309, 400)
[[0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.18857208 0.         ... 0.         0.         0.        ]
 ...
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.3372818  ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.18629501 0.         0.        ]]
```

```
from torch.utils.data import DataLoader, TensorDataset


# Train/Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Convert to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test, dtype=torch.long)
```

```
# Create DataLoaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

train_loader = DataLoader(dataset=train_dataset, batch_size=8, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=8, shuffle=False)

# Define the Neural Network Model
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(MLP, self).__init__()
        self.linear_1 = nn.Linear(input_size, hidden_size)
        self.linear_2 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.linear_1(x)
        out = F.sigmoid(out)
        out = self.linear_2(out)
        out = F.softmax(out, dim=1)
        return out

# Initialize the model
input_size = X_train.shape[1]
num_classes = len(np.unique(y))  # Number of unique authors
hidden_layers = 500
model = MLP(input_size, hidden_layers, num_classes)

# Training Setup
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)


train_losses = []
test_losses = []
accuracies = []

# Training Loop
n_epochs = 200
for epoch in range(n_epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    train_losses.append(running_loss / len(train_loader))

# Evaluate on Test Data
    model.eval()
    with torch.no_grad():
        running_test_loss = 0.0
        correct = 0
        total = 0
        for inputs, labels in test_loader:
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            running_test_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

        test_losses.append(running_test_loss / len(test_loader))
        accuracies.append(correct / total)



# Plot losses and accuracy
plt.figure()
plt.plot(train_losses, label='Training Loss')
plt.plot(test_losses, label='Testing Loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()

plt.figure()
plt.plot(accuracies)
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.show()
```



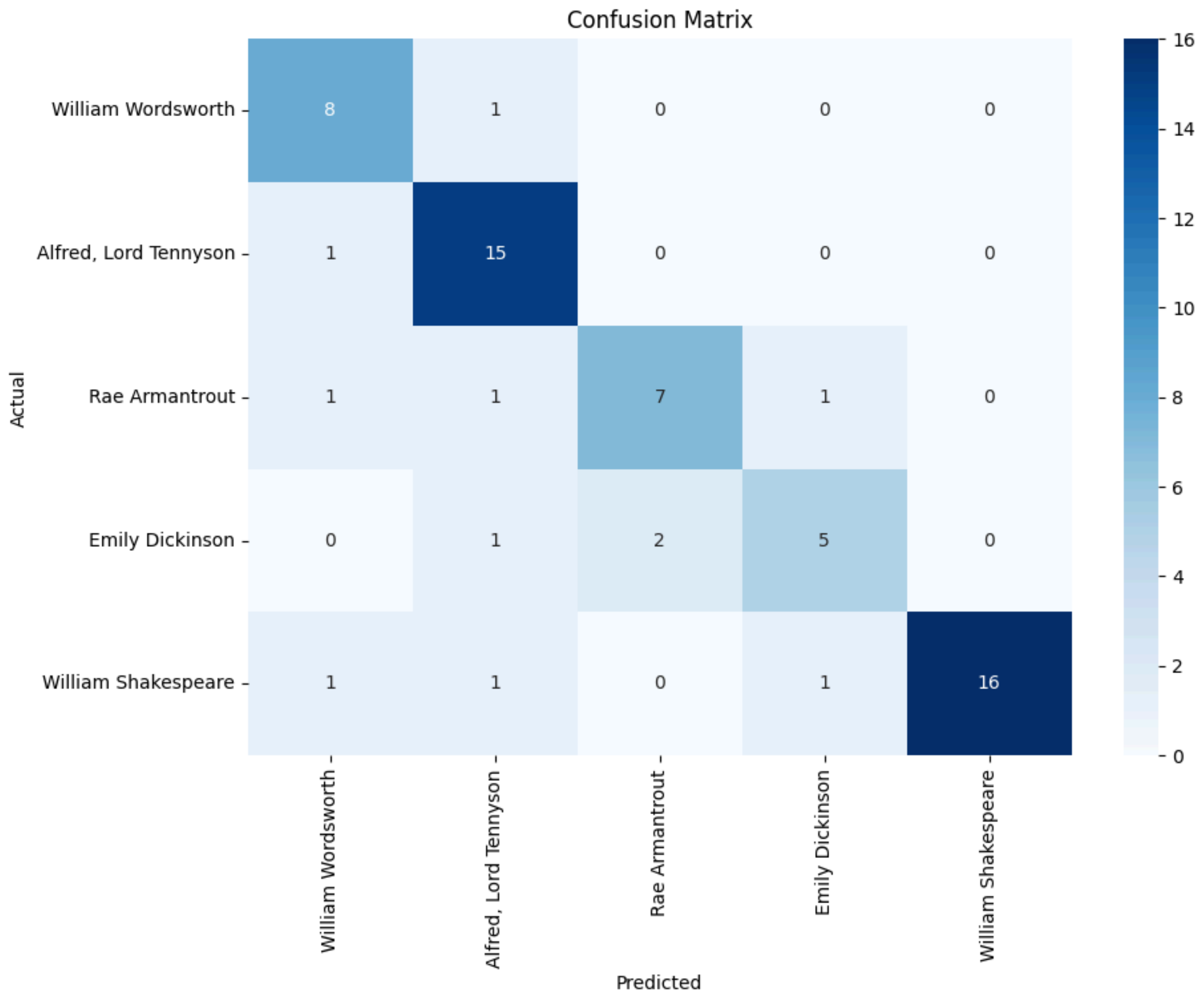


It worked! Our accuracy is up to 80% and both the training and testing loss level out with an increase in epochs. Lets create a confusion matrix to visualize the final model.

9b. Evaluating the TF-IDF Model

```
# Evaluate the Model
model.eval()
correct = 0
total = 0
y_pred = []
y_true = []

with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
        y_pred.extend(predicted.cpu().numpy())
        y_true.extend(labels.cpu().numpy())

accuracy = correct / total
print(f'Accuracy: {accuracy:.4f}')

# Calculate precision, recall, and F1 score
precision = precision_score(y_true, y_pred, average='weighted')
recall = recall_score(y_true, y_pred, average='weighted')
f1 = 2 * (precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1 Score: {f1:.4f}')

# Calculate the confusion matrix
cm = confusion_matrix(y_true, y_pred)

# Create a heatmap to visualize the confusion matrix
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=[index_to_author[i] for i in range(len(index_to_author))],
            yticklabels=[index_to_author[i] for i in range(len(index_to_author))])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

```
Accuracy: 0.8226
Precision: 0.8334
Recall: 0.8226
F1 Score: 0.8279
```



As show above, we have successfully plotted our data with an accuracy score of 82%, a recall of 82%, and a precision of 83%. This far exceeds my expectations after spending a dozen hours trying to get this dataset to cooperate.

10. Conclusion and Analysis

In conclusion, our attempt to classify poets based on their poems yielded a maximum accuracy of 80%, with several key observations and takeaways along the way.

First, when experimenting with Logistic Regression and the MLP (Multi-Layer Perceptron) models, both underperformed, achieving accuracies of around 35%. This result highlighted that simpler models might struggle to capture the complex patterns in the textual data, especially when dealing with high-dimensional inputs like word indices from poems. While these methods could classify basic features, they likely weren't sophisticated enough to grasp the nuances of different poets' styles or the variability across poems.

Next, we explored different word counts and varying the number of hidden layers in the MLP to see if either parameter could significantly improve performance. While adjusting word counts and network complexity showed marginal improvements, it was clear that simply increasing model depth or tweaking input sizes wasn't enough to boost accuracy in a meaningful way. This suggested that the raw word indices alone weren't providing sufficient distinguishing power for the classifier.

The real improvement came when I implemented TF-IDF vectorization to transform the poems into more informative features (thank you Paul for this suggestion!). By focusing on the most important words in the dataset and normalizing their frequencies, TF-IDF helped the MLP model to extract more meaningful patterns from the poems. This resulted in a consistent accuracy of 80%. This reinforces the importance of preprocessing and feature extraction in text classification tasks, where the raw text alone may not be enough for models to learn effectively.

This poet classification model could be used for tasks like organizing poetry collections, identifying anonymous authors, or verifying disputed works. It could help digital libraries or platforms like the Poetry Foundation by automatically sorting poems based on their writing style. Additionally, it could be applied in recommendation systems, matching readers with poets who write in a style they prefer. This approach makes processing large volumes of literary content faster and more efficient.