Author Attribution

Author: Andrew Sen

Reading and Manipulating Data

In [ ]:
```
import pandas as pd

# read data
df = pd.read_csv('federalist.csv')
df.author = df.author.astype('category')

# print count by author
df.author.value_counts()
```

Out[ ]:
```
HAMILTON                49
MADISON                 15
HAMILTON OR MADISON     11
JAY                      5
HAMILTON AND MADISON     3
Name: author, dtype: int64
```

In [ ]:
```
from sklearn.model_selection import train_test_split

# splitting data into train/test
X = df['text']
y = df.author
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

print('Dimensions:')
print(f'X_train: {str(X_train.shape)}')
print(f'y_train: {str(y_train.shape)}')
print(f'X_test: {str(X_test.shape)}')
print(f'y_test: {str(y_test.shape)}')
```

```
Dimensions:
X_train: (66,)
y_train: (66,)
X_test: (17,)
y_test: (17,)
```

In [ ]:
```
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer

# removing stopwords and performing tf-idf vectorization
stopwords = set(stopwords.words('english'))
vectorizer = TfidfVectorizer(stop_words=stopwords)
X_train_vectorized = vectorizer.fit_transform(X_train)
X_test_vectorized = vectorizer.transform(X_test)

print('New dimensions:')
print(f'X_train: {str(X_train_vectorized.shape)}')
print(f'X_test: {str(X_test_vectorized.shape)}')
```

```
New dimensions:
X_train: (66, 7876)
X_test: (17, 7876)
```

Classification

Naive Bayes

In [ ]:
```
from sklearn.naive_bayes import BernoulliNB
from sklearn.metrics import accuracy_score

nb = BernoulliNB()
nb.fit(X_train_vectorized, y_train)
pred_nb = nb.predict(X_test_vectorized)
print('Accuracy: ', accuracy_score(y_test, pred_nb))
```

```
Accuracy:  0.5882352941176471
```

To improve the results, we'll try different settings for the vectorizer.

In [ ]:
```
vectorizer = TfidfVectorizer(
    stop_words=stopwords,
    max_features=1000,
    ngram_range=(1, 2)
)
X_train_vectorized = vectorizer.fit_transform(X_train)
X_test_vectorized = vectorizer.transform(X_test)
```

In [ ]:
```
nb2 = BernoulliNB()
nb2.fit(X_train_vectorized, y_train)
pred_nb2 = nb2.predict(X_test_vectorized)
print('Accuracy: ', accuracy_score(y_test, pred_nb2))
```

```
Accuracy:  0.9411764705882353
```

The accuracy greatly improved when we decreased the number of features and added bigrams to the model. This is likely because differences between different authors' writing styles become more pronounced when bigrams are considered.

Logistic Regression

In [ ]:
```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression

# logistic regression with default settings
lr1 = Pipeline([
    ('tfidf', TfidfVectorizer()),
    ('logreg', LogisticRegression()),
])

lr1.fit(X_train, y_train)
pred_lr1 = lr1.predict(X_test)
print('Accuracy: ', accuracy_score(y_test, pred_lr1))
```

```
Accuracy:  0.5882352941176471
```

In [ ]:
```
# logistic regression with modified settings
lr2 = Pipeline([
    ('tfidf', TfidfVectorizer()),
    ('logreg', LogisticRegression(
        multi_class='multinomial',
        class_weight='balanced',
        solver='lbfgs'
    )),
])

lr2.fit(X_train, y_train)
pred_lr2 = lr2.predict(X_test)
print('Accuracy: ', accuracy_score(y_test, pred_lr2))
```

```
Accuracy:  0.9411764705882353
```

Of the two logistic regression models, the second performed far better. This is likely because setting the `class_weight` parameter to `balanced` allowed the model to adjust to the imbalanced frequencies of each author.

Neural Networks

In [ ]:
```python
from sklearn.neural_network import MLPClassifier

nn1 = Pipeline([
    ('tfidf', TfidfVectorizer()),
    ('neuralnet', MLPClassifier(
        solver='lbfgs',
        hidden_layer_sizes=(15, 7),
        random_state=1234,
        max_iter=1000
    )),
])

nn1.fit(X_train, y_train)
pred_nn1 = nn1.predict(X_test)
print('Accuracy: ', accuracy_score(y_test, pred_nn1))
```

```
Accuracy:  0.6470588235294118
```

In [ ]:
```python
nn2 = Pipeline([
    ('tfidf', TfidfVectorizer(
        ngram_range=(1,2)
    )),
    ('neuralnet', MLPClassifier(
        solver='lbfgs',
        hidden_layer_sizes=(15, 7),
        random_state=1234,
        max_iter=1000
    )),
])

nn2.fit(X_train, y_train)
pred_nn2 = nn2.predict(X_test)
print('Accuracy: ', accuracy_score(y_test, pred_nn2))
```

```
Accuracy:  0.7058823529411765
```

In [ ]:
```python
nn3 = Pipeline([
    ('tfidf', TfidfVectorizer(
        ngram_range=(1,2)
    )),
    ('neuralnet', MLPClassifier(
        solver='lbfgs',
        hidden_layer_sizes=(20, 15, 7),
        random_state=1234,
        max_iter=1000
    )),
])

nn3.fit(X_train, y_train)
pred_nn3 = nn3.predict(X_test)
print('Accuracy: ', accuracy_score(y_test, pred_nn3))
```

```
Accuracy:  0.7647058823529411
```

The best accuracy I could get using neural networks was ~76%. Adding more layers would result in the accuracy decreasing due to overfit, and adjusting the number of nodes in each layer seems to have little effect. This makes logistic regression and Bernoulli Naive Bayes the far better performers in my testing.