Author Attribution

This notebook explores sklearn and uses NLP algorithms to determine who wrote the Federalist Papers.

Step 1: Read in the csv

Convert the author column to categorical

```
1 import pandas as pd
```

```
1 fed_df = pd.read_csv('federalist.csv')
```

```
1 fed_df['author'] = fed_df['author'] .astype('category')
```

Display the first few rows of the data frame

```
1 fed_df.head()
```

	author	text
0	HAMILTON	FEDERALIST. No. 1 General Introduction For the...
1	JAY	FEDERALIST No. 2 Concerning Dangers from Forei...
2	JAY	FEDERALIST No. 3 The Same Subject Continued (C...
3	JAY	FEDERALIST No. 4 The Same Subject Continued (C...
4	JAY	FEDERALIST No. 5 The Same Subject Continued (C...

Display counts by author

```
1 fed_df.groupby('author').count()
```

author	text
HAMILTON	49
HAMILTON AND MADISON	3
HAMILTON OR MADISON	11
JAY	5
MADISON	15

Step 2: Divide into train and test

```
from sklearn.model_selection import train_test_split
```

```
text_X = fed_df['text']
author_y = fed_df['author']
```

```
X_train, X_test, y_train, y_test = train_test_split(text_X, author_y, test_size=0.2, train_size=0.8, ran
```

```
X_train.shape
```

```
(66,)
```

```
y_train.shape
```

```
(66,)
```

```
X_test.shape
```

```
(17,)
```

```
y_test.shape
```

```
(17,)
```

Step 3: Preprocess text data

```
from nltk.corpus import stopwords
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
import nltk
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
True
```

TF-IDF vectorize and remove stopwords

```
stopwords = set(stopwords.words('english'))
vectorizer = TfidfVectorizer(stop_words=stopwords)
```

```
X_train = vectorizer.fit_transform(X_train)
X_test = vectorizer.transform(X_test)
```

Output shape

```
X_train.shape
```

```
(66, 7876)
```

```
X_test.shape
```

```
(17, 7876)
```

Step 4: Bernoulli Naive Bayes Model

Accuracy: 58.82%

```
from sklearn.naive_bayes import BernoulliNB
```

```
model = BernoulliNB()
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
0.5882352941176471
```

Step 5: Enhanced Bernoulli Naive Bayes

Using only 1000 most frequent words and bigrams added as a feature

Accuracy: 94.12%

Much better!

```
# reset the test and train data
X_train, X_test, y_train, y_test = train_test_split(text_X, author_y, test_size=0.2, train_size=0.8, ran
```

```
vectorizer = TfidfVectorizer(stop_words=stopwords, max_features = 1000, ngram_range = (1,2))
```

```
X_train = vectorizer.fit_transform(X_train)
X_test = vectorizer.transform(X_test)
```

```
model = BernoulliNB()
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

```
0.9411764705882353
```

Step 6: Logistic Regression

```
from sklearn import linear_model
```

No Parameters

Accuracy: 58.82%

Same as the original Bernoulli Naive Bayes

```
1 X_train, X_test, y_train, y_test = train_test_split(text_X, author_y, test_size=0.2, train_size=0.8, ran
```

```
1 vectorizer = TfidfVectorizer(stop_words=stopwords)
```

```
1 X_train = vectorizer.fit_transform(X_train)
2 X_test = vectorizer.transform(X_test)
```

```
1 log_model = linear_model.LogisticRegression()
2 log_model.fit(X_train, y_train)
3 print(log_model.score(X_test, y_test))
```

```
    0.5882352941176471
```

With Parameters

Accuracy: 70.59%

Better when multiclass is specified and when the model is penalized for wrong guesses.

Accuracy: 76.47% if parameters are added to the vectorizer

```
1 X_train, X_test, y_train, y_test = train_test_split(text_X, author_y, test_size=0.2, train_size=0.8, ran
```

```
1 vectorizer = TfidfVectorizer(stop_words=stopwords, max_features = 1000, ngram_range = (1,2))
```

```
1 X_train = vectorizer.fit_transform(X_train)
2 X_test = vectorizer.transform(X_test)
```

```
1 log_model = linear_model.LogisticRegression(multi_class = 'multinomial', class_weight = 'balanced')
2 log_model.fit(X_train, y_train)
3 print(log_model.score(X_test, y_test))
```

```
    0.7647058823529411
```

Step 7: Neural Network

Accuracy: 82.35%

This was the highest percentage I could get by playing with the hidden layers and activation types for the nodes.

```
1 from sklearn.neural_network import MLPClassifier
```

```
1 X_train, X_test, y_train, y_test = train_test_split(text_X, author_y, test_size=0.2, train_size=0.8, ran
2 vectorizer = TfidfVectorizer(stop_words=stopwords, max_features = 1000, ngram_range = (1,2))
3 X_train = vectorizer.fit_transform(X_train)
4 X_test = vectorizer.transform(X_test)
```

```
1 X_train.shape
```

```
(66, 1000)
```

```
1 nn_model = MLPClassifier(solver='lbfgs', alpha=1e-5, hidden_layer_sizes=(33, 16), activation = 'relu')
2 nn_model.fit(X_train, y_train)
3 print(nn_model.score(X_test, y_test))
```

```
0.8235294117647058
```

```
1
```