1. Read in the csv file using pandas. Convert the author column to categorical data. Display the first few rows. Display the counts by author

```
import·pandas·as·pd
df·=·pd.read_csv('federalist.csv')
df['author']·=·df['author'].astype("category")
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...

```
df['author'].value_counts()
```

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

2. Divide into train and test, with 80% in train. Use random state 1234. Display the shape of train and test.

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

```
X·=·df['text']·#features
y·=·df['author']·#target
print(X)
```

```
0     FEDERALIST. No. 1 General Introduction For the...
1     FEDERALIST No. 2 Concerning Dangers from Forei...
2     FEDERALIST No. 3 The Same Subject Continued (C...
3     FEDERALIST No. 4 The Same Subject Continued (C...
4     FEDERALIST No. 5 The Same Subject Continued (C...
                            ...
78    FEDERALIST No. 79 The Judiciary Continued From...
```

```
79    FEDERALIST No. 80 The Powers of the Judiciary ...
80    FEDERALIST. No. 81 The Judiciary Continued, an...
81    FEDERALIST No. 82 The Judiciary Continued From...
82    FEDERALIST No. 83 The Judiciary Continued in R...
Name: text, Length: 83, dtype: object
```

```
# train test split
X_train,·X_test,·y_train,·y_test·=·train_test_split(X,·y,·test_size=0.2,·train_size=0.8,·ranc
```

```
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(66,)
(17,)
(66,)
(17,)
```

3. Process the text by removing stop words and performing tf-idf vectorization, fit to the training data only, and applied to train and test. Output the training set shape and the test set shape.

Double-click (or enter) to edit

```
from sklearn.feature_extraction.text import TfidfVectorizer
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
```

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

```
stopwords = set(stopwords.words('english'))
vectorizer = TfidfVectorizer(stop_words=stopwords) #tf-idf vectorization
```

```
X_train1 = vectorizer.fit_transform(X_train) # fit and transform the train data
X_test1 = vectorizer.transform(X_test)        # transform only the test data

print(X_train1.shape)
print(X_test1.shape)
```

```
(66, 7876)
(17, 7876)
```

4. Try a Bernoulli Naïve Bayes model. What is your accuracy on the test set? -> Got an accuracy of 58.8% as seen below

```
from sklearn.naive_bayes import BernoulliNB

naive_bayes = BernoulliNB()
naive_bayes.fit(X_train1, y_train)

     BernoulliNB()
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusic
# make predictions on the test data
pred = naive_bayes.predict(X_test1)
print('accuracy score: ', accuracy_score(y_test, pred))

     accuracy score:  0.5882352941176471
```

5. Redo the vectorization with max_features option set to use only the 1000 most frequent words. In addition to the words, add bigrams as a feature. Try Naïve Bayes again on the new train/test vectors and compare your results.

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

```
X_train2 = vectorizer.fit_transform(X_train) # fit and transform the train data
X_test2 = vectorizer.transform(X_test)        # transform only the test data
X_train2.shape

     (66, 1000)
```

```
naive_bayes = BernoulliNB()
naive_bayes.fit(X_train2, y_train)

     BernoulliNB()
```

```
pred = naive_bayes.predict(X_test2)
print('accuracy score: ', accuracy_score(y_test, pred))

     accuracy score:  0.5882352941176471
```

Changing the max_features and bigrams did not seem to affect the results, as I still get an accuracy of 58.8%

6. Try logistic regression -> without the max_features parameter. I get the same accuracy, although adding it and setting it to 1000 actually improves the accuracy this time to 82.3%

```
from sklearn.linear_model import LogisticRegression

vectorizer3 = TfidfVectorizer(binary=True, max_features= 1000)
X_train3 = vectorizer3.fit_transform(X_train)
X_test3 = vectorizer3.transform(X_test)
classifier = LogisticRegression(solver='lbfgs', class_weight='balanced')
classifier.fit(X_train3, y_train)
```

```
     LogisticRegression(class_weight='balanced')
```

```
pred3 = classifier.predict(X_test3)
print('accuracy score: ', accuracy_score(y_test, pred3))
```

```
     accuracy score:  0.8235294117647058
```

```
Try a neural network -> Adjusted layer sizes to increase the accuracy a bit.
Other changes that I tried did not seem to improve the accuracy much.
```

Try a neural network -> Adjusted layer sizes to increase the accuracy a bit. Other changes that I tried did not seem to improve the accuracy much.

```
from sklearn.neural_network import MLPClassifier


vectorizer4 = TfidfVectorizer(binary=True, max_features= 1000)
X_train4 = vectorizer4.fit_transform(X_train)
X_test4 = vectorizer4.transform(X_test)
classifier = MLPClassifier(solver='lbfgs', alpha=1e-5,
                   hidden_layer_sizes=(30, 2), random_state=1)
classifier.fit(X_train4, y_train)
```

```
     MLPClassifier(alpha=1e-05, hidden_layer_sizes=(30, 2), random_state=1,
                   solver='lbfgs')
```

```
pred = classifier.predict(X_test4)
print('accuracy score: ', accuracy_score(y_test, pred))
```

```
     accuracy score:  0.7647058823529411
```

✓ 0s completed at 9:04 PM ● ✕