Classifying Doctor's Note by Category of Ejection Fraction Measurement

Dependencies

In [1]:
```
import pandas as pd
import numpy as np
import time
import matplotlib.pyplot as plt
import seaborn as sns
import imblearn # for oversampling and undersampling

from sklearn.naive_bayes import MultinomialNB
from create_training_set import create_data
from sklearn.feature_extraction.text import CountVectorizer, TfidfV
ectorizer
#from sklearn import decomposition
#from scipy import linalg
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_score, recall
_score, f1_score, confusion_matrix
```

Import Data

In [2]:
```
start = time.time()
df = create_data()
end = time.time()
print("Time elapsed: ", end-start)
```

```
Operation complete.  Quitting.
Time elapsed:  70.43897533416748
```

Preprocessing

In [3]:
```
# Relabel Method from strings to a numerical representation (0 for
2d, 1 for 3d, and 2 for None)
df['METHOD'] = df['METHOD'].astype('category')
df['METHOD'] = df['METHOD'].cat.rename_categories({'2d simpson bipl
ane': 0, '3d imaging': 1, 'None': 2})
```

In [4]:
```
# Cut the set down to just the cleaned Note Text and to the Label
df = df[['NOTE_CLEAN', 'METHOD']]
# Separate out the labels
labels = np.array(df['METHOD'])
print("Shape of label vector: ", labels.shape)
print("Type: ", type(labels))
```

```
Shape of label vector:  (5056,)
Type:  <class 'numpy.ndarray'>
```

In [5]:
```
# Convert the features into a document term matrix

# Word Counts
# vectorizer = CountVectorizer(stop_words='english') #, tokenizer=L
emmaTokenizer())
# vectors = vectorizer.fit_transform(df['NOTE_CLEAN']).todense()

# TF-IDF
vectorizer_tfidf = TfidfVectorizer(stop_words='english')
vectors_tfidf = vectorizer_tfidf.fit_transform(df['NOTE_CLEAN']).to
dense() # (documents, vocab)

print("Shape of document term matrix: ", vectors_tfidf.shape)
print("Type: ", type(vectors_tfidf))
```

```
Shape of document term matrix:  (5056, 3478)
Type:  <class 'numpy.matrix'>
```

In [6]:
```
vocab = np.array(vectorizer_tfidf.get_feature_names())
```

In [7]:
```
vocab[100:120]
```

Out[7]:
```
array(['109', '10cm²', '10mm', '10mmhg', '10x', '11', '110', '111',
       '1110', '1114', '112', '112020', '1132', '114', '115', '1153
',
       '1155', '116', '118', '119'], dtype='<U19')
```

In [8]:
```
df['METHOD'].describe()
target = np.array(df['METHOD'])
```

Multinomial NBC

In [9]:
```
X_train, X_test, y_train, y_test = train_test_split(vectors_tfidf,
target)
```

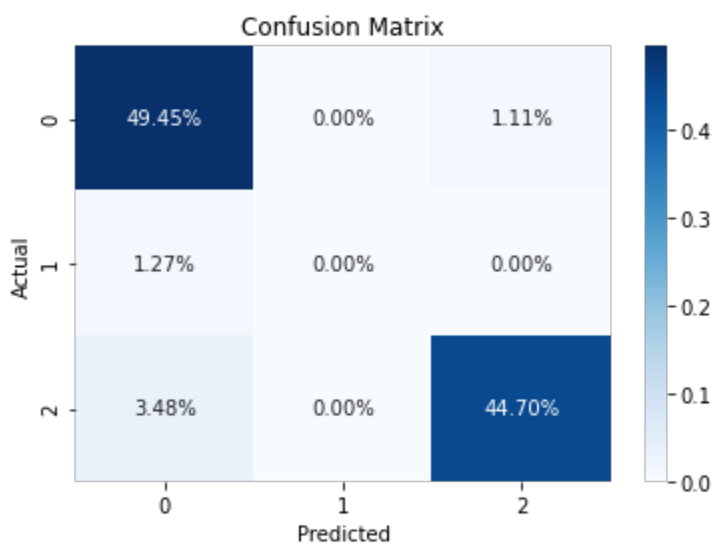
In [10]:
```
clf = MultinomialNB().fit(X_train, y_train)
```

In [11]:
```
# make predictions
yhat = clf.predict(X_test)
# evaluate predictions
acc = accuracy_score(y_test, yhat)
prec = precision_score(y_test, yhat, average='micro')
rec = recall_score(y_test, yhat, average='micro')
f1 = f1_score(y_test, yhat, average='micro')
print('Accuracy: %.3f' % acc)
print('Precision: %.3f' % prec)
print('Recall: %.3f' % rec)
print('F1 Score: %.3f' % f1)
```

```
Accuracy: 0.941
Precision: 0.941
Recall: 0.941
F1 Score: 0.941
```

In [12]:
```
cm = confusion_matrix(y_test, yhat)
sns.heatmap(cm/np.sum(cm), annot=True, fmt='.2%', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
```

Out[12]:
```
Text(0.5, 1.0, 'Confusion Matrix')
```

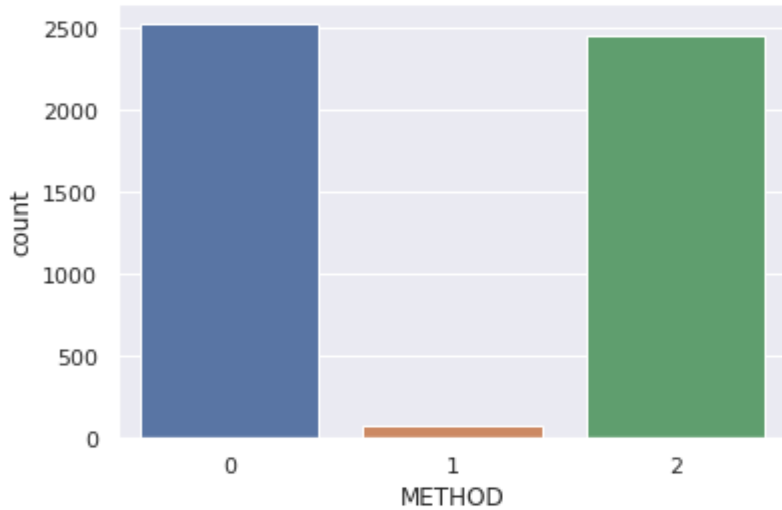


Analysis: The results show that we can use a classification algorithm to sort documents according to our defined categories of Ejection Fraction Measurement Methods. We can do this with remarkable accuracy, according to the metrics. However, upon examining the confusion matrix, it appears clear that are algorithm excels at correctly classify 2D Simpson Biplane documents (0) and None documents (2), but not 3D Imaging (1).

Examining a barplot of the counts of each methods tells us why. There is a severe class imbalance, and so we'll need to change our sampling method to adjust for this.

In [13]:
```
sns.set()
sns.countplot(x='METHOD', data=df)
```

Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x7f612f3d92e0>



Re-sampling to compensate for class imbalance

I will attempt to oversample Class 1, 3D Imaging, to a threshold of 50% of the training set. I'll then try and undersample the other classes so that they're more or less present in equal proportions. I was originally going to just oversample the minority class, but the barplot of the entire distribution of classes has me concerned that relying on oversampling alone would create an overfit model that wouldn't be useful as an applicable tool in the future. Even so, oversampling the minority class to 50% might still result in overfitting, so I may have to play with the proportions.

It may still be useful to try a straightforward oversampling anyway, just to compare results. I might even be surprised.

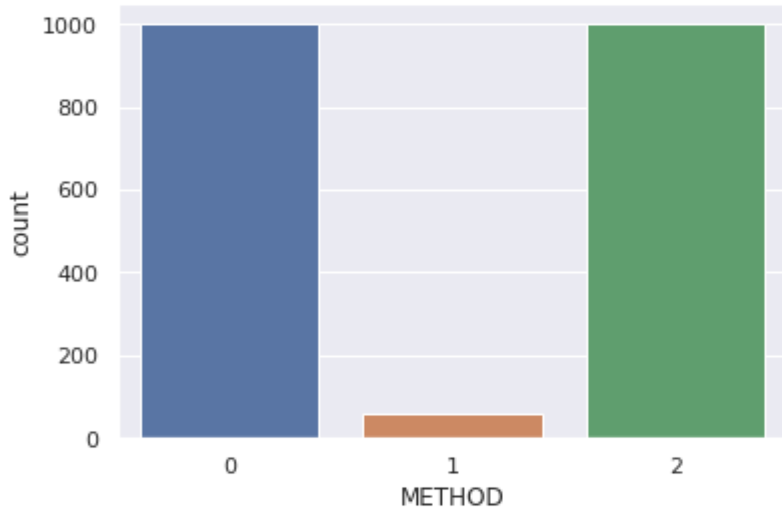
Undersample Majority Classes

Undersample Classes 0 and 2 to 1000 samples.

In [14]:
```
# Define undersampling strategy
undersample = imblearn.under_sampling.RandomUnderSampler(sampling_s
trategy={0: 1000, 2: 1000})
# Fit and apply the transform
X_under, y_under = undersample.fit_resample(X_train, y_train)
```

In [15]:
```
sns.countplot(x='METHOD', data = pd.DataFrame(y_under, columns=['ME
THOD']))
```

Out[15]: <matplotlib.axes._subplots.AxesSubplot at 0x7f611c223790>



Oversample Minority Class

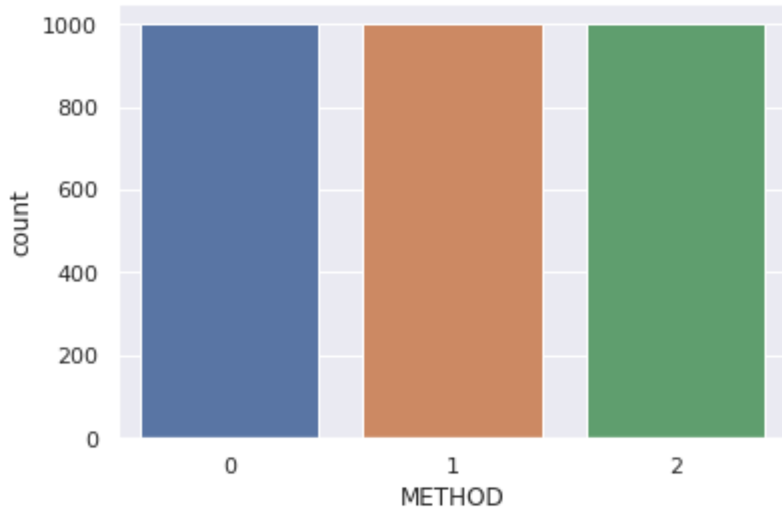
Oversample Class 1 to have the same number of samples as Classes 0 and 2 (1000 samples each).

In [16]:
```
# Define oversample strategy
oversample = imblearn.over_sampling.RandomOverSampler(sampling_stra
tegy='minority') # oversample minority class
# Fit and apply the transform
X_over, y_over = oversample.fit_resample(X_under, y_under)
```

Barplot of methods after re-sampling

In [17]:
```
sns.countplot(x='METHOD', data = pd.DataFrame(y_over, columns=['METHOD']))
```

Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x7f611c21d850>



Create a new classifier with the resampled data

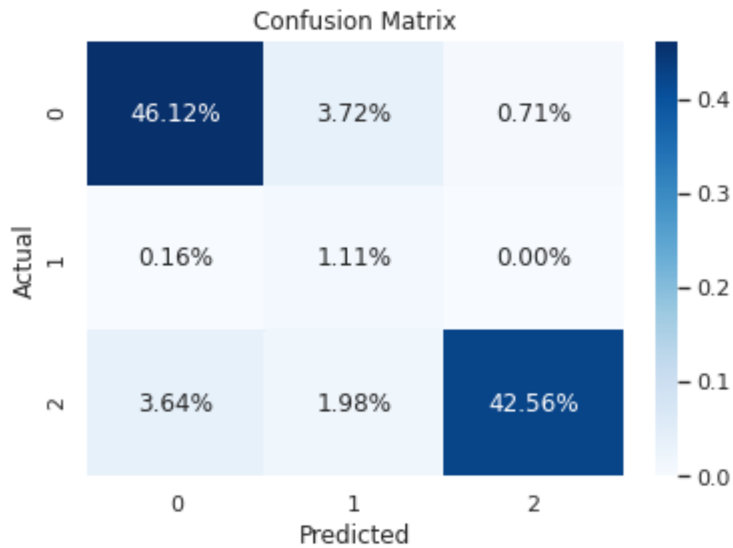
In [18]:
```
clf = MultinomialNB().fit(X_over, y_over)
```

In [19]:
```
# make predictions
yhat = clf.predict(X_test)
# evaluate predictions
acc = accuracy_score(y_test, yhat)
prec = precision_score(y_test, yhat, average='micro')
rec = recall_score(y_test, yhat, average='micro')
f1 = f1_score(y_test, yhat, average='micro')
print('Accuracy: %.3f' % acc)
print('Precision: %.3f' % prec)
print('Recall: %.3f' % rec)
print('F1 Score: %.3f' % f1)
```

```
Accuracy: 0.898
Precision: 0.898
Recall: 0.898
F1 Score: 0.898
```

In [20]:
```
cm = confusion_matrix(y_test, yhat)
sns.heatmap(cm/np.sum(cm), annot=True, fmt='.2%', cmap='Blues')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
```

Out[20]:
```
Text(0.5, 1.0, 'Confusion Matrix')
```



Analysis

The good news is that according to the confusion matrix, we are now able to correctly classify some documents as "3D Imaging". It's hard to tell how good we are at doing this, since so few examples of 3D Imaging exists. One way to get a good idea of how close we are to accurately classifying documents that incorporated 3D Imaging as a measurement technique would be to look at the actual proportion of 3D Imaging documents in the original dataset. If they are comparable, it's a good sign that we're on the right track.

In [21]:
```
proportion_class1 = len(df['METHOD'] == 1)/ len(df['METHOD'])
print('Percentage of Class 1, 3D Imaging, in set: %.2f' % proportio
n_class1)
```

```
Percentage of Class 1, 3D Imaging, in set: 1.00
```

It looks like 3D Imaging only made up about 1% of the original data, which is comparable to the 1.03% we classified when using the test set. Why is the percentage slightly higher? If I had to guess, it is because the splits are random samples, so the actual proportion of observations aren't going to be necessarily the same as the whole set. In other words, when the testing sample was created at the start, it could have had a slightly higher proportion of observations labeled as 3D Imaging than in the training sample, which would explain why the testing sample has a higher proportion of 3D Imaging observations than the parent sample.

Next Steps

One thing I could try next would be to duplicate this process using other classification algorithms to compare and contrast performance. Multinomial Logistic Regression, Decision Trees/Random Forests, or employing AdaBoost to increase performance. I could even tinker with Neural Networks, although I am not sure the need justifies such a powerful tool.

I should also run the model on samples and actually compare the output with the corresponding original text with my own eyes. I'll need to write code to attach the predictions to a table of samples that includes the text and the classification labels.

I would also like to try pickling the model and incorporating it into a Django application. Incorporating it into an app would be great not just as an educational activity, but it could also turn it into a useable tool for others at HHC.

It may also be useful to apply the lessons I have learned here to create a binary classifier for documents that are focused on ejection fration and documents that are not. Conceivably, the two models could work together in a sort of pipeline. First, we sort documents by mention of ejection fraction. Then, among the documents remaining, we label according to measurement method. Isolating the measurement alone would still require some hardcoded pattern matching relying on some combination of SpaCy, NLTK, and/or regex.

In [ ]: