Spooky author identification

I'm going to load the train data, consisting of text extracts from novels written by 3 different authors (Edgar Allan Poe, HP. Lovecraft and Mary Shelley), identified by author's name. The idea is to train a model (using Natural Language Processing methods) to recognize the authors' styles and apply the model to the test data (unidentified author names).

Import and read

In [1]:
```
import pandas as pd
import numpy as np
from textblob import TextBlob
from nltk import *
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
import gensim

#to plot inside the document
%matplotlib inline
import matplotlib.pyplot as plt
```

In [2]:
```
train = pd.read_csv("G:/My Drive/Formation ENI/Datasets/Spooky Author Identifi
cation/train.csv")
train.head()
```

Out[2]:

	id	text	author
0	id26305	This process, however, afforded me no means of...	EAP
1	id17569	It never once occurred to me that the fumbling...	HPL
2	id11008	In his left hand was a gold snuff box, from wh...	EAP
3	id27763	How lovely is spring As we looked from Windsor...	MWS
4	id12958	Finding nothing else, not even gold, the Super...	HPL

In [3]:
```
train.describe()
```

Out[3]:

	id	text	author
count	19579	19579	19579
unique	19579	19579	3
top	id08039	For a moment only did I lose recollection; I f...	EAP
freq	1	1	7900

So we're working with 19,579 extracts from 3 different authors.

Tokenizing

To be able to study the pieces of text, I'll break them down (tokenize) by words. I'll also break them down by sentences to see if we only have sentence-sized extracts or if some are multiple-sentenced.

The first thing I need to do is instantiate the columns I'll be using to store the information, and change their dtype to object (otherwise, Pandas returns an error).

In [4]:
```
train['tokens'] = None
train['sentences'] = None
train['nb_tokens'] = None
train['word_length'] = None
train['nb_sentences'] = None
train[['tokens', 'sentences']] = train[['tokens', 'sentences']].astype('objec
t')
```

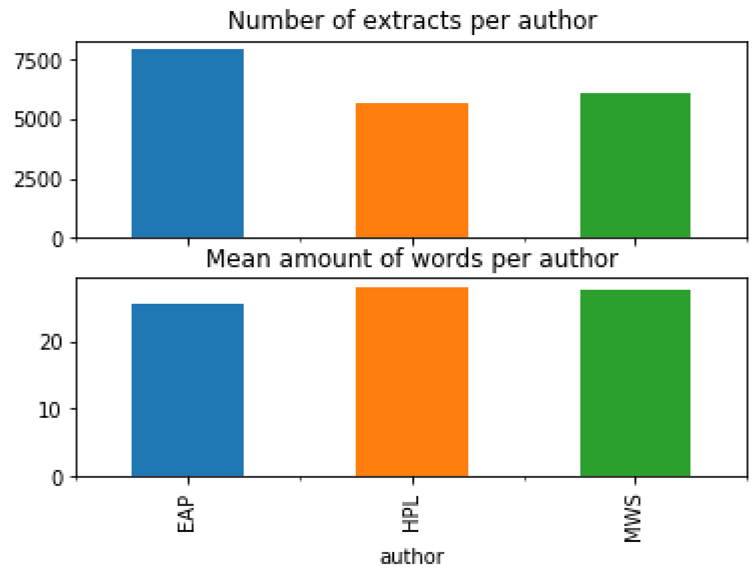
In [5]:
```
for i, row in train.iterrows():
    train.loc[i, 'tokens'] = TextBlob(row['text'].lower()).words
    train.loc[i, 'nb_tokens'] = len(train.loc[i, 'tokens'])
    train.loc[i, 'word_length'] = np.mean([len(x) for x in train.loc[i, 'token
s']])
    train.loc[i, 'sentences'] = TextBlob(row['text']).sentences
    train.loc[i, 'nb_sentences'] = len(train.loc[i, 'sentences'])
```

Distribution and average words

In [6]:
```
grouped = pd.DataFrame({'nb_words':train.groupby('author')['nb_tokens'].aggreg
ate(np.sum)})
grouped['nb_extracts'] = train.groupby('author')['nb_tokens'].aggregate('coun
t')
grouped['mean_words'] = grouped['nb_words'] / grouped['nb_extracts']
```

In [7]:
```
f, axarr = plt.subplots(2, sharex=True)
grouped['nb_extracts'].plot(kind='bar', ax=axarr[0])
axarr[0].set_title('Number of extracts per author')
grouped['mean_words'].plot(kind='bar', ax=axarr[1])
axarr[1].set_title('Mean amount of words per author')
```

Out[7]: Text(0.5, 1.0, 'Mean amount of words per author')



We can observe that the extracts are not entirely equally distributed (Poe has about 2000 more extracts than Lovecraft and Shelley).

On the contrary, Poe uses the least amount of words in his extracts (on average), while Shelley and Lovecraft have the same average.

Stopwords

Remove parasitic words ("the", "and", etc.) from tokens to avoid being flooded with them in analysis.

In [8]:
```
train['useful'] = None
train['useful'] = train['useful'].astype('object')

stop_words = stopwords.words('english')
stop_words.extend(['one', "'s"])
for i, row in train.iterrows():
    train.loc[i, 'useful'] = [x for x in row['tokens'] if x not in stop_words]
```

Part of Speech and Lemmatizing

I need to tag each word in the sentences to know what part of speech they are (verb, subject, adverb, etc.)

In [9]:
```
train['tags'] = train.apply(lambda row: pos_tag(row['useful']), axis = 1)
```

Unhappily, lemmatizing does not function with the same format of tags as the ones generated by pos_tag. I need to make a function to transform the tags to something that can be lemmatized.

In [10]:
```
def translate_tag_pos(tuple):
    if tuple[1].startswith('N'):
        new_tuple = (tuple[0], 'n')
    else:
        if tuple[1].startswith('V'):
            new_tuple = (tuple[0], 'v')
        else:
            if tuple[1].startswith('R'):
                new_tuple = (tuple[0], 'r')
            else:
                if tuple[1].startswith('J'):
                    new_tuple = (tuple[0], 'a')
                else:
                    return None
    return new_tuple

def lemmatize_with_new_tags(tags):
    lemmas = []
    for t in tags:
        new_tag = translate_tag_pos(t)
        if new_tag is None:
            lemmas.append(t[0])
        else:
            lemmas.append(wordnet_lemmatizer.lemmatize(new_tag[0], pos=new_tag
[1]))
    return lemmas
```

Lemmatizing is grouping words by their root, so that plural occurences of a word are not separated from the singular ones, or conjugated verbs spread out. This allows more consistency in understanding texts.

In [11]:
```
wordnet_lemmatizer = WordNetLemmatizer()

train['lemma'] = train.apply(lambda row: lemmatize_with_new_tags(row['tags']),
axis=1)
```

I'll measure the wealth of vocabulary in each extract, by dividing the number of unique words (stopwords excluded) by the total number of words in the extract.

In [12]:
```
train['vocab_wealth'] = train.apply(lambda row: len(set(row['lemma'])) / len(r
ow['tokens']) * 100, axis=1)
```

Wordclouds

I'll create a wordcloud image for each author. For this, I'll split the texts by author, generate a mask with images I've downloaded, and plot the wordcloud on those masks.

In [13]:
```
from wordcloud import WordCloud
from PIL import Image
from scipy.misc import imread


EAP = [item for sublist in train[train.author=="EAP"]['lemma'].values for item
in sublist]
HPL = [item for sublist in train[train.author=="HPL"]['lemma'].values for item
in sublist]
MWS = [item for sublist in train[train.author=="MWS"]['lemma'].values for item
in sublist]
```

In [14]:
```
poe_mask = np.array(Image.open("G:/My Drive/Formation ENI/Datasets/Spooky Auth
or Identification/poe.jpg"))
lovecraft_mask = np.array(Image.open("G:/My Drive/Formation ENI/Datasets/Spook
y Author Identification/cthulhu.jpg"))
shelley_mask = np.array(Image.open("G:/My Drive/Formation ENI/Datasets/Spooky
 Author Identification/frankenstein.jpg"))
```

In [15]:
```
masks = [
    {'title':'Edgar Allan Poe (a portrait)', 'save_name': "poe_wc.png", 'mask'
: poe_mask, 'text': ' '.join(EAP)},
    {'title':'HP Lovecraft (Cthulhu)', 'save_name': "cthulhu_wc.png", 'mask':
lovecraft_mask, 'text': ' '.join(HPL)},
    {'title':'Mary Shelley (Frankenstein)', 'save_name': "frankenstein_wc.png"
, 'mask': shelley_mask, 'text': ' '.join(MWS)}
]
```

In [ ]:
```
for m in masks:
    wc = WordCloud(background_color="white", max_words=2000, mask=m['mask'])
    # generate word cloud
    wc.generate(m['text'])

    # store to file
    wc.to_file(m['save_name'])

    # show
    plt.imshow(wc, interpolation='bilinear')
    plt.axis("off")
    plt.title(m['title'])
    plt.figure()
    plt.show()
```

We can see with these wordclouds that their semantic spaces are quite different.

I'll now take this to my advantage and create a table of their vocabulary frequencies.

Word frequency

I'll save the frequency of words (and successive couples of words) in each authors' corpuses, and use this as a measure of probability that new extracts are written by them.

In [17]:
```
hpl_freq = FreqDist(HPL)
poe_freq = FreqDist(EAP)
mws_freq = FreqDist(MWS)
```

In [18]:
```
hpl_bg_freq = FreqDist(bigrams(HPL))
poe_bg_freq = FreqDist(bigrams(EAP))
mws_bg_freq = FreqDist(bigrams(MWS))
```

Now I'll mark each extract with the proportion of words from each authors that they use.

In [19]:
```
for i, row in train.iterrows():
    train.loc[i, 'hpl_word_count'] = np.sum([hpl_freq[word] for word in train.
loc[i, 'lemma']]) / len(train.loc[i, 'lemma'])
    train.loc[i, 'poe_word_count'] = np.sum([poe_freq[word] for word in train.
loc[i, 'lemma']]) / len(train.loc[i, 'lemma'])
    train.loc[i, 'mws_word_count'] = np.sum([mws_freq[word] for word in train.
loc[i, 'lemma']]) / len(train.loc[i, 'lemma'])

    bg = list(bigrams(train.loc[i, 'lemma']))
    train.loc[i, 'hpl_bigram_count'] = np.sum([hpl_freq[bigram] for bigram in
bg]) / len(bg)
    train.loc[i, 'poe_bigram_count'] = np.sum([poe_freq[bigram] for bigram in
bg]) / len(bg)
    train.loc[i, 'mws_bigram_count'] = np.sum([mws_freq[bigram] for bigram in
bg]) / len(bg)
```

```
c:\users\berder\appdata\local\programs\python\python37\lib\site-packages\ipyk
ernel_launcher.py:7: RuntimeWarning: invalid value encountered in double_scal
ars
  import sys
c:\users\berder\appdata\local\programs\python\python37\lib\site-packages\ipyk
ernel_launcher.py:8: RuntimeWarning: invalid value encountered in double_scal
ars

c:\users\berder\appdata\local\programs\python\python37\lib\site-packages\ipyk
ernel_launcher.py:9: RuntimeWarning: invalid value encountered in double_scal
ars
  if __name__ == '__main__':
c:\users\berder\appdata\local\programs\python\python37\lib\site-packages\ipyk
ernel_launcher.py:2: RuntimeWarning: invalid value encountered in double_scal
ars

c:\users\berder\appdata\local\programs\python\python37\lib\site-packages\ipyk
ernel_launcher.py:3: RuntimeWarning: invalid value encountered in double_scal
ars
  This is separate from the ipykernel package so we can avoid doing imports u
ntil
c:\users\berder\appdata\local\programs\python\python37\lib\site-packages\ipyk
ernel_launcher.py:4: RuntimeWarning: invalid value encountered in double_scal
ars
  after removing the cwd from sys.path.
```

A few extracts are only composed of stopwords, which means no lemmas are found. Therefore, the two types of ratio above return NaN, which I need to replace by 0.

In [20]:
```
train.loc[pd.isnull(train['hpl_word_count']), ['hpl_word_count', 'poe_word_cou
nt', 'mws_word_count']] = 0
train.loc[pd.isnull(train['hpl_bigram_count']), ['hpl_bigram_count', 'poe_bigr
am_count', 'mws_bigram_count']] = 0
```

Model definition and Validation

In [21]:
```
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

# First transform the author name into a numeric code (0, 1 and 2)
encoder = preprocessing.LabelEncoder()
train['target'] = encoder.fit_transform(train['author'])


# Then split the train in two for a test
x_train, x_validate, y_train, y_validate = train_test_split(train[['nb_tokens'
, 'word_length', 'nb_sentences', 'vocab_wealth',
                                                                    'hpl_word_c
ount', 'poe_word_count', 'mws_word_count',
                                                                    'hpl_bigram
_count', 'poe_bigram_count', 'mws_bigram_count']],
                                                    train['target'],
                                                    train_size=0.75, test_size
=0.25)

# Train the model
rfc_class = RandomForestClassifier(n_estimators=800, max_depth=20, max_feature
s='sqrt').fit(x_train, y_train)

# Make predictions on the test sample and probabilities
prediction = pd.Series(rfc_class.predict(x_validate), index=x_validate.index)
probabilities = pd.DataFrame(rfc_class.predict_proba(x_validate), columns=["EA
P", "HPL", "MWS"], index=x_validate.index)


validate = train.filter(x_validate.index, axis=0)
validate[["EAP", "HPL", "MWS"]] = probabilities
validate['target_predict'] = prediction
validate['predicted_author'] = encoder.inverse_transform(validate['target_pred
ict'])
```

Performance evaluation

In [22]:
```
from sklearn.metrics import classification_report

print(classification_report(y_true= validate.author, y_pred=validate.predicted
_author))
```

```
              precision    recall  f1-score   support

         EAP       0.70      0.77      0.73      1949
         HPL       0.72      0.69      0.71      1411
         MWS       0.72      0.66      0.69      1535

    accuracy                           0.71      4895
   macro avg       0.71      0.71      0.71      4895
weighted avg       0.71      0.71      0.71      4895
```

Application to the test dataset

Now, all that's left to do is to apply all the same preprocessing to the test dataset, and make predictions based on the type of model we just trained on the train dataset (which we'll retrain on the overall dataset without the split).

In [26]:
```
test = pd.read_csv('G:/My Drive/Formation ENI/Datasets/Spooky Author Identific
ation/test.csv')
test['tokens'] = None
test['sentences'] = None
test['useful'] = None
test['tags'] = None
test['lemma'] = None
test['word_length'] = None
test['vocab_wealth'] = None
test['hpl_word_count'] = None
test['poe_word_count'] = None
test['mws_word_count'] = None
test['hpl_bigram_count'] = None
test['poe_bigram_count'] = None
test['mws_bigram_count'] = None


test = test.astype('object')
```

In [ ]:
```
for i, row in test.iterrows():
    tokens = TextBlob(row['text'].lower()).words
    test['tokens'][i] = tokens
    test.loc[i, 'nb_tokens'] = len(tokens)
    test.loc['word_length', i] = np.mean([len(x) for x in tokens])
    sentences = TextBlob(test.loc[i, 'text']).sentences
    test.loc['sentences', i] = sentences
    test.loc[i, 'nb_sentences'] = len(sentences)
    useful = [x for x in tokens if x not in stop_words]
    test.loc['useful', i] = useful
    tags = pos_tag(useful)
    test.['tags', i] = tags
    lemmas = lemmatize_with_new_tags(tags)
    test.loc['lemma', i] = lemmas
    test.loc['vocab_wealth', i] = len(set(lemmas)) / len(tokens) * 100
    test.loc['hpl_word_count', i] = np.sum([hpl_freq[word] for word in lemmas
]) / len(lemmas)
    test.loc['poe_word_count', i] = np.sum([poe_freq[word] for word in lemmas
]) / len(lemmas)
    test.loc['mws_word_count', i] = np.sum([mws_freq[word] for word in lemmas
]) / len(lemmas)
    bg = list(bigrams(lemmas))
    test.loc['hpl_bigram_count', i] = np.sum([hpl_freq[bigram] for bigram in b
g]) / len(bg)
    test.loc['poe_bigram_count', i] = np.sum([poe_freq[bigram] for bigram in b
g]) / len(bg)
    test.loc['mws_bigram_count', i] = np.sum([mws_freq[bigram] for bigram in b
g]) / len(bg)

test.loc[pd.isnull(test['hpl_word_count']), ['hpl_word_count', 'poe_word_coun
t', 'mws_word_count']] = 0
test.loc[pd.isnull(test['hpl_bigram_count']), ['hpl_bigram_count', 'poe_bigram
_count', 'mws_bigram_count']] = 0
```

Retrain the model on the whole training dataset (re-integration of the validation dataset)

In [ ]:
```
rfc_class = RandomForestClassifier(n_estimators=800,
                                   max_depth=20,
                                   max_features='sqrt').fit(X=train[['nb_token
s', 'word_length', 'nb_sentences', 'vocab_wealth',
                                                                    'hpl_word_c
ount', 'poe_word_count', 'mws_word_count',
                                                                    'hpl_bigram
_count', 'poe_bigram_count', 'mws_bigram_count']],
                                                             y=train['target'])
```

In [ ]:
```
prediction = pd.Series(rfc_class.predict(test[['nb_tokens', 'word_length', 'nb
_sentences', 'vocab_wealth',
                                               'hpl_word_count', 'poe_word_cou
nt', 'mws_word_count',
                                               'hpl_bigram_count', 'poe_bigram
_count', 'mws_bigram_count']]), index=test.index)
probabilities = pd.DataFrame(rfc_class.predict_proba(test[['nb_tokens', 'word_
length', 'nb_sentences', 'vocab_wealth',
                                                           'hpl_word_count',
'poe_word_count', 'mws_word_count',
                                                           'hpl_bigram_count',
'poe_bigram_count', 'mws_bigram_count']]),
                             columns=["EAP", "HPL", "MWS"], index=test.index)


test[["EAP", "HPL", "MWS"]] = probabilities
test['target_predict'] = prediction
test['predicted_author'] = encoder.inverse_transform(test['target_predict'])
test.head(4)
```

In [ ]:
```
test[['id', 'EAP', 'HPL', 'MWS']].to_csv("submission.csv", index=False, sep=
',')
```