Implement bag of words and tf-idf using naive Bayes algorithm check the accuracy

Load data

In [ ]:
```
#Installing pyLDAvis
#to interpret the topics in a topic model that has been fit to a corpus of text data
!pip install pyLDAvis
```

In [ ]:
```
# import libraries
import pandas as pd
import numpy as np

import os

# to make this notebook's output stable across runs
np.random.seed(42)
import seaborn as sns
import matplotlib
import matplotlib.pyplot as plt

import re

# Gensim
import gensim
import gensim.corpora as corpora
from gensim.utils import simple_preprocess
from gensim.models import CoherenceModel

# Plotting tools
import pyLDAvis
import pyLDAvis.gensim

# set options for rendering plots
%matplotlib inline

# display multiple outputs within a cell
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all";

import warnings
warnings.filterwarnings('ignore');
```

In [ ]:
```
#Loading data
train = pd.read_csv(r'train.csv')
test = pd.read_csv(r'test.csv')
```

Data exploration

In [ ]:
```
#Fetching data
train.head()
train.shape
train["author"].value_counts()
```

Three Authors

Edgar Allen Poe
Mary Wollstonecraft Shelley
H.P Lovecraft

In [ ]:
```
#to display the count of categorical observations in each bin in the dataset
sns.countplot('author', data = train, palette='dark');
```

In [ ]:
```
# look at some of the writing from edgar allen poe
train[train["author"] == "EAP"]["text"][0]
```

In [ ]:
```
# add a rough count of words in the sentences as a feature
train['word_length'] = train.text.str.count(' ')
train.head()
```

In [ ]:
```
# look at some of the writing from edgar allen poe
train[train["author"] == "EAP"]["word_length"].describe()
```

In [ ]:
```
plt.figure(figsize=(8,8))
sns.boxplot(x="author", y="word_length", data=train, palette='bright');
```

In [ ]:
```
plt.figure(figsize=(10,5))
sns.set_context('talk')
sns.distplot(train['word_length'], color='black');
```

In [ ]:
```
f,ax=plt.subplots(1,3,figsize=(16,6));
sns.distplot(train[train['author']=='EAP'].word_length,ax=ax[0]);
ax[0].set_title('Edgar Allen Poe');
sns.distplot(train[train['author']=='HPL'].word_length,ax=ax[1], color='r')
ax[1].set_title('H.P. Lovecraft');
sns.distplot(train[train['author']=='MWS'].word_length,ax=ax[2], color='g')
ax[2].set_title('Mary Shelley');
plt.show();
```

In [ ]:
```
train[train["author"] == "HPL"]["word_length"].describe()
```

In [ ]:
```
train[train["author"] == "MWS"]["word_length"].describe()
```

Preprocessing and feature extraction

stop words and tokens

In [ ]:
```
import nltk
nltk.download('stopwords')

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

data = train["text"]

stop_words = set(stopwords.words('english'))

word_tokens = word_tokenize(data[1])

filtered_sentence = [w for w in word_tokens if not w in stop_words]

print(word_tokens)
print(filtered_sentence)
#finding stopwords,tokenizing and filtering them
```

In [ ]:
```
def sent_to_words(sentences):
    for sentence in sentences:
        yield(gensim.utils.simple_preprocess(str(sentence), deacc=True))  # deacc=True removes punctuations

data_words = list(sent_to_words(data))

print(data_words[:1])
```

In [ ]:
```
# flatten list and join together as a string
flat_list = [item for sublist in data_words for item in sublist]
str1 = ' '.join(flat_list)
```

bigram, trigram

In [ ]:
```
# Build the bigram and trigram models
bigram = gensim.models.Phrases(data_words, min_count=5, threshold=100) # higher threshold fewer phrases.
trigram = gensim.models.Phrases(bigram[data_words], threshold=100)

# Faster way to get a sentence clubbed as a trigram/bigram
bigram_mod = gensim.models.phrases.Phraser(bigram)
trigram_mod = gensim.models.phrases.Phraser(trigram)

print(trigram_mod[bigram_mod[data_words[0]]])
```

In [ ]:
```
bigrams = []
for phrase in bigram.export_phrases(data_words[:100]):
    bigrams.append(phrase)
bigrams[:10]
```

In [ ]:
```
# Define functions for stopwords, bigrams, trigrams and lemmatization
def remove_stopwords(texts):
```

```
    return [[word for word in simple_preprocess(str(doc)) if word not in stop_words] for doc in texts]

train_text = remove_stopwords(train['text'])
test_text = remove_stopwords(test['text'])
```

bag of words

In [ ]:
```
train_text = [' '.join(sent) for sent in train_text]
test_text = [' '.join(sent) for sent in test_text]
```

In [ ]:
```
from sklearn.feature_extraction.text import CountVectorizer

# Initialize the "CountVectorizer" object, which is scikit-learn's
# bag of words tool.
vectorizer = CountVectorizer(analyzer = "word",   \
                             tokenizer = None,    \
                             preprocessor = None, \
                             stop_words = None,   \
                             max_features = 6000)

feature_vec = vectorizer.fit_transform(train_text)
```

tf-idf

In [ ]:
```
from sklearn.feature_extraction.text import TfidfTransformer
tf_transformer = TfidfTransformer(use_idf=True).fit(feature_vec)
X_train_tf = tf_transformer.transform(feature_vec)
X_train_tf.shape
```

Training Classifier Models

In [ ]:
```
# create train/test set
train_data = train_text
train_labels = train["author"]
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test=train_test_split(train_data,train_labels,test_size=0.20,random_state=0)
```

In [ ]:
```
# feature processing pipeline
from sklearn.pipeline import Pipeline

text_features = Pipeline([
    ('vect', CountVectorizer()),
    ('tfidf', TfidfTransformer()),
])

text_features.fit_transform(X_train)
```

Multinomial Naive Bayes

In [ ]:
```
# for classification with discrete features
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, log_loss

pipe = Pipeline([
    ('features', text_features),
    ('clf', MultinomialNB()),
])

pipe.fit(X_train, y_train)

nb_pred = pipe.predict(X_test)
nb_probs = pipe.predict_proba(X_test)

print("Accuracy score: " + str(accuracy_score(y_test, nb_pred)))
print("Log loss: " + str(log_loss(y_test, nb_probs)));
```

In [ ]:
```
#performing multiple operations (e.g., calling function after function) in a sequence,
#for each element of an iterable, in such a way that the output of each element is the input of the next
pipe.get_params().keys()
```

In [ ]:
```
# Grid Search
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
log_loss_build = make_scorer(log_loss, greater_is_better=False, needs_proba=True)

parameters = {'features__vect__max_features': [10000, 12000, 15000],
```

```
              'features__vect__ngram_range': [(1,1), (1,2)],
              'features__tfidf__use_idf': [True, False],
              'clf__alpha': [0.01, 0.1, 1]
             }

gs = GridSearchCV(pipe, parameters, cv=5, n_jobs=-1, scoring=log_loss_build)

# Fit and tune model
gs.fit(X_train, y_train);
```

In [ ]:
```
gs.best_params_
gs.best_score_
final_model = gs.best_estimator_

final_model.fit(X_train, y_train)
final_pred = final_model.predict(X_test)
final_probs = final_model.predict_proba(X_test);

print("Accuracy score: " + str(accuracy_score(y_test, final_pred)))
print("Log loss: " + str(log_loss(y_test, final_probs)))
```

Complement NB

In [ ]:
```
#to correct the severe assumptions made by the standard Multinomial Naive Bayes classifier
from sklearn.naive_bayes import ComplementNB

pipe = Pipeline([
    ('vect', CountVectorizer(max_features=10000, ngram_range=(1,2))),
    ('tfidf', TfidfTransformer(use_idf=False)),
    ('clf', ComplementNB(alpha=0.01))
])

pipe.fit(X_train, y_train)
cnb_pred = pipe.predict(X_test)
cnb_probs = pipe.predict_proba(X_test);

print("Accuracy score: " + str(accuracy_score(y_test, cnb_pred)))
print("Log loss: " + str(log_loss(y_test, cnb_probs)));
```

Random Forest

In [ ]:
```
#combines the output of multiple decision trees to reach a single result
from sklearn.ensemble import RandomForestClassifier

pipe = Pipeline([
    ('vect', CountVectorizer(max_features=10000)),
    ('tfidf', TfidfTransformer(use_idf=False)),
    ('clf', RandomForestClassifier(n_estimators = 50))
])

pipe.fit(X_train, y_train)
rf_pred = pipe.predict(X_test)
rf_probs = pipe.predict_proba(X_test);

print("Accuracy score: " + str(accuracy_score(y_test, rf_pred)))
print("Log loss: " + str(log_loss(y_test, rf_probs)));
```

In [ ]:
```
pipe.get_params().keys()
```

In [ ]:
```
# Grid Search
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import make_scorer
log_loss_build = make_scorer(log_loss, greater_is_better=False, needs_proba=True)

parameters = {'clf__max_depth': [16, 32],
              'clf__max_leaf_nodes': [24, 36],
              'clf__n_estimators': [250, 500]
             }

gs = GridSearchCV(pipe, parameters, cv=5, n_jobs=-1, scoring=log_loss_build)

# Fit and tune model
gs.fit(X_train, y_train);
```

In [ ]:
```
gs.best_params_
gs.best_score_
final_rf = gs.best_estimator_

final_rf.fit(X_train, y_train)
```

```
rf_pred = final_rf.predict(X_test)
rf_probs = final_rf.predict_proba(X_test);

print("Accuracy score: " + str(accuracy_score(y_test, rf_pred)))
print("Log loss: " + str(log_loss(y_test, rf_probs)))
```

Logistic Regression with Stochastic Gradient Descent

In [ ]:
```
#fitting linear classifiers and regressors under convex loss functions
from sklearn.linear_model import SGDClassifier

pipe = Pipeline([
    ('vect', CountVectorizer(max_features=10000)),
    ('tfidf', TfidfTransformer(use_idf=False)),
    ('clf', SGDClassifier(loss='log', penalty='l2'))
])

pipe.fit(X_train, y_train)
lr_pred = pipe.predict(X_test)
lr_probs = pipe.predict_proba(X_test);

print("Accuracy score: " + str(accuracy_score(y_test, rf_pred)))
print("Log loss: " + str(log_loss(y_test, rf_probs)));
```

In [ ]:
```
pipe.get_params().keys()
```

In [ ]:
```
# Grid search
alpha_range = 10.0**-np.arange(1,7)

parameters = {'clf__alpha': alpha_range,
              'clf__penalty': ['l1', 'l2'],
              'clf__max_iter': [10, 50]
             }

gs = GridSearchCV(pipe, parameters, cv=5, n_jobs=-1, scoring=log_loss_build)

# Fit and tune model
gs.fit(X_train, y_train);

gs.best_params_
lr_model = gs.best_estimator_
```

In [ ]:
```
gs.best_params_
gs.best_score_
lr_model = gs.best_estimator_

lr_model.fit(X_train, y_train)
lr_pred = lr_model.predict(X_test)
lr_probs = lr_model.predict_proba(X_test);

print("Accuracy score: " + str(accuracy_score(y_test, lr_pred)))
print("Log loss: " + str(log_loss(y_test, lr_probs)))
```

Ensemble Classifiers

In [ ]:
```
#to generate various base classifiers from which a new classifier is derived which performs better than any constituent clas
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
ada = AdaBoostClassifier(DecisionTreeClassifier(max_depth=2),
                         algorithm="SAMME",
                         n_estimators=600)

pipe = Pipeline([
    ('vect', CountVectorizer(max_features=10000)),
    ('tfidf', TfidfTransformer(use_idf=False)),
    ('clf', ada)
])

pipe.fit(X_train, y_train)
ada_pred = pipe.predict(X_test)
ada_probs = pipe.predict_proba(X_test);

print("Accuracy score: " + str(accuracy_score(y_test, ada_pred)))
print("Log loss: " + str(log_loss(y_test, ada_probs)));
```

In [ ]:
```
from sklearn.ensemble import VotingClassifier
eclf1 = VotingClassifier(estimators=[
        ('rf', final_rf), ('lr', lr_model), ('nb', final_model)], voting='soft')
```

```
pipe = Pipeline([
    ('vect', CountVectorizer(max_features=10000, ngram_range=(1,2))),
    ('tfidf', TfidfTransformer(use_idf=False))
])

pipe.fit_transform(X_train)
eclf1.fit(X_train, y_train)
vote_pred = eclf1.predict(X_test)
vote_probs = eclf1.predict_proba(X_test);

print("Accuracy score: " + str(accuracy_score(y_test, vote_pred)))
print("Log loss: " + str(log_loss(y_test, vote_probs)));
```

Final submission

In [ ]:
```
X_test = test_text
predictions = final_model.predict_proba(X_test)
```

In [ ]:
```
preds = pd.DataFrame(data=predictions, columns = final_model.named_steps['clf'].classes_)
```

In [ ]:
```
# generating a submission file
result = pd.concat([test[['id']], preds], axis=1)
result.set_index('id', inplace = True)
result.head()
```

Dataset

https://www.kaggle.com/competitions/spooky-author-identification/data