```python
# -*- coding: utf-8 -*-
"""Bernoulli NB Testing.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/199uJ4b4t-1xxUSn2onGApxLUPMPofQDt

<center><h1>Mini Project 2 - Bernoulli Naïve Bayes</h1>
<h4>This file tests the performance of the Bernoulli Naïve Bayes implemented by ourselves.</h4></center>

<h3>Team Members:</h3>
<center>
Yi Zhu, 260716006<br>
Fei Peng, 260712440<br>
Yukai Zhang, 260710915
</center>
"""

from google.colab import drive
drive.mount('/content/drive')

# make path = './' in-case you are running this locally
path = '/content/drive/My Drive/ECSE_551_F_2020/Mini_Project_02/'

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from time import time
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import Normalizer
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.feature_extraction import text
from sklearn import metrics
from sklearn.model_selection import GridSearchCV, cross_val_score, KFold
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import LabelEncoder

!pip install nltk
import nltk
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')

from nltk.stem import PorterStemmer
from nltk import word_tokenize
from nltk import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet

"""# Import Data"""

reddit_dataset = pd.read_csv(path+"train.csv")
reddit_test = pd.read_csv(path+"test.csv")

X = reddit_dataset['body']
y = reddit_dataset['subreddit']

"""# Define Vectorizer
### (To vectorize the text-based data to numerical features)

```

```python
1. CountVectorizer
1) Use "CountVectorizer" to transform text data to feature vectors.
2) Normalize your feature vectors
"""

def count_vectorizer(X_train, X_test):
    vectorizer = CountVectorizer(binary=True)
    vectors_train = vectorizer.fit_transform(X_train)
    vectors_test = vectorizer.transform(X_test)

    return vectors_train, vectors_test

"""2. CountVectorizer with stop word
1) Use "CountVectorizer" with stop word to transform text data to vector.
2) Normalize your feature vectors
"""

def count_vec_with_sw(X_train, X_test, max_features):
    stop_words = text.ENGLISH_STOP_WORDS
    if max_features:
        vectorizer = CountVectorizer(stop_words=stop_words, binary=True, max_features=5000)
    else:
        vectorizer = CountVectorizer(stop_words=stop_words, binary=True)
    vectors_train_stop = vectorizer.fit_transform(X_train)
    vectors_test_stop = vectorizer.transform(X_test)

    return vectors_train_stop, vectors_test_stop

"""3. TF-IDF
1) use "TfidfVectorizer" to weight features based on your train set.
2) Normalize your feature vectors
"""

def tfidf_vectorizer(X_train, X_test):
    tf_idf_vectorizer = TfidfVectorizer(binary=True)
    vectors_train_idf = tf_idf_vectorizer.fit_transform(X_train)
    vectors_test_idf = tf_idf_vectorizer.transform(X_test)

    return vectors_train_idf, vectors_test_idf

"""4. CountVectorizer with stem tokenizer
1) Use "StemTokenizer" to transform text data to vector.
2) Normalize your feature vectors
"""

class StemTokenizer:
     def __init__(self):
       self.wnl =PorterStemmer()
     def __call__(self, doc):
       return [self.wnl.stem(t) for t in word_tokenize(doc) if t.isalpha()]


def count_vec_stem(X_train, X_test):
    vectorizer = CountVectorizer(tokenizer=StemTokenizer(), binary=True)
    vectors_train_stem = vectorizer.fit_transform(X_train)
    vectors_test_stem = vectorizer.transform(X_test)

    return vectors_train_stem, vectors_test_stem

"""5. CountVectorizer with lemma tokenizer
1) Use "LemmaTokenizer" to transform text data to vector.
2) Normalize your feature vectors
```

```python
"""

def get_wordnet_pos(word):
    """Map POS tag to first character lemmatize() accepts"""
    tag = nltk.pos_tag([word])[0][1][0].upper()
    tag_dict = {"J": wordnet.ADJ,
                "N": wordnet.NOUN,
                "V": wordnet.VERB,
                "R": wordnet.ADV}
    return tag_dict.get(tag, wordnet.NOUN)


class LemmaTokenizer:
     def __init__(self):
       self.wnl = WordNetLemmatizer()
     def __call__(self, doc):
       return [self.wnl.lemmatize(t,pos =get_wordnet_pos(t)) for t in word_tokenize(doc) if t.isalpha()]


def count_vec_lemma(X_train, X_test):
    vectorizer = CountVectorizer(tokenizer=LemmaTokenizer(), binary=True)
    vectors_train_lemma = vectorizer.fit_transform(X_train)
    vectors_test_lemma = vectorizer.transform(X_test)

    return vectors_train_lemma, vectors_test_lemma

"""# Bernoulli Naïve Bayes Classifier"""

# Bernoulli Naïve Bayes
class BernoulliNB:
    '''
        This is the Bernoulli Naïve Bayes class, containing fit, perdict and accu_eval functions,
        as well as many other useful functions.
    '''

    def __init__(self, laplace):
        self.laplace = laplace  # true for performing Laplace smoothing
        self.le = LabelEncoder()  # encoder for classes

    def fit(self, X, y):
        '''
            This function takes the training data X and its corresponding labels vector y as input,
            and execute the model training.

            X - features of traning data
            y - class labels
        '''
        # Laplace smoothing paramerters
        num = 0
        den = 0
        if self.laplace:
            num += 1
            den += 2

        # encode the text-based class type to numerical values
        le = self.le
        le.fit(y)
        y_label = le.transform(y)
        n_k = len(le.classes_)  # number of classes
        n_j = X.shape[1]  # number of features
```

```python
        N = len(y)  # number of samples

        theta_k = np.zeros(n_k)  # probability of class k
        theta_j_k = np.zeros((n_k, n_j))  # probability of feature j given class k

        # compute theta values
        for k in range(n_k):
            count_k = (y_label==k).sum()
            theta_k[k] = count_k / N
            for j in range(n_j):
                theta_j_k[k][j] = (X[y_label==k, j].sum()+num) / (count_k+den)

        # store the theta values to this instance
        self.theta_k = theta_k
        self.theta_j_k = theta_j_k
        # print("Finished fitting...")

    def predict(self, X):
        '''
            This function takes a set of data as input and outputs predicted labels for the input points.
        '''
        le = self.le
        theta_k = self.theta_k
        theta_j_k = self.theta_j_k

        # this part works the same as the pseudo-code provided in Lecture 12
        # but matrix multiplication is much faster than nested loops
        i_m = np.zeros_like(X)  # identity matrix
        # predict classes
        y_pred = np.argmax(X.dot(np.log(theta_j_k).T)+(i_m-X).dot(np.log(1-theta_j_k).T)+theta_k, axis=1)

        # transform back to text-based values
        y_pred = le.inverse_transform(y_pred)
        return y_pred

"""### 1. K-fold validation using CountVectorizer"""

tic = time()
accuracies = []
clf = BernoulliNB(laplace=True)
kf = KFold(n_splits=5, shuffle=True)
for train_index, test_index in kf.split(X):
    vectors_train, vectors_test = count_vectorizer(X[train_index], X[test_index])
    clf.fit(vectors_train, y[train_index])
    a_s = metrics.accuracy_score(y[test_index], clf.predict(vectors_test))
    print(a_s)
    accuracies.append(a_s)

print("\t- Bernoulli Naïve Bayes + CountVectorizer -\nAccuracy: {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))

"""### 2. K-fold validation using CountVectorizer with stop word, max_features=5000"""

# with max_features=5000
tic = time()
accuracies = []
clf = BernoulliNB(laplace=True)
kf = KFold(n_splits=5, shuffle=True)
```

```python
for train_index, test_index in kf.split(X):
    vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[test_index], max_features=True)
    clf.fit(vectors_train, y[train_index])
    a_s = metrics.accuracy_score(y[test_index], clf.predict(vectors_test))
    print(a_s)
    accuracies.append(a_s)

print("\t- Bernoulli Naïve Bayes + CountVectorizer with stop word, Max Features = 5000 -\nAccuracy: {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))

"""### 3. K-fold validation using CountVectorizer with stop word"""

tic = time()
accuracies = []
clf = BernoulliNB(laplace=True)
kf = KFold(n_splits=5, shuffle=True)
for train_index, test_index in kf.split(X):
    vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[test_index])
    clf.fit(vectors_train, y[train_index])
    a_s = metrics.accuracy_score(y[test_index], clf.predict(vectors_test))
    print(a_s)
    accuracies.append(a_s)

print("\t- Bernoulli Naïve Bayes + CountVectorizer with stop word -\nAccuracy: {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))

"""### 4. K-fold validation using CountVectorizer with stem tokenizer"""

tic = time()
accuracies = []
clf = BernoulliNB(laplace=True)
kf = KFold(n_splits=5, shuffle=True)
for train_index, test_index in kf.split(X):
    vectors_train, vectors_test = count_vec_stem(X[train_index], X[test_index])
    clf.fit(vectors_train, y[train_index])
    a_s = metrics.accuracy_score(y[test_index], clf.predict(vectors_test))
    print(a_s)
    accuracies.append(a_s)

print("\t- Bernoulli Naïve Bayes + CountVectorizer with stem tokenizer -\nAccuracy: {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))

"""### 5. K-fold validation using CountVectorizer with lemma tokenizer"""

tic = time()
accuracies = []
clf = BernoulliNB(laplace=True)
kf = KFold(n_splits=5, shuffle=True)
for train_index, test_index in kf.split(X):
    vectors_train, vectors_test = count_vec_lemma(X[train_index], X[test_index])
    clf.fit(vectors_train, y[train_index])
    a_s = metrics.accuracy_score(y[test_index], clf.predict(vectors_test))
    print(a_s)
    accuracies.append(a_s)

print("\t- Bernoulli Naïve Bayes + CountVectorizer with lemma tokenizer -\nAccuracy: {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
```