ECSE 551 Fall 2020 Mini-project 2 Report

Fei Peng
260712440
fei.peng@mail.mcgill.ca

Yukai Zhang
260710915
yukai.zhang@mail.mcgill.ca

Yi Zhu
260716006
yi.zhu6@mail.mcgill.ca

Abstract

Bernoulli Naïve Bayes classification has become one of the most popular approaches in machine learning for document/text classification [1]. In this mini-project, we investigated and evaluated the performance of Bernoulli Naïve Bayes classifier by implementing it from scratch, and used it to analyze text from the website Reddit. Before feeding the dataset to the classifier, it was vectorized to change its text-based nature into numerical features. Different types of vectorization techniques were explored, and the best one was selected, which reached the highest accuracy in the k-fold validation, for reporting overall performance. Furthermore, experiments were conducted on the dataset using eight other additional classifiers from the SciKit learn package. Among them, the model that achieved the highest accuracy in k-fold validation was chosen for classifying the Kaggle test data. According to our testing result, tweaking the hyperparameters of the models does not have any significant impact on the result, however, choosing the correct type of vectorizer and classifier has the dominant effect on improving accuracy. Details will be presented in the following parts of this report.

1 Introduction

The boost in the popularity of participatory web and social networking sites has brought about many machine learning opportunities and challenges [2], for instance, content classification. As a significant portion of social media data is in natural language format, the main objective of this mini-project is to apply machine learning methods on classification of text data. The dataset was obtained from Reddit, a popular website where users post and comment on content in different themed communities (i.e., subreddit), with eight classification categories: rpg, anime, datascience, hardware, cars, gamernews, gamedev, and computers.

Before feeding any data into the classifiers, it has to be vectorized to change its text-based nature into numerical values. We implemented five different vectorizers, namely count vectorizer (CV), count vectorizer with stopwords (CVSW), TF-IDF vectorizer (TF-IDFV), count vectorizer with stemming (CVS), and count vectorizer with lemmatization (CVL). Except for Bernoulli Naïve Bayes classifier which requires binary input (TF-IDF vectorizer is therefore not feasible), all other classifiers were fed with z-score normalized input to improve accuracy.

In the first part of this project, we implemented a Bernoulli Naïve Bayes classifier from scratch and investigated its performance using k-fold cross validation. Bernoulli Naïve Bayes algorithm is a less costly generative model which proposes Naïve Bayes assumption (assume features of sample x are conditionally independent given class y) for simplification. Compared to discriminative learning (e.g., Logistic Regression) which directly estimate $P(y|x)$, generative learning models $P(x|y)$ and $P(y)$ separately and use Bayes' rule to estimate $P(y|x)$ which allows more flexibility. Moreover, to avoid the occurrence of zero probability, we also implemented Laplace smoothing in the fit method.

The second part is to perform experiments using other classifiers from SciKit learn package, and measure their performance in terms of speed and accuracy. The classifiers we investigated are: Logistic Regression (LR), Multinomial Naïve Bayes (MNB), Support Vector Machine (SVM), Random Forest (RF), Decision Tree (DT), AdaBoost (AB), K-Neighbors (KN), and Multi-layer Perceptron classifier (MLP) [3, 4]. In addition, we explored the behaviour of Hard/Soft Voting Classifiers and the Stacking Classifier (SC) [3, 4] to further improve the accuracy. According to our experiments, while tweaking the hyperparameters of the classifiers could slightly increase the accuracy (at the cost of dramatically increasing training time), choosing the proper vectorizer and classifier helped improve the performance significantly.

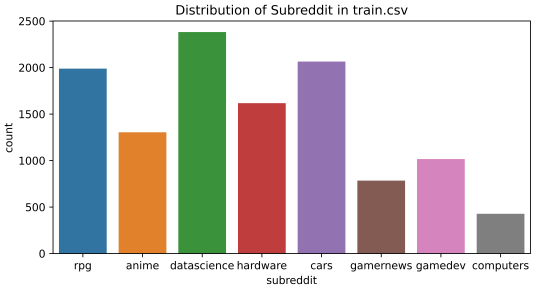

Figure 1: Class/Subreddit Distribution of Train Data

Table 1: Number of Features and Processing Time of Five Different Types of Vectorization Methods

	CV	CVSW	TF-IDFV	CVS	CVL
Number of Features	41033	40729	41033	20765	24777
Time (s)	0.96842	0.91925	1.0158	29.1	144.17

2 Datasets

Before performing any investigation on machine learning models, it is important that the characteristics of the dataset is properly analyzed and understood. Aside from that, data preprocessing lays the groundwork for the following data analysis, as its product is the final training set [5]. The datasets we used are *train.csv* which contains 11582 samples and their corresponding class/subreddit, and *test.csv* that includes 2898 test samples with their ID and features from which we predicted their class/subreddit (and submitted to Kaggle competition). In this project, the following data preprocessing had been performed.

2.1 Entropy Analysis

The classification of the Reddit dataset is an example of multi-class classification. The distribution of each class/subreddit is shown in Figure 1. The uncertainty in prediction can be quantified by the entropy of each dataset with equation 1, and the calculated data entropy is 2.847 which stands for a relatively even distribution and could be preferable for some classification techniques.

$$H(X) = -\sum_{x \epsilon X} p(x) log_2 p(x) \tag{1}$$

2.2 Vectorization

The features of the raw Reddit dataset is in text format, thus it is necessary that they are vectorized and transformed into numerical values before feeding the data into any classifier [4]. The pure count vectorizer simply counts the occurrence of words, while count vectorizer with stopwords eliminates the typical English stopwords before counting. TF-IDF vectorizer calculates the term frequency times inverse document frequency. Stemming and lemmatization recover the canonical form of the words. The numbers of features after vectorization are presented in Table 1, as well as the processing time for performing each vectorization method. It is indicated that stemming and lemmatization decrease feature number by nearly half which could help increase the training speeding for training classifiers in the future, however, the processing time of themselves is approximately ten to a hundred times that of without stemming or lemmatization. They are even less preferable as they failed to improve classification accuracy either.
On the other hand, we also considered obtaining binary vector representation for the text input (by setting $binary = True$ for vectorization). In this case, the result would rely on the presence or absence of each feature, instead of the number of its occurrences. This argument must be set to $True$ for Bernoulli Naïve Bayes classifier which requires the features to be binary-valued [3]. It was also tested for other classifiers to see whether it would help increase accuracy, and the result will be discussed in the following parts of this report.

2.3 Normalization

Normalizing the data into z-score can centralize the value distribution of the data while keeping its precision. However, for text-based datasets like Reddit, this preprocessing approach might not provide a satisfying result, as the meaning of the numerical representation of features is the number of occurrences of each word in each comment. This value is usually either 0 or 1.

2.4 Maximum Features

The $max\_features$ argument of the vectorizer functions is provided by sklearn to build a vocabulary that only considers the top $max\_features$ ordered by term frequency across the corpus [3, 4]. For the preprocessing input data of Bernoulli Naïve Bayes classifier, we set $max\_features = 5000$ as our implementation may not be optimized for speed. By setting the maximum features, the time consumed by fit and predict functions was significantly shortened, as the number of features was one eighth of the original dataset.

3 Proposed Approach

In this section, we are going to discuss the features selection and methods implementation in terms of accuracy, robustness/consistency, and speed. However, due to time constraint of this project and page limitation of this report, we are only going to discuss the methods with outstanding performances.

3.1 Training/Validation Splits

As mentioned in the previous section, the training datasets contains 11582 samples and testing datasets contains 2898 samples. The ratio of training to testing samples is 3.99655 (4:1). Therefore we chose to perform 5-fold validation (training:testing = 4:1) on all the chosen classifiers to compare and select the optimum model for competition. In this way, we could ensure consistency between the validation and testing process, and increase the possibility to reach higher accuracy on testing data.

3.2 Laplace Smoothing

Originally we would use equation 2 to calculate $\theta_{j,k}$ matrix (the probability that feature j occurs given class k). However, it could cause zero division when a certain class does not occur in the dataset ($P(y = k) = 0$). By applying Laplace smoothing, we could prevent zero division by adding the numerator by 1 and denominator by 2.

$$\theta_{j,k} = P(x_j = 1|y = k) = \frac{P((x_j = 1 \cap y = k)}{P(y = k)} \tag{2}$$

3.3 Algorithm Selection

At the beginning of this project, we started with building a Bernoulli Naïve Bayes classifier (BNB) from scratch, which is one of the linear classification methods. Like all generative learning approaches, Bernoulli Naïve Bayes separately model $P(x|y)$ and $P(y)$ and use Bayes' rule to estimate $P(y|x)$ where x represents features and y is the corresponding classes. Moreover, compared to linear discriminant analysis (LDA) which is another generative learning method, it proposes Naïve Bayes assumption to reduce calculation cost. Therefore, assume there are overall k classes and j features, it could directly calculate the probability of each class (i.e., k) and the probability of each feature given each class (i.e., $\theta_{j,k}$). Laplace smoothing could be applied when calculating $\theta_{j,k}$ to prevent zero division. Compared to Multinomial Naïve Bayes where the input data are typically represented as word vector counts [3], Bernoulli Naïve Bayes only allows binary-valued input, therefore count vectorizer (with $binary = True$) without normalization should be used to preprocess the input data. Finally, it would use these probabilities to make predictions for the testing data.
Aside from Bernoulli Naïve Bayes, all other classifiers accept non-binary features, thus we have the flexibility to choose the most suitable vectorizer-classifier pair. Next, we continued to investigate the behaviour of linear models by performing experiments on Logistic Regression classifier and Linear Support Vector Machine (SVM) provided by sklearn [4]. In contrast to generative learning, Logistic Regression, a discriminative learning method, directly estimate $P(y|x)$ by first obtaining the weight (w) of each feature with gradient descent and then apply a logistic/sigmoid function to predict classes. Linear SVM predicts classes by separating samples with hyperplanes (i.e., linear classifiers) and optimizing them by maximizing the margin of each classifier. These linear models, especially SVM, have promising performance when the number of samples is relatively small when deep learning is not feasible, and their accuracy could be increased by applying regularization, which will be discussed in the next subsection.
Furthermore, we focused on improving the accuracy by exploring more complex methods. The idea is to combine the simple learning algorithms into a Neural Network or Ensemble methods [6]. Multi-layer Perceptron (MLP) is a Neural Network classifier provided by sklearn [4], which uses a network (with hidden layers) of stacked perceptron-like elements to represent non-linearly separate functions, where perceptrons are simple linear classifiers. Stacking Classifier (SC) is one of the stacking Ensemble methods [4], which consists in stacking the output of different base learners and using a final classifier/meta model to compute the final prediction (usually Logistic Regression). The features of the final classifier would be the combination of output from base learners. This method combines the strength of each individual estimator to produce the optimum output.
To select the best model, we mainly focused on choosing the one that provides highest accuracy, but also made certain trade-offs between accuracy and speed. We found that by tweaking regularization terms and hyperparameters by a certain amount, we could bring some improvements to accuracy. In addition, using the best models as base learners, we could train a Stacking Classifier to achieve better accuracy. We also considered other classifiers such as K-Neighbors, Decision Trees, and other Ensemble classifiers like Random Forest (Bagging), AdaBoost (Boosting), and Hard/Soft Voting Classifier (Stacking), but they either were not as satisfying enough or fails to increase accuracy by sacrificing speed, thus we would not go deeper into them.

3.4 Regularization and Hyperparameters

In linear models such as Logistic Regression and Linear SVM, there is a regularization parameter (i.e., C) when defining the classifier. It is the inverse of regularization strength where smaller values specify stronger regularization. For the MLP classifier, the regularization is controlled by parameter alpha which is the L2 penalty [3]. The detail effect and result of regularization will be discussed in the next section. As for hyperparameter, we increased the value of max_iter (i.e., maximum iteration) from default (set by sklearn and varies for different classifiers) to 1000, to make sure the algorithm converges. Moreover, for the MLP classifier, the hidden layer size and learning rate also influences the accuracy and speed which should be considered.

Table 2: Accuracy and Processing Time for Five Classifiers

	BNB	LR	Linear SVM	MLP	SC
Maximum Accuracy	0.81316	0.89613	0.89026	0.90822	0.91029
Processing Time (s)	168.842	116.797	321.477	1379.423	19795.276

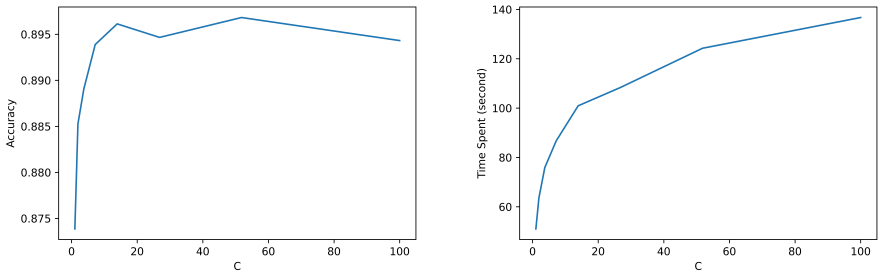

Figure 2: Mean Validation Accuracy (left) and Processing Time (right) with respect to regularization term of Logistic Regression

4 Results

The results we obtained in this project are summarized in Table 2, where the processing time is for 5-fold validation and represented in seconds. Due to page limitation, please refer to Appendix A for more details. The result used for final submission on Kaggle was obtained using Stacking Classifier since it has the highest cross validation accuracy, and the test set leaderboard accuracy we achieved is 0.93095.
All combinations of vectorizer type and classification methods were evaluated, and details are illustrated in tables of Appendix A. The optimum vectorizer for preprocessing the input of concerned sklearn classifiers is TF-IDF vectorizer, one of the explanation is that TF-IDF vectorizer determine if a term is important/indicative of a document by not only the number of occurance in the document but also whether it is a relative rare word overall.

4.1 Bernoulli Naïve Baye

This classifier was implemented from scratch. The accuracy and speed are shown in Table 2. The maximum accuracy was achieved when the $max\_features$ is 5000, binary is set to $True$, and $stop\_words$ equals to $ENGLISH\_STOP\_WORDS$ for count vectorizer during data preprocessing. Laplace smoothing is also applied to prevent zero division. To have a more thorough understanding of its performance, we compared the accuracy and speed with that provided by sklearn. The maximum achieved accuracy is 0.83448, with processing time equals 4.717 seconds, while the exact same vectorizer was used. Note that the sklearn model was able to cooperate with the TF-IDF vectorizer, but this is not the case for our model. The sklearn model does not require binary input because there is a $binarize$ parameter which handles non-binary input. Overall, the accuracy of our model is no worse than that from sklearn, however, ours requires around 40 times processing time.

4.2 Logistic Regression:

As shown in Figure2, the accuracy could be increased by applying less regularization in Logistic Regression. It was explained earlier that C is the inverse of regularization strength, and the default value is 1.0. The accuracy reaches its maximum when C is around 40.0, and the processing time remains within a reasonable range. The accuracy and speed are shown in Table 2.

4.3 Linear SVM

As indicated in Figure 3, the accuracy could as well be increased by applying less regularization in Linear SVM. The accuracy reaches its maximum when C is around 1.0, and this is also when the processing time is the lowest among all models. The accuracy and speed are shown in Table 2.

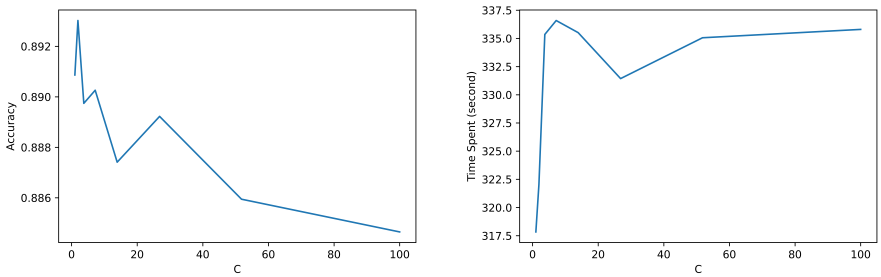

Figure 3: Mean Validation Accuracy (left) and Processing Time (right) with respect to regularization term of Linear SVM

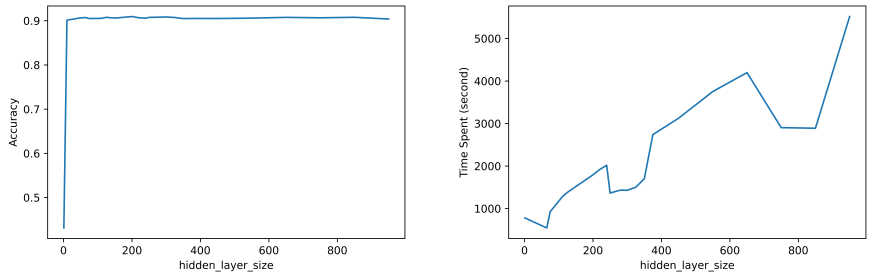

Figure 4: Mean Validation Accuracy (left) and Processing Time (right) with respect to hidden layer size of MLP Classifier

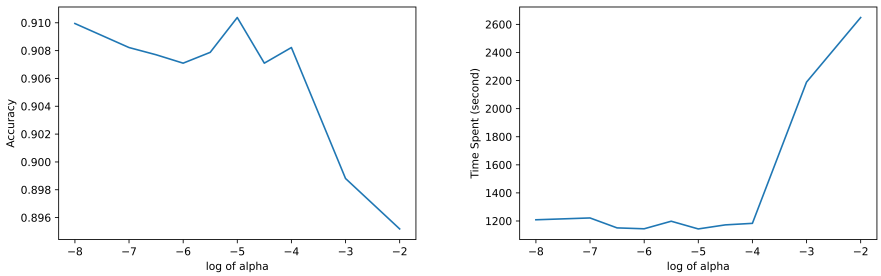

Figure 5: Mean Validation Accuracy (left) and Processing Time (right) with respect to regularization term of MLP Classifier

4.4 MLP Classifier

For this classifier, we set the $learning\_rate\_init = 0.0001$ and $learning\_rate =$'$adaptive$' (keeps the learning rate constant to '$learning\_rate\_init$' as long as training loss keeps decreasing), which helps increase the accuracy slightly over that of default setting. Moreover, we investigated the effect of changing hidden layer size, and the result is illustrated in Figure 4. As shown in the plots, the optimum choice would be 100 hidden layers in terms of both accuracy and speed. In addition, the effect of regularization was examined, and the result is indicated in Figure 5, the classifier reaches maximum accuracy when $alpha$ equals 10^{-5}. The accuracy and processing time are shown in Table 2.

4.5 Stacking Classifier

Compared to the Voting Classifier which uses voting as its final classifier, Stacking Classifier feeds the output of base learners into another machine learning method as its final classifier. In our implementation, the three best models from Logistic Regression, Linear SVM, and MLP Classifier are selected as base learners, which have already shown promising performance, and the final classifier is another Logistic Regression model. The accuracy and speed are shown in Table 2. The Stacking method improves accuracy by combining the advantage of each base learner, in return, it sacrifices more than 90% of the training speed.

5 Discussion and Conclusion

In conclusion, the Bernoulli Naïve Bayes classifier implemented in this project performed as expected on the Reddit dataset. It achieved an accuracy of 81.3%, with processing time equals 168.842 seconds. However, compared to the Bernoulli Naïve Bayes classifier in sklearn, our model is significantly slower, and not able to accept non-binary input, therefore cannot coorperate with TF-IDF vectorizer or any other non-binary vectorizer. The future investigation could be improving the speed of our algorithm in terms of data structure by changing the remaining nested loops with numpy matrix multiplication. Another direction would be dealing with non-binary data by binarizing the input before the fit function.

On the other hand, we achieved a leaderboard accuracy of 0.93095 using the Stacking Classifier in sklearn, with Logistic Regression, Linear SVM, and MLP Classifier as three base learners and another Logistic Regression as final classifier. We found that linear models as well as their combinations have relatively better performance than other (non-linear) machine learning models we investigated. In the future, we may take a closer look at the cause of this phenomenon, and find out a better model to improve accuracy.

6 Statement of Contributions

Fei Peng implemented the Bernoulli Naïve Bayes classifier from scratch, as well as various types of vectorizers for data preprocessing. Yukai Zhang was in charge of researching feasible classifier implementations in sklearn, measuring validation accuracies of different classifiers using k-fold validation, and selecting the best model for kaggle test. Yi Zhu was responsible for examining the classifiers in terms of the training speed, and the effect of choosing different hyperparameters on speed and accuracy.

References

[1] G. Singh, B. Kumar, L. Gaur, and A. Tyagi, "Comparison between Multinomial and Bernoulli Naïve Bayes for Text Classification," in *2019 International Conference on Automation, Computational and Technology Management (ICACTM),* 24-26 April 2019 2019, pp. 593-596, doi: 10.1109/ICACTM.2019.8776800.

[2] L. Tang and H. Liu, "Leveraging social media networks for classification," *Data Mining and Knowledge Discovery,* vol. 23, no. 3, pp. 447-478, 2011/11/01 2011, doi: 10.1007/s10618-010-0210-x.

[3] F. Pedregosa et al.,"Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research,* vol. 12, pp. 2825–2830, 2011.

[4] L. Buitinck et al., "API design for machine learning software: Experiences from the scikit-learn project," *API Design for Machine Learning Software: Experiences from the Scikit-learn Project,* 09/01 2013.

[5] S. Kotsiantis, D. Kanellopoulos, and P. Pintelas, "Data preprocessing for supervised learning," *International Journal of Computer Science,* vol. 1, no. 2, pp. 111-117, 2006.

[6] T. G. Dietterich, "Ensemble Methods in Machine Learning," Berlin, Heidelberg, 2000: Springer Berlin Heidelberg, in *Multiple Classifier Systems,* pp. 1-15.

Appendix

(Please see following attached pages for Appendix A: Results and Appendix B: Code Implementations)

Appendix A: Results

Table A.1: Performance of Bernoulli Naïve Bayes Implemented by Ourselves.

Vectorizer	Normalization	Max Features	Time Spent (second)	Accuracy (%)
CountVectorizer	Yes	Unlimited	1804.497	74.288
CountVectorizer with Stopwords	Yes	5000	168.842	81.316
CountVectorizer with Stopwords	Yes	Unlimited	1285.886	78.890
CountVectorizer with Stemming	Yes	Unlimited	1049.033	72.224
CountVectorizer with Lemmatization	Yes	Unlimited	1750.286	72.906

Table A.2: Performance of Logistic Regression.

Vectorizer	Normalization	Max Features	Time Spent (second)	Accuracy (%)
CountVectorizer	Yes	Unlimited	64.152	82.214
CountVectorizer	No	Unlimited	117.682	86.695
CountVectorizer with Stopwords	Yes	Unlimited	47.928	86.609
CountVectorizer with Stopwords	No	Unlimited	37.534	87.455
TF-IDF Vectorizer	Yes	Unlimited	116.797	89.613
TF-IDF Vectorizer	No	Unlimited	116.167	89.233
CountVectorizer with Stemming	Yes	Unlimited	189.210	81.748
CountVectorizer with Stemming	No	Unlimited	236.312	85.970
CountVectorizer with Lemmatization	Yes	Unlimited	743.057	81.471
CountVectorizer with Lemmatization	No	Unlimited	809.459	86.039

Table A.3: Accuracy of Multinomial Naïve Bayes.

Vectorizer	Normalization	Max Features	Accuracy (%)
CountVectorizer	Yes	Unlimited	58.012
CountVectorizer with Stopwords	Yes	Unlimited	73.441
TF-IDF Vectorizer	Yes	Unlimited	70.100
CountVectorizer with Stemming	Yes	Unlimited	58.962
CountVectorizer with Lemmatization	Yes	Unlimited	57.503

Table A.4: Accuracy of Support Vector Machine.

Vectorizer	Normalization	Max Features	Accuracy (%)
CountVectorizer	Yes	Unlimited	84.234
CountVectorizer with Stopwords	Yes	Unlimited	87.351
TF-IDF Vectorizer	Yes	Unlimited	88.715
CountVectorizer with Stemming	Yes	Unlimited	83.543
CountVectorizer with Lemmatization	Yes	Unlimited	83.284

Table A.5: Accuracy of Random Forest.

Vectorizer	Normalization	Max Features	Accuracy (%)
CountVectorizer	Yes	Unlimited	30.358
CountVectorizer with Stopwords	Yes	Unlimited	32.283
TF-IDF Vectorizer	Yes	Unlimited	31.842
CountVectorizer with Stemming	Yes	Unlimited	38.111
CountVectorizer with Lemmatization	Yes	Unlimited	34.528

Table A.6: Accuracy of Decision Tree.

Vectorizer	Normalization	Max Features	Accuracy (%)
CountVectorizer	Yes	Unlimited	67.087
CountVectorizer with Stopwords	Yes	Unlimited	70.471
TF-IDF Vectorizer	Yes	Unlimited	66.906
CountVectorizer with Stemming	Yes	Unlimited	66.517
CountVectorizer with Lemmatization	Yes	Unlimited	65.153

Table A.7: Accuracy of AdaBoost.

Vectorizer	Normalization	Max Features	Accuracy (%)
CountVectorizer	Yes	Unlimited	68.434
CountVectorizer with Stopwords	Yes	Unlimited	67.968
TF-IDF Vectorizer	Yes	Unlimited	68.546
CountVectorizer with Stemming	Yes	Unlimited	70.868
CountVectorizer with Lemmatization	Yes	Unlimited	69.807

Table A.8: Accuracy of K-Neighbors.

Vectorizer	Normalization	Max Features	Accuracy (%)
CountVectorizer	Yes	Unlimited	46.194
CountVectorizer with Stopwords	Yes	Unlimited	65.447
TF-IDF Vectorizer	Yes	Unlimited	79.019
CountVectorizer with Stemming	Yes	Unlimited	48.860
CountVectorizer with Lemmatization	Yes	Unlimited	47.643

Table A.9: Accuracy of Neural Network.

Vectorizer	Normalization	Max Features	Accuracy (%)
CountVectorizer	Yes	Unlimited	89.803
CountVectorizer with Stopwords	Yes	Unlimited	89.898
TF-IDF Vectorizer	Yes	Unlimited	90.494
CountVectorizer with Stemming	Yes	Unlimited	88.361
CountVectorizer with Lemmatization	Yes	Unlimited	88.517

Table A.10: Performance of Bernoulli Naïve Bayes Implemented by Sklearn.

Vectorizer	Normalization	Max Features	Time Spent (second)	Accuracy (%)
CountVectorizer	Yes	Unlimited	5.479	73.934
CountVectorizer with Stopwords	Yes	Unlimited	4.885	74.633
CountVectorizer with Stopwords	Yes	5000	4.717	83.448
TF-IDF Vectorizer	Yes	Unlimited	5.328	73.891
CountVectorizer with Stemming	Yes	Unlimited	139.701	78.216
CountVectorizer with Lemmatization	Yes	Unlimited	689.932	76.826

Table A.11: Time Required for Classifiers with TF-IDF Vectorizer

Classifier	Time Spent (second)
Multinomial NB	5.411
Support Vector Machine	321.477
Random Forest	7.130
Decision Tree	33.605
Ada Boost	70.709
K-Neighbors	12.815
Neural Network	1379.423

Table A.12: Performance of Ensemble Classifier.

Classifier	Base Learner 1	Base Learner 2	Base Learner 3	Time Spent (s)	Accuracy (%)
Voting Classifier	Neural Classifier	Logistic Reg.	SVM	3707.132	90.511
Stacking Classifier	Neural Classifier	SVM	Logistic Reg.	19795.276	91.029

Appendix B: Code Implementations

(Please see following attached pages)

```
# -*- coding: utf-8 -*-
"""Data Preprocessing.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1bRmLzkFmGP7xgUOUI6iv_p6dW1hX6R6P

<center><h1>Mini Project 2 - Bernoulli Naïve Bayes</h1>
<h3>Data Preprocessing</h3>
<h4>This file performs some of the operations on Data Preprocessing and Analysis.</h4></center>

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

# Importations
"""

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
import seaborn as sns
import random
from scipy import stats
from google.colab import files
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

!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

from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier

"""# Data Preprocessing"""

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

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

class Data_Processing:
    def __init__(self, data, name='New Data'):
        self.data = data
        self.name = name

    def show_y_dist(self, ydata):
        plt.figure(figsize=(8,4))
        plt.subplot(111), sns.countplot(x='subreddit', data=ydata)
        plt.title('Distribution of Subreddit in {}'.format(self.name))
        plt.savefig("Distribution of Subreddit in {}.png".format(self.name), dpi = 1200)
        files.download("Distribution of Subreddit in {}.png".format(self.name))
        plt.show()

data_analysis = Data_Processing(reddit_dataset.values, 'train.csv')
data_analysis.show_y_dist(reddit_dataset)

# calculate the data entropy
from sklearn.preprocessing import LabelEncoder
le = LabelEncoder()  # encoder for classes
le.fit(y)
y_label = le.transform(y)
n_k = len(le.classes_)
N = len(y)
theta_k = np.zeros(n_k)  # probability of class k
# compute theta values
for k in range(n_k):
    count_k = (y_label==k).sum()
    theta_k[k] = count_k / N

from scipy.stats import entropy
print("Data entropy is", entropy(theta_k, base=2))

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

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

def count_vectorizer(X_train):
    vectorizer = CountVectorizer()
    vectors_train = vectorizer.fit_transform(X_train)
    return vectors_train

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

```
124
125 def count_vec_with_sw(X_train):
126     stop_words = text.ENGLISH_STOP_WORDS
127     vectorizer = CountVectorizer(stop_words=stop_words)
128     vectors_train_stop = vectorizer.fit_transform(X_train)
129     return vectors_train_stop
130
131 """3. TF-IDF
132 1) use "TfidfVectorizer" to weight features based on your train set.
133 2) Normalize your feature vectors
134 """
135
136 def tfidf_vectorizer(X_train):
137     tf_idf_vectorizer = TfidfVectorizer()
138     vectors_train_idf = tf_idf_vectorizer.fit_transform(X_train)
139     return vectors_train_idf
140
141 """4. CountVectorizer with stem tokenizer
142 1) Use "StemTokenizer" to transform text data to vector.
143 2) Normalize your feature vectors
144 """
145
146 class StemTokenizer:
147      def __init__(self):
148        self.wnl =PorterStemmer()
149      def __call__(self, doc):
150        return [self.wnl.stem(t) for t in word_tokenize(doc) if t.isalpha()]
151
152
153 def count_vec_stem(X_train):
154     vectorizer = CountVectorizer(tokenizer=StemTokenizer())
155     vectors_train_stem = vectorizer.fit_transform(X_train)
156     return vectors_train_stem
157
158 """5. CountVectorizer with lemma tokenizer
159 1) Use "LemmaTokenizer" to transform text data to vector.
160 2) Normalize your feature vectors
161 """
162
163 def get_wordnet_pos(word):
164     """Map POS tag to first character lemmatize() accepts"""
165     tag = nltk.pos_tag([word])[0][1][0].upper()
166     tag_dict = {"J": wordnet.ADJ,
167                 "N": wordnet.NOUN,
168                 "V": wordnet.VERB,
169                 "R": wordnet.ADV}
170     return tag_dict.get(tag, wordnet.NOUN)
171
172
173 class LemmaTokenizer:
174      def __init__(self):
175        self.wnl = WordNetLemmatizer()
176      def __call__(self, doc):
177        return [self.wnl.lemmatize(t,pos =get_wordnet_pos(t)) for t in
    word_tokenize(doc) if t.isalpha()]
178
179
180 def count_vec_lemma(X_train):
181     vectorizer = CountVectorizer(tokenizer=LemmaTokenizer())
182     vectors_train_lemma = vectorizer.fit_transform(X_train)
183     return vectors_train_lemma
184
185 """# Measure the time required for each vectorizer to perform vectorization
```

```
186
187 ### 1. CountVectorizer
188 """
189
190 tic = time()
191 X_vec = count_vectorizer(X)
192 print("\t\t- Count Vectorizer - \nfeature number: ", X_vec.shape[1], "\t\tTime
     spent: ", time()-tic, "s.")
193
194 """### 2. CountVectorizer with stop word"""
195
196 tic = time()
197 X_vec = count_vec_with_sw(X)
198 print("\t\t- Count Vectorizer with stop word - \nfeature number: ", X_vec.
    shape[1], "\t\tTime spent: ", time()-tic, "s.")
199
200 """### 3. TF-IDF"""
201
202 tic = time()
203 X_vec = tfidf_vectorizer(X)
204 print("\t\t- TF-IDF Vectorizer - \nfeature number: ", X_vec.shape[1], "\t\t
    Time spent: ", time()-tic, "s.")
205
206 """### 4. CountVectorizer with stem tokenizer"""
207
208 tic = time()
209 X_vec = count_vec_stem(X)
210 print("\t\t- CountVectorizer with stem tokenizer - \nfeature number: ", X_vec.
    shape[1], "\t\tTime spent: ", time()-tic, "s.")
211
212 """### 5. CountVectorizer with lemma tokenizer"""
213
214 tic = time()
215 X_vec = count_vec_lemma(X)
216 print("\t\t- CountVectorizer with lemma tokenizer - \nfeature number: ", X_vec
    .shape[1], "\t\tTime spent: ", time()-tic, "s.")
```

```
# -*- coding: utf-8 -*-
"""Additional Classifiers.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1jQsRTZihNlPC99pBzRYqvTbowAY-tqi1

<center><h1>Mini Project 2 - Bernoulli Naïve Bayes</h1>
<h4>This file records the accuracies of the combinations of 8 classifiers and 5 vectorizers.</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 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

!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

"""Additional classifiers:
1. Logistic Regression
2. Multinomial Naïve Bayes
3. Support Vector Machine
4. Random Forest
5. Decision Tree
6. Ada Boost
7. k-Neighbors
8. Neural Network
"""

from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
```

```
 63 from sklearn import svm
 64 from sklearn.ensemble import RandomForestClassifier
 65 from sklearn.tree import DecisionTreeClassifier
 66 from sklearn.ensemble import AdaBoostClassifier
 67 from sklearn.neighbors import KNeighborsClassifier
 68 from sklearn.neural_network import MLPClassifier
 69
 70 reddit_dataset = pd.read_csv(path+"train.csv")
 71 reddit_test = pd.read_csv(path+"test.csv")
 72
 73 X = reddit_dataset['body']
 74 y = reddit_dataset['subreddit']
 75
 76 """# Define Vectorizer
 77 ### (To vectorize the text-based data to numerical features)
 78
 79 1. CountVectorizer
 80 1) Use "CountVectorizer" to transform text data to feature vectors.
 81 2) Normalize your feature vectors
 82 """
 83
 84 def count_vectorizer(X_train, X_test):
 85     vectorizer = CountVectorizer()
 86     vectors_train = vectorizer.fit_transform(X_train)
 87     vectors_test = vectorizer.transform(X_test)
 88
 89     normalizer_train = Normalizer().fit(X=vectors_train)
 90     vectors_train = normalizer_train.transform(vectors_train)
 91     vectors_test = normalizer_train.transform(vectors_test)
 92
 93     return vectors_train, vectors_test
 94
 95 """2. CountVectorizer with stop word
 96 1) Use "CountVectorizer" with stop word to transform text data to vector.
 97 2) Normalize your feature vectors
 98 """
 99
100 def count_vec_with_sw(X_train, X_test):
101     stop_words = text.ENGLISH_STOP_WORDS
102     vectorizer = CountVectorizer(stop_words=stop_words)
103     vectors_train_stop = vectorizer.fit_transform(X_train)
104     vectors_test_stop = vectorizer.transform(X_test)
105
106     normalizer_train = Normalizer().fit(X=vectors_train_stop)
107     vectors_train_stop= normalizer_train.transform(vectors_train_stop)
108     vectors_test_stop = normalizer_train.transform(vectors_test_stop)
109
110     return vectors_train_stop, vectors_test_stop
111
112 """3. TF-IDF
113 1) use "TfidfVectorizer" to weight features based on your train set.
114 2) Normalize your feature vectors
115 """
116
117 def tfidf_vectorizer(X_train, X_test):
118     tf_idf_vectorizer = TfidfVectorizer()
119     vectors_train_idf = tf_idf_vectorizer.fit_transform(X_train)
120     vectors_test_idf = tf_idf_vectorizer.transform(X_test)
121
122     normalizer_train = Normalizer().fit(X=vectors_train_idf)
123     vectors_train_idf= normalizer_train.transform(vectors_train_idf)
124     vectors_test_idf = normalizer_train.transform(vectors_test_idf)
125
```

```
126     return vectors_train_idf, vectors_test_idf
127
128 """4. CountVectorizer with stem tokenizer
129 1) Use "StemTokenizer" to transform text data to vector.
130 2) Normalize your feature vectors
131 """
132
133 class StemTokenizer:
134      def __init__(self):
135        self.wnl =PorterStemmer()
136      def __call__(self, doc):
137        return [self.wnl.stem(t) for t in word_tokenize(doc) if t.isalpha()]
138
139
140 def count_vec_stem(X_train, X_test):
141     vectorizer = CountVectorizer(tokenizer=StemTokenizer())
142     vectors_train_stem = vectorizer.fit_transform(X_train)
143     vectors_test_stem = vectorizer.transform(X_test)
144
145     normalizer_train = Normalizer().fit(X=vectors_train_stem)
146     vectors_train_stem= normalizer_train.transform(vectors_train_stem)
147     vectors_test_stem = normalizer_train.transform(vectors_test_stem)
148
149     return vectors_train_stem, vectors_test_stem
150
151 """5. CountVectorizer with lemma tokenizer
152 1) Use "LemmaTokenizer" to transform text data to vector.
153 2) Normalize your feature vectors
154 """
155
156 def get_wordnet_pos(word):
157     """Map POS tag to first character lemmatize() accepts"""
158     tag = nltk.pos_tag([word])[0][1][0].upper()
159     tag_dict = {"J": wordnet.ADJ,
160                 "N": wordnet.NOUN,
161                 "V": wordnet.VERB,
162                 "R": wordnet.ADV}
163     return tag_dict.get(tag, wordnet.NOUN)
164
165
166 class LemmaTokenizer:
167      def __init__(self):
168        self.wnl = WordNetLemmatizer()
169      def __call__(self, doc):
170        return [self.wnl.lemmatize(t,pos =get_wordnet_pos(t)) for t in
    word_tokenize(doc) if t.isalpha()]
171
172
173 def count_vec_lemma(X_train, X_test):
174     vectorizer = CountVectorizer(tokenizer=LemmaTokenizer())
175     vectors_train_lemma = vectorizer.fit_transform(X_train)
176     vectors_test_lemma = vectorizer.transform(X_test)
177
178     normalizer_train = Normalizer().fit(X=vectors_train_lemma)
179     vectors_train_lemma= normalizer_train.transform(vectors_train_lemma)
180     vectors_test_lemma = normalizer_train.transform(vectors_test_lemma)
181
182     return vectors_train_lemma, vectors_test_lemma
183
184 """# Measure Accuracies of different classifiers using K-fold Validation
185
186 ## 1. Logistic Regression
187
```

```
188 ### 1. CountVectorizer
189 """
190 
191 accuracies = []
192 clf = LogisticRegression(C=1.0, max_iter=1000, random_state=0)
193 kf = KFold(n_splits=5, shuffle=True)
194 for train_index, test_index in kf.split(X):
195     vectors_train, vectors_test = count_vectorizer(X[train_index], X[
    test_index])
196     clf.fit(vectors_train, y[train_index])
197     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
198 
199 print(np.mean(accuracies))
200 
201 """### 2. CountVectorizer with stop word"""
202 
203 accuracies = []
204 clf = LogisticRegression(C=1.0, max_iter=1000, random_state=0)
205 kf = KFold(n_splits=5, shuffle=True)
206 for train_index, test_index in kf.split(X):
207     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
    test_index])
208     clf.fit(vectors_train, y[train_index])
209     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
210 
211 print(np.mean(accuracies))
212 
213 """### 3. TF-IDF"""
214 
215 accuracies = []
216 clf = LogisticRegression(C=40.0, max_iter=1000, random_state=0)
217 kf = KFold(n_splits=5, shuffle=True)
218 for train_index, test_index in kf.split(X):
219     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
220     clf.fit(vectors_train, y[train_index])
221     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
222 
223 print(np.mean(accuracies))
224 
225 """accuracy vs. hyperparameter C"""
226 
227 c_vecs = np.logspace(0, 2, 5)
228 acc = []
229 for c_vec in c_vecs:
230     accuracies = []
231     clf = LogisticRegression(C=c_vec, max_iter=1000, random_state=0)
232     kf = KFold(n_splits=5, shuffle=True)
233     for train_index, test_index in kf.split(X):
234         vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
235         clf.fit(vectors_train, y[train_index])
236         accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
237 
238     acc.append(np.mean(accuracies))
239 
240 plt.xlabel('C')
241 plt.ylabel('Accuracy')
242 plt.plot(c_vecs, acc)
```

```
243
244 """### 4. CountVectorizer with stem tokenizer"""
245
246 accuracies = []
247 clf = LogisticRegression(C=1.0, max_iter=1000, random_state=0)
248 kf = KFold(n_splits=5, shuffle=True)
249 for train_index, test_index in kf.split(X):
250     vectors_train, vectors_test = count_vec_stem(X[train_index], X[test_index
    ])
251     clf.fit(vectors_train, y[train_index])
252     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
253
254 print(np.mean(accuracies))
255
256 """### 5. CountVectorizer with lemma tokenizer"""
257
258 accuracies = []
259 clf = LogisticRegression(C=1.0, max_iter=1000, random_state=0)
260 kf = KFold(n_splits=5, shuffle=True)
261 for train_index, test_index in kf.split(X):
262     vectors_train, vectors_test = count_vec_lemma(X[train_index], X[test_index
    ])
263     clf.fit(vectors_train, y[train_index])
264     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
265
266 print(np.mean(accuracies))
267
268 """## 2. Multinomial Naïve Bayes
269
270 ### 1. CountVectorizer
271 """
272
273 accuracies = []
274 clf = MultinomialNB()
275 kf = KFold(n_splits=5, shuffle=True)
276 for train_index, test_index in kf.split(X):
277     vectors_train, vectors_test = count_vectorizer(X[train_index], X[
    test_index])
278     clf.fit(vectors_train, y[train_index])
279     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
280
281 print(np.mean(accuracies))
282
283 """### 2. CountVectorizer with stop word"""
284
285 accuracies = []
286 clf = MultinomialNB()
287 kf = KFold(n_splits=5, shuffle=True)
288 for train_index, test_index in kf.split(X):
289     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
    test_index])
290     clf.fit(vectors_train, y[train_index])
291     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
292
293 print(np.mean(accuracies))
294
295 """### 3. TF-IDF"""
296
297 accuracies = []
```

```
298 clf = MultinomialNB()
299 kf = KFold(n_splits=5, shuffle=True)
300 for train_index, test_index in kf.split(X):
301     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
302     clf.fit(vectors_train, y[train_index])
303     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
304
305 print(np.mean(accuracies))
306
307 """### 4. CountVectorizer with stem tokenizer"""
308
309 accuracies = []
310 clf = MultinomialNB()
311 kf = KFold(n_splits=5, shuffle=True)
312 for train_index, test_index in kf.split(X):
313     vectors_train, vectors_test = count_vec_stem(X[train_index], X[test_index
    ])
314     clf.fit(vectors_train, y[train_index])
315     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
316
317 print(np.mean(accuracies))
318
319 """### 5. CountVectorizer with lemma tokenizer"""
320
321 accuracies = []
322 clf = MultinomialNB()
323 kf = KFold(n_splits=5, shuffle=True)
324 for train_index, test_index in kf.split(X):
325     vectors_train, vectors_test = count_vec_lemma(X[train_index], X[test_index
    ])
326     clf.fit(vectors_train, y[train_index])
327     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
328
329 print(np.mean(accuracies))
330
331 """## 3. Support Vector Machine
332
333 ### 1. CountVectorizer
334 """
335
336 accuracies = []
337 clf = svm.SVC(kernel='linear', gamma='auto', C=1)
338 kf = KFold(n_splits=5, shuffle=True)
339 for train_index, test_index in kf.split(X):
340     vectors_train, vectors_test = count_vectorizer(X[train_index], X[
    test_index])
341     clf.fit(vectors_train, y[train_index])
342     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
343
344 print(np.mean(accuracies))
345
346 """### 2. CountVectorizer with stop word"""
347
348 accuracies = []
349 clf = svm.SVC(kernel='linear', gamma='auto', C=1)
350 kf = KFold(n_splits=5, shuffle=True)
351 for train_index, test_index in kf.split(X):
352     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
```

```
352 test_index])
353     clf.fit(vectors_train, y[train_index])
354     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
355 
356 print(np.mean(accuracies))
357 
358 """### 3. TF-IDF
359 
360 Linear
361 """
362 
363 accuracies = []
364 clf = svm.SVC(kernel='linear', gamma='auto', C=1)
365 kf = KFold(n_splits=5, shuffle=True)
366 for train_index, test_index in kf.split(X):
367     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
368     clf.fit(vectors_train, y[train_index])
369     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
370 
371 print(np.mean(accuracies))
372 
373 """Non-Linear"""
374 
375 accuracies = []
376 clf = svm.SVC(gamma='auto', C=1)
377 kf = KFold(n_splits=5, shuffle=True)
378 for train_index, test_index in kf.split(X):
379     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
380     clf.fit(vectors_train, y[train_index])
381     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
382 
383 print(np.mean(accuracies))
384 
385 """### 4. CountVectorizer with stem tokenizer"""
386 
387 accuracies = []
388 clf = svm.SVC(kernel='linear', gamma='auto', C=1)
389 kf = KFold(n_splits=5, shuffle=True)
390 for train_index, test_index in kf.split(X):
391     vectors_train, vectors_test = count_vec_stem(X[train_index], X[test_index
    ])
392     clf.fit(vectors_train, y[train_index])
393     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
394 
395 print(np.mean(accuracies))
396 
397 """### 5. CountVectorizer with lemma tokenizer"""
398 
399 accuracies = []
400 clf = svm.SVC(kernel='linear', gamma='auto', C=1)
401 kf = KFold(n_splits=5, shuffle=True)
402 for train_index, test_index in kf.split(X):
403     vectors_train, vectors_test = count_vec_lemma(X[train_index], X[test_index
    ])
404     clf.fit(vectors_train, y[train_index])
405     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
```

```
406
407 print(np.mean(accuracies))
408
409 """## 4. Random Forest
410
411 ### 1. CountVectorizer
412 """
413
414 accuracies = []
415 clf = RandomForestClassifier(max_depth=2, random_state=0)
416 kf = KFold(n_splits=5, shuffle=True)
417 for train_index, test_index in kf.split(X):
418     vectors_train, vectors_test = count_vectorizer(X[train_index], X[
    test_index])
419     clf.fit(vectors_train, y[train_index])
420     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
421
422 print(np.mean(accuracies))
423
424 """### 2. CountVectorizer with stop word"""
425
426 accuracies = []
427 clf = RandomForestClassifier(max_depth=2, random_state=0)
428 kf = KFold(n_splits=5, shuffle=True)
429 for train_index, test_index in kf.split(X):
430     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
    test_index])
431     clf.fit(vectors_train, y[train_index])
432     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
433
434 print(np.mean(accuracies))
435
436 """### 3. TF-IDF"""
437
438 accuracies = []
439 clf = RandomForestClassifier(max_depth=2, random_state=0)
440 kf = KFold(n_splits=5, shuffle=True)
441 for train_index, test_index in kf.split(X):
442     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
443     clf.fit(vectors_train, y[train_index])
444     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
445
446 print(np.mean(accuracies))
447
448 """### 4. CountVectorizer with stem tokenizer"""
449
450 accuracies = []
451 clf = RandomForestClassifier(max_depth=2, random_state=0)
452 kf = KFold(n_splits=5, shuffle=True)
453 for train_index, test_index in kf.split(X):
454     vectors_train, vectors_test = count_vec_stem(X[train_index], X[test_index
    ])
455     clf.fit(vectors_train, y[train_index])
456     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
457
458 print(np.mean(accuracies))
459
460 """### 5. CountVectorizer with lemma tokenizer"""
```

```
461
462 accuracies = []
463 clf = RandomForestClassifier(max_depth=2, random_state=0)
464 kf = KFold(n_splits=5, shuffle=True)
465 for train_index, test_index in kf.split(X):
466     vectors_train, vectors_test = count_vec_lemma(X[train_index], X[test_index
    ])
467     clf.fit(vectors_train, y[train_index])
468     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
469
470 print(np.mean(accuracies))
471
472 """## 5. Decision Tree
473
474 ### 1. CountVectorizer
475 """
476
477 accuracies = []
478 clf = DecisionTreeClassifier(random_state=0)
479 kf = KFold(n_splits=5, shuffle=True)
480 for train_index, test_index in kf.split(X):
481     vectors_train, vectors_test = count_vectorizer(X[train_index], X[
    test_index])
482     clf.fit(vectors_train, y[train_index])
483     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
484
485 print(np.mean(accuracies))
486
487 """### 2. CountVectorizer with stop word"""
488
489 accuracies = []
490 clf = DecisionTreeClassifier(random_state=0)
491 kf = KFold(n_splits=5, shuffle=True)
492 for train_index, test_index in kf.split(X):
493     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
    test_index])
494     clf.fit(vectors_train, y[train_index])
495     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
496
497 print(np.mean(accuracies))
498
499 """### 3. TF-IDF"""
500
501 accuracies = []
502 clf = DecisionTreeClassifier(random_state=0)
503 kf = KFold(n_splits=5, shuffle=True)
504 for train_index, test_index in kf.split(X):
505     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
506     clf.fit(vectors_train, y[train_index])
507     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
508
509 print(np.mean(accuracies))
510
511 """### 4. CountVectorizer with stem tokenizer"""
512
513 accuracies = []
514 clf = DecisionTreeClassifier(random_state=0)
515 kf = KFold(n_splits=5, shuffle=True)
```

```
516 for train_index, test_index in kf.split(X):
517     vectors_train, vectors_test = count_vec_stem(X[train_index], X[test_index
    ])
518     clf.fit(vectors_train, y[train_index])
519     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
520 
521 print(np.mean(accuracies))
522 
523 """### 5. CountVectorizer with lemma tokenizer"""
524 
525 accuracies = []
526 clf = DecisionTreeClassifier(random_state=0)
527 kf = KFold(n_splits=5, shuffle=True)
528 for train_index, test_index in kf.split(X):
529     vectors_train, vectors_test = count_vec_lemma(X[train_index], X[test_index
    ])
530     clf.fit(vectors_train, y[train_index])
531     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
532 
533 print(np.mean(accuracies))
534 
535 """## 6. Ada Boost
536 
537 ### 1. CountVectorizer
538 """
539 
540 accuracies = []
541 clf = AdaBoostClassifier(n_estimators=100, learning_rate=0.5, random_state=0)
542 kf = KFold(n_splits=5, shuffle=True)
543 for train_index, test_index in kf.split(X):
544     vectors_train, vectors_test = count_vectorizer(X[train_index], X[
    test_index])
545     clf.fit(vectors_train, y[train_index])
546     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
547 
548 print(np.mean(accuracies))
549 
550 """### 2. CountVectorizer with stop word"""
551 
552 accuracies = []
553 clf = AdaBoostClassifier(n_estimators=100, learning_rate=0.5, random_state=0)
554 kf = KFold(n_splits=5, shuffle=True)
555 for train_index, test_index in kf.split(X):
556     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
    test_index])
557     clf.fit(vectors_train, y[train_index])
558     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
559 
560 print(np.mean(accuracies))
561 
562 """### 3. TF-IDF"""
563 
564 # Commented out IPython magic to ensure Python compatibility.
565 # find the best hyperparameters
566 clf = AdaBoostClassifier(random_state=0)
567 model = make_pipeline(clf)
568 
569 param_grid = {'adaboostclassifier__n_estimators': [50, 100],
570               'adaboostclassifier__learning_rate': [0.1, 0.5, 1]}
```

```
grid = GridSearchCV(model, param_grid)

X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=0.8, test_size=0.2, random_state=0)
vectors_train, vectors_test = tfidf_vectorizer(X_train, X_test)
# %time grid.fit(vectors_train, y_train)
print(grid.best_params_)

accuracies = []
clf = AdaBoostClassifier(n_estimators=100, learning_rate=0.5, random_state=0)
kf = KFold(n_splits=5, shuffle=True)
for train_index, test_index in kf.split(X):
    vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[test_index])
    clf.fit(vectors_train, y[train_index])
    accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(vectors_test)))

print(np.mean(accuracies))

"""### 4. CountVectorizer with stem tokenizer"""

accuracies = []
clf = AdaBoostClassifier(n_estimators=100, learning_rate=0.5, random_state=0)
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])
    accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(vectors_test)))

print(np.mean(accuracies))

"""### 5. CountVectorizer with lemma tokenizer"""

accuracies = []
clf = AdaBoostClassifier(n_estimators=100, learning_rate=0.5, random_state=0)
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])
    accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(vectors_test)))

print(np.mean(accuracies))

"""## 7. k-Neighbors

### 1. CountVectorizer
"""

accuracies = []
neigh = KNeighborsClassifier(n_neighbors=3)
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])
    neigh.fit(vectors_train, y[train_index])
    accuracies.append(metrics.accuracy_score(y[test_index], neigh.predict(vectors_test)))

```

```
print(np.mean(accuracies))

"""### 2. CountVectorizer with stop word"""

accuracies = []
neigh = KNeighborsClassifier(n_neighbors=3)
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])
    neigh.fit(vectors_train, y[train_index])
    accuracies.append(metrics.accuracy_score(y[test_index], neigh.predict(
vectors_test)))

print(np.mean(accuracies))

"""### 3. TF-IDF"""

accuracies = []
neigh = KNeighborsClassifier(n_neighbors=3)
kf = KFold(n_splits=5, shuffle=True)
for train_index, test_index in kf.split(X):
    vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
test_index])
    neigh.fit(vectors_train, y[train_index])
    accuracies.append(metrics.accuracy_score(y[test_index], neigh.predict(
vectors_test)))

print(np.mean(accuracies))

"""### 4. CountVectorizer with stem tokenizer"""

accuracies = []
neigh = KNeighborsClassifier(n_neighbors=3)
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
])
    neigh.fit(vectors_train, y[train_index])
    accuracies.append(metrics.accuracy_score(y[test_index], neigh.predict(
vectors_test)))

print(np.mean(accuracies))

"""### 5. CountVectorizer with lemma tokenizer"""

accuracies = []
neigh = KNeighborsClassifier(n_neighbors=3)
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
])
    neigh.fit(vectors_train, y[train_index])
    accuracies.append(metrics.accuracy_score(y[test_index], neigh.predict(
vectors_test)))

print(np.mean(accuracies))

"""## 8. Neural Network

### 1. CountVectorizer
"""

```

```
680 accuracies = []
681 kf = KFold(n_splits=5, shuffle=True)
682 for train_index, test_index in kf.split(X):
683     vectors_train, vectors_test = count_vectorizer(X[train_index], X[
    test_index])
684     clf = MLPClassifier(random_state=0, max_iter=300).fit(vectors_train, y[
    train_index])
685     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
686
687 print(np.mean(accuracies))
688
689 """### 2. CountVectorizer with stop word"""
690
691 accuracies = []
692 kf = KFold(n_splits=5, shuffle=True)
693 for train_index, test_index in kf.split(X):
694     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
    test_index])
695     clf = MLPClassifier(random_state=0, max_iter=300).fit(vectors_train, y[
    train_index])
696     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
697
698 print(np.mean(accuracies))
699
700 """### 3. TF-IDF"""
701
702 accuracies = []
703 kf = KFold(n_splits=5, shuffle=True)
704 for train_index, test_index in kf.split(X):
705     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
706     clf = MLPClassifier(random_state=0, max_iter=300).fit(vectors_train, y[
    train_index])
707     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
708
709 print(np.mean(accuracies))
710
711 """### 4. CountVectorizer with stem tokenizer"""
712
713 accuracies = []
714 kf = KFold(n_splits=5, shuffle=True)
715 for train_index, test_index in kf.split(X):
716     vectors_train, vectors_test = count_vec_stem(X[train_index], X[test_index
    ])
717     clf = MLPClassifier(random_state=0, max_iter=300).fit(vectors_train, y[
    train_index])
718     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
719
720 print(np.mean(accuracies))
721
722 """### 5. CountVectorizer with lemma tokenizer"""
723
724 accuracies = []
725 kf = KFold(n_splits=5, shuffle=True)
726 for train_index, test_index in kf.split(X):
727     vectors_train, vectors_test = count_vec_lemma(X[train_index], X[test_index
    ])
728     clf = MLPClassifier(random_state=0, max_iter=300).fit(vectors_train, y[
    train_index])
```

```
729     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
730
731 print(np.mean(accuracies))
```

```
# -*- coding: utf-8 -*-
"""Additional Classifiers Testing.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/16E3avMbgZz1YnYvo1uuc4HLzgHZiaDCQ

<center><h1>Mini Project 2 - Bernoulli Naïve Bayes</h1></center>
This file consists two parts:

In the first part, it measures the accuracies and time spent of the Logistic Regression based on the output of 5 different classifiers. The effect of data normalization is also measured.

In the second part, it measures the accuracies and time spent of the remaining classifiers given that a TF-IDF vectorizer is used. In the end of the second part, we also did some tests on the Bernoulli Naïve Bayes implemented by sklearn upon all different vectorizers.

<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

!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

"""Additional classifiers:
1. Logistic Regression
2. Multinomial Naïve Bayes
3. Support Vector Machine
```

```
 59 4. Random Forest
 60 5. Decision Tree
 61 6. Ada Boost
 62 7. k-Neighbors
 63 8. Neural Network
 64 """
 65
 66 from sklearn.linear_model import LogisticRegression
 67 from sklearn.naive_bayes import MultinomialNB
 68 from sklearn import svm
 69 from sklearn.ensemble import RandomForestClassifier
 70 from sklearn.tree import DecisionTreeClassifier
 71 from sklearn.ensemble import AdaBoostClassifier
 72 from sklearn.neighbors import KNeighborsClassifier
 73 from sklearn.neural_network import MLPClassifier
 74
 75 reddit_dataset = pd.read_csv(path+"train.csv")
 76 reddit_test = pd.read_csv(path+"test.csv")
 77
 78 X = reddit_dataset['body']
 79 y = reddit_dataset['subreddit']
 80
 81 """# Define Vectorizer
 82 ### (To vectorize the text-based data to numerical features)
 83
 84 1. CountVectorizer
 85 1) Use "CountVectorizer" to transform text data to feature vectors.
 86 2) Normalize your feature vectors
 87 """
 88
 89 def count_vectorizer(X_train, X_test, normalize=True):
 90     vectorizer = CountVectorizer()
 91     vectors_train = vectorizer.fit_transform(X_train)
 92     vectors_test = vectorizer.transform(X_test)
 93
 94     if normalize:
 95         normalizer_train = Normalizer().fit(X=vectors_train)
 96         vectors_train = normalizer_train.transform(vectors_train)
 97         vectors_test = normalizer_train.transform(vectors_test)
 98
 99     return vectors_train, vectors_test
100
101 """2. CountVectorizer with stop word
102 1) Use "CountVectorizer" with stop word to transform text data to vector.
103 2) Normalize your feature vectors
104 """
105
106 def count_vec_with_sw(X_train, X_test, normalize=True, features_5k=False):
107     stop_words = text.ENGLISH_STOP_WORDS
108     if features_5k:
109         vectorizer = CountVectorizer(stop_words=stop_words, max_features=5000)
110     else:
111         vectorizer = CountVectorizer(stop_words=stop_words)
112     vectors_train_stop = vectorizer.fit_transform(X_train)
113     vectors_test_stop = vectorizer.transform(X_test)
114
115     if normalize:
116         normalizer_train = Normalizer().fit(X=vectors_train_stop)
117         vectors_train_stop= normalizer_train.transform(vectors_train_stop)
118         vectors_test_stop = normalizer_train.transform(vectors_test_stop)
119
120     return vectors_train_stop, vectors_test_stop
121
```

```
122 """3. TF-IDF
123 1) use "TfidfVectorizer" to weight features based on your train set.
124 2) Normalize your feature vectors
125 """
126
127 def tfidf_vectorizer(X_train, X_test, normalize=True):
128     tf_idf_vectorizer = TfidfVectorizer()
129     vectors_train_idf = tf_idf_vectorizer.fit_transform(X_train)
130     vectors_test_idf = tf_idf_vectorizer.transform(X_test)
131
132     if normalize:
133         normalizer_train = Normalizer().fit(X=vectors_train_idf)
134         vectors_train_idf= normalizer_train.transform(vectors_train_idf)
135         vectors_test_idf = normalizer_train.transform(vectors_test_idf)
136
137     return vectors_train_idf, vectors_test_idf
138
139 """4. CountVectorizer with stem tokenizer
140 1) Use "StemTokenizer" to transform text data to vector.
141 2) Normalize your feature vectors
142 """
143
144 class StemTokenizer:
145      def __init__(self):
146        self.wnl =PorterStemmer()
147      def __call__(self, doc):
148        return [self.wnl.stem(t) for t in word_tokenize(doc) if t.isalpha()]
149
150
151 def count_vec_stem(X_train, X_test, normalize=True):
152     vectorizer = CountVectorizer(tokenizer=StemTokenizer())
153     vectors_train_stem = vectorizer.fit_transform(X_train)
154     vectors_test_stem = vectorizer.transform(X_test)
155
156     if normalize:
157         normalizer_train = Normalizer().fit(X=vectors_train_stem)
158         vectors_train_stem= normalizer_train.transform(vectors_train_stem)
159         vectors_test_stem = normalizer_train.transform(vectors_test_stem)
160
161     return vectors_train_stem, vectors_test_stem
162
163 """5. CountVectorizer with lemma tokenizer
164 1) Use "LemmaTokenizer" to transform text data to vector.
165 2) Normalize your feature vectors
166 """
167
168 def get_wordnet_pos(word):
169     """Map POS tag to first character lemmatize() accepts"""
170     tag = nltk.pos_tag([word])[0][1][0].upper()
171     tag_dict = {"J": wordnet.ADJ,
172                 "N": wordnet.NOUN,
173                 "V": wordnet.VERB,
174                 "R": wordnet.ADV}
175     return tag_dict.get(tag, wordnet.NOUN)
176
177
178 class LemmaTokenizer:
179      def __init__(self):
180        self.wnl = WordNetLemmatizer()
181      def __call__(self, doc):
182        return [self.wnl.lemmatize(t,pos =get_wordnet_pos(t)) for t in
    word_tokenize(doc) if t.isalpha()]
183
```

```
184
185 def count_vec_lemma(X_train, X_test, normalize=True):
186     vectorizer = CountVectorizer(tokenizer=LemmaTokenizer())
187     vectors_train_lemma = vectorizer.fit_transform(X_train)
188     vectors_test_lemma = vectorizer.transform(X_test)
189
190     if normalize:
191         normalizer_train = Normalizer().fit(X=vectors_train_lemma)
192         vectors_train_lemma= normalizer_train.transform(vectors_train_lemma)
193         vectors_test_lemma = normalizer_train.transform(vectors_test_lemma)
194
195     return vectors_train_lemma, vectors_test_lemma
196
197 """# Measure Accuracies and Time Spent of different classifiers using K-fold
    Validation
198
199 ## 1. Logistic Regression
200
201 ### 1. CountVectorizer
202 """
203
204 tic = time()
205 accuracies = []
206 clf = LogisticRegression(C=1.0, max_iter=1000, random_state=0)
207 kf = KFold(n_splits=5, shuffle=True)
208 for train_index, test_index in kf.split(X):
209     vectors_train, vectors_test = count_vectorizer(X[train_index], X[
    test_index])
210     clf.fit(vectors_train, y[train_index])
211     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
212
213 print("\t- Logestic Regression + CountVectorizer + Normalize -\nAccuracy: {}%\
    tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
214
215 tic = time()
216 accuracies = []
217 clf = LogisticRegression(C=1.0, max_iter=1000, random_state=0)
218 kf = KFold(n_splits=5, shuffle=True)
219 for train_index, test_index in kf.split(X):
220     vectors_train, vectors_test = count_vectorizer(X[train_index], X[
    test_index], False)
221     clf.fit(vectors_train, y[train_index])
222     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
223
224 print("\t- Logestic Regression + CountVectorizer + Unnormalize -\nAccuracy
    : {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
225
226 """### 2. CountVectorizer with stop word"""
227
228 tic = time()
229 accuracies = []
230 clf = LogisticRegression(C=1.0, max_iter=1000, random_state=0)
231 kf = KFold(n_splits=5, shuffle=True)
232 for train_index, test_index in kf.split(X):
233     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
    test_index])
234     clf.fit(vectors_train, y[train_index])
235     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
236
237 print("\t- Logestic Regression + CountVectorizer with stop word + Normalize -\
```

```
237 nAccuracy: {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
238
239 tic = time()
240 accuracies = []
241 clf = LogisticRegression(C=1.0, max_iter=1000, random_state=0)
242 kf = KFold(n_splits=5, shuffle=True)
243 for train_index, test_index in kf.split(X):
244     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
    test_index], False)
245     clf.fit(vectors_train, y[train_index])
246     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
247
248 print("\t- Logestic Regression + CountVectorizer with stop word + Unnormalize
     -\nAccuracy: {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
249
250 """### 3. TF-IDF"""
251
252 tic = time()
253 accuracies = []
254 clf = LogisticRegression(C=40.0, max_iter=1000, random_state=0)
255 kf = KFold(n_splits=5, shuffle=True)
256 for train_index, test_index in kf.split(X):
257     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
258     clf.fit(vectors_train, y[train_index])
259     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
260
261 print("\t- Logestic Regression + TF-IDF Vectorizer + Normalize -\nAccuracy
    : {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
262
263 tic = time()
264 accuracies = []
265 clf = LogisticRegression(C=40.0, max_iter=1000, random_state=0)
266 kf = KFold(n_splits=5, shuffle=True)
267 for train_index, test_index in kf.split(X):
268     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index], False)
269     clf.fit(vectors_train, y[train_index])
270     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
271
272 print("\t- Logestic Regression + TF-IDF Vectorizer + Unnormalize -\nAccuracy
    : {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
273
274 """### 4. CountVectorizer with stem tokenizer"""
275
276 tic = time()
277 accuracies = []
278 clf = LogisticRegression(C=1.0, max_iter=1000, random_state=0)
279 kf = KFold(n_splits=5, shuffle=True)
280 for train_index, test_index in kf.split(X):
281     vectors_train, vectors_test = count_vec_stem(X[train_index], X[test_index
    ])
282     clf.fit(vectors_train, y[train_index])
283     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
284
285 print("\t- Logestic Regression + CountVectorizer with stem tokenizer +
    Normalize -\nAccuracy: {}%\tTime Spent: {}s".format(np.mean(accuracies), time
    ()-tic))
286
```

```
tic = time()
accuracies = []
clf = LogisticRegression(C=1.0, max_iter=1000, random_state=0)
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], False)
    clf.fit(vectors_train, y[train_index])
    accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(vectors_test)))

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

"""### 5. CountVectorizer with lemma tokenizer"""

tic = time()
accuracies = []
clf = LogisticRegression(C=1.0, max_iter=1000, random_state=0)
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])
    accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(vectors_test)))

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

tic = time()
accuracies = []
clf = LogisticRegression(C=1.0, max_iter=1000, random_state=0)
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], False)
    clf.fit(vectors_train, y[train_index])
    accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(vectors_test)))

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

"""## 2. Multinomial Naïve Bayes"""

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

print("\t- Multinomial Naïve Bayes + TF-IDF Vectorizer + Normalize -\nAccuracy: {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))

```

```
335 """## 3. Support Vector Machine
336 
337 Linear
338 """
339 
340 tic = time()
341 accuracies = []
342 clf = svm.SVC(kernel='linear', gamma='auto', C=1)
343 kf = KFold(n_splits=5, shuffle=True)
344 for train_index, test_index in kf.split(X):
345     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
346     clf.fit(vectors_train, y[train_index])
347     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
348 
349 print("\t- Linear Support Vector Machine + TF-IDF Vectorizer + Normalize -\n
    Accuracy: {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
350 
351 """## 4. Random Forest"""
352 
353 tic = time()
354 accuracies = []
355 clf = RandomForestClassifier(max_depth=2, random_state=0)
356 kf = KFold(n_splits=5, shuffle=True)
357 for train_index, test_index in kf.split(X):
358     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
359     clf.fit(vectors_train, y[train_index])
360     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
361 
362 print("\t- Random Forest + TF-IDF Vectorizer + Normalize -\nAccuracy: {}%\t
    Time Spent: {}s".format(np.mean(accuracies), time()-tic))
363 
364 """## 5. Decision Tree"""
365 
366 tic = time()
367 accuracies = []
368 clf = DecisionTreeClassifier(random_state=0)
369 kf = KFold(n_splits=5, shuffle=True)
370 for train_index, test_index in kf.split(X):
371     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
372     clf.fit(vectors_train, y[train_index])
373     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
374 
375 print("\t- Decision Tree + TF-IDF Vectorizer + Normalize -\nAccuracy: {}%\t
    Time Spent: {}s".format(np.mean(accuracies), time()-tic))
376 
377 """## 6. Ada Boost"""
378 
379 tic = time()
380 accuracies = []
381 clf = AdaBoostClassifier(n_estimators=100, learning_rate=0.5, random_state=0)
382 kf = KFold(n_splits=5, shuffle=True)
383 for train_index, test_index in kf.split(X):
384     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
385     clf.fit(vectors_train, y[train_index])
386     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
```

```
387
388 print("\t- Ada Boost + TF-IDF Vectorizer + Normalize -\nAccuracy: {}%\tTime
    Spent: {}s".format(np.mean(accuracies), time()-tic))
389
390 """## 7. k-Neighbors"""
391
392 tic = time()
393 accuracies = []
394 neigh = KNeighborsClassifier(n_neighbors=3)
395 kf = KFold(n_splits=5, shuffle=True)
396 for train_index, test_index in kf.split(X):
397     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
398     neigh.fit(vectors_train, y[train_index])
399     accuracies.append(metrics.accuracy_score(y[test_index], neigh.predict(
    vectors_test)))
400
401 print("\t- k-Neighbors + TF-IDF Vectorizer + Normalize -\nAccuracy: {}%\tTime
    Spent: {}s".format(np.mean(accuracies), time()-tic))
402
403 """## 8. Neural Network"""
404
405 tic = time()
406 accuracies = []
407 kf = KFold(n_splits=5, shuffle=True)
408 for train_index, test_index in kf.split(X):
409     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
410     clf = MLPClassifier(random_state=0, max_iter=300).fit(vectors_train, y[
    train_index])
411     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
412
413 print("\t- Neural Network + TF-IDF Vectorizer + Normalize -\nAccuracy: {}%\t
    Time Spent: {}s".format(np.mean(accuracies), time()-tic))
414
415 """## 9. Bernoulli Naïve Bayes (Sklearn Version)
416 <h2>This part is only used to test and compare the performance of the
    Bernoulli Naïve Bayes implemented by ourselves.</h2>
417 """
418
419 from sklearn.naive_bayes import BernoulliNB
420
421 """### 1. CountVectorizer"""
422
423 tic = time()
424 accuracies = []
425 kf = KFold(n_splits=5, shuffle=True)
426 for train_index, test_index in kf.split(X):
427     vectors_train, vectors_test = count_vectorizer(X[train_index], X[
    test_index])
428     clf = BernoulliNB().fit(vectors_train, y[train_index])
429     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
430
431 print("\t- BernoulliNB + CountVectorizer + Normalize -\nAccuracy: {}\tTime
    Spent: {}s".format(np.mean(accuracies), time()-tic))
432
433 """### 2. CountVectorizer with stop word"""
434
435 tic = time()
436 accuracies = []
437 kf = KFold(n_splits=5, shuffle=True)
```

```
438 for train_index, test_index in kf.split(X):
439     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
    test_index])
440     clf = BernoulliNB().fit(vectors_train, y[train_index])
441     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
442 
443 print("\t- BernoulliNB + CountVectorizer with stop word + Normalize -\n
    Accuracy: {}\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
444 
445 """### 3. CountVectorizer with stop word, max_features = 5000"""
446 
447 tic = time()
448 accuracies = []
449 kf = KFold(n_splits=5, shuffle=True)
450 for train_index, test_index in kf.split(X):
451     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
    test_index], features_5k=True)
452     clf = BernoulliNB().fit(vectors_train, y[train_index])
453     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
454 
455 print("\t- BernoulliNB + CountVectorizer with stop word, max_features = 5000
     + Normalize -\nAccuracy: {}\tTime Spent: {}s".format(np.mean(accuracies),
    time()-tic))
456 
457 """### 4. TF-IDF"""
458 
459 tic = time()
460 accuracies = []
461 kf = KFold(n_splits=5, shuffle=True)
462 for train_index, test_index in kf.split(X):
463     vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[
    test_index])
464     clf = BernoulliNB().fit(vectors_train, y[train_index])
465     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
466 
467 print("\t- BernoulliNB + TF-IDF Vectorizer + Normalize -\nAccuracy: {}\tTime
    Spent: {}s".format(np.mean(accuracies), time()-tic))
468 
469 """### 5. CountVectorizer with stem tokenizer"""
470 
471 tic = time()
472 accuracies = []
473 kf = KFold(n_splits=5, shuffle=True)
474 for train_index, test_index in kf.split(X):
475     vectors_train, vectors_test = count_vec_stem(X[train_index], X[test_index
    ])
476     clf = BernoulliNB().fit(vectors_train, y[train_index])
477     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
478 
479 print("\t- BernoulliNB + CountVectorizer with stem tokenizer + Normalize -\n
    Accuracy: {}\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
480 
481 """### 6. CountVectorizer with lemma tokenizer"""
482 
483 tic = time()
484 accuracies = []
485 kf = KFold(n_splits=5, shuffle=True)
486 for train_index, test_index in kf.split(X):
487     vectors_train, vectors_test = count_vec_lemma(X[train_index], X[test_index
```

```
487 ])
488     clf = BernoulliNB().fit(vectors_train, y[train_index])
489     accuracies.append(metrics.accuracy_score(y[test_index], clf.predict(
    vectors_test)))
490
491 print("\t- BernoulliNB + CountVectorizer with lemma tokenizer + Normalize -\n
    Accuracy: {}\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
```

```
# -*- coding: utf-8 -*-
"""Bernoulli Naïve Bayes.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/19asN10XEleCYuHFT9Yao8DAXamARVxFy

<center><h1>Mini Project 2 - Bernoulli Naïve Bayes</h1>
<h4>The hyperparameters and models used in this file are chosen based on the findings in the testing file.</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 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)

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):
    stop_words = text.ENGLISH_STOP_WORDS
    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
"""

def get_wordnet_pos(word):
    """Map POS tag to first character lemmatize() accepts"""
    tag = nltk.pos_tag([word])[0][1][0].upper()
```

```
126     tag_dict = {"J": wordnet.ADJ,
127                 "N": wordnet.NOUN,
128                 "V": wordnet.VERB,
129                 "R": wordnet.ADV}
130     return tag_dict.get(tag, wordnet.NOUN)
131 
132 
133 class LemmaTokenizer:
134      def __init__(self):
135        self.wnl = WordNetLemmatizer()
136      def __call__(self, doc):
137        return [self.wnl.lemmatize(t,pos =get_wordnet_pos(t)) for t in
    word_tokenize(doc) if t.isalpha()]
138 
139 
140 def count_vec_lemma(X_train, X_test):
141     vectorizer = CountVectorizer(tokenizer=LemmaTokenizer(), binary=True)
142     vectors_train_lemma = vectorizer.fit_transform(X_train)
143     vectors_test_lemma = vectorizer.transform(X_test)
144 
145     return vectors_train_lemma, vectors_test_lemma
146 
147 """# Bernoulli Naïve Bayes Classifier"""
148 
149 # Bernoulli Naïve Bayes
150 class BernoulliNB:
151     '''
152         This is the Bernoulli Naïve Bayes class, containing fit, perdict and
    accu_eval functions,
153         as well as many other useful functions.
154     '''
155 
156     def __init__(self, laplace):
157         self.laplace = laplace  # true for performing Laplace smoothing
158         self.le = LabelEncoder()  # encoder for classes
159 
160     def fit(self, X, y):
161         '''
162             This function takes the training data X and its corresponding
    labels vector y as input,
163             and execute the model training.
164 
165             X - features of traning data
166             y - class labels
167         '''
168         # Laplace smoothing paramerters
169         num = 0
170         den = 0
171         if self.laplace:
172             num += 1
173             den += 2
174 
175         # encode the text-based class type to numerical values
176         le = self.le
177         le.fit(y)
178         y_label = le.transform(y)
179         n_k = len(le.classes_)  # number of classes
180         n_j = X.shape[1]  # number of features
181         N = len(y)  # number of samples
182 
183         theta_k = np.zeros(n_k)  # probability of class k
184         theta_j_k = np.zeros((n_k, n_j))  # probability of feature j given
    class k
```

```
185
186         # compute theta values
187         for k in range(n_k):
188             count_k = (y_label==k).sum()
189             theta_k[k] = count_k / N
190             for j in range(n_j):
191                 theta_j_k[k][j] = (X[y_label==k, j].sum()+num) / (count_k+den)
192
193         # store the theta values to this instance
194         self.theta_k = theta_k
195         self.theta_j_k = theta_j_k
196         print("Finished fitting...")
197
198     def predict(self, X):
199         '''
200             This function takes a set of data as input and outputs predicted
    labels for the input points.
201         '''
202         le = self.le
203         theta_k = self.theta_k
204         theta_j_k = self.theta_j_k
205
206         # this part works the same as the pseudo-code provided in Lecture 12
207         # but matrix multiplication is much faster than nested loops
208         i_m = np.zeros_like(X)  # identity matrix
209         # predict classes
210         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)
211
212         # transform back to text-based values
213         y_pred = le.inverse_transform(y_pred)
214         return y_pred
215
216 """### 1. K-fold validation using CountVectorizer"""
217
218 accuracies = []
219 clf = BernoulliNB(laplace=True)
220 kf = KFold(n_splits=5, shuffle=True)
221 for train_index, test_index in kf.split(X):
222     vectors_train, vectors_test = count_vectorizer(X[train_index], X[
    test_index])
223     clf.fit(vectors_train, y[train_index])
224     a_s = metrics.accuracy_score(y[test_index], clf.predict(vectors_test))
225     print(a_s)
226     accuracies.append(a_s)
227
228 print(np.mean(accuracies))
229
230 """### 2. K-fold validation using CountVectorizer with stop word, max_features
    =5000"""
231
232 # with max_features=5000
233 accuracies = []
234 clf = BernoulliNB(laplace=True)
235 kf = KFold(n_splits=5, shuffle=True)
236 for train_index, test_index in kf.split(X):
237     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
    test_index])
238     clf.fit(vectors_train, y[train_index])
239     a_s = metrics.accuracy_score(y[test_index], clf.predict(vectors_test))
240     print(a_s)
241     accuracies.append(a_s)
242
```

```
print(np.mean(accuracies))

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

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(np.mean(accuracies))

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

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(np.mean(accuracies))

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

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(np.mean(accuracies))
```

```
# -*- 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)

```

```
63 1. CountVectorizer
64 1) Use "CountVectorizer" to transform text data to feature vectors.
65 2) Normalize your feature vectors
66 """
67
68 def count_vectorizer(X_train, X_test):
69     vectorizer = CountVectorizer(binary=True)
70     vectors_train = vectorizer.fit_transform(X_train)
71     vectors_test = vectorizer.transform(X_test)
72
73     return vectors_train, vectors_test
74
75 """2. CountVectorizer with stop word
76 1) Use "CountVectorizer" with stop word to transform text data to vector.
77 2) Normalize your feature vectors
78 """
79
80 def count_vec_with_sw(X_train, X_test, max_features):
81     stop_words = text.ENGLISH_STOP_WORDS
82     if max_features:
83         vectorizer = CountVectorizer(stop_words=stop_words, binary=True,
   max_features=5000)
84     else:
85         vectorizer = CountVectorizer(stop_words=stop_words, binary=True)
86     vectors_train_stop = vectorizer.fit_transform(X_train)
87     vectors_test_stop = vectorizer.transform(X_test)
88
89     return vectors_train_stop, vectors_test_stop
90
91 """3. TF-IDF
92 1) use "TfidfVectorizer" to weight features based on your train set.
93 2) Normalize your feature vectors
94 """
95
96 def tfidf_vectorizer(X_train, X_test):
97     tf_idf_vectorizer = TfidfVectorizer(binary=True)
98     vectors_train_idf = tf_idf_vectorizer.fit_transform(X_train)
99     vectors_test_idf = tf_idf_vectorizer.transform(X_test)
100
101     return vectors_train_idf, vectors_test_idf
102
103 """4. CountVectorizer with stem tokenizer
104 1) Use "StemTokenizer" to transform text data to vector.
105 2) Normalize your feature vectors
106 """
107
108 class StemTokenizer:
109      def __init__(self):
110        self.wnl =PorterStemmer()
111      def __call__(self, doc):
112        return [self.wnl.stem(t) for t in word_tokenize(doc) if t.isalpha()]
113
114
115 def count_vec_stem(X_train, X_test):
116     vectorizer = CountVectorizer(tokenizer=StemTokenizer(), binary=True)
117     vectors_train_stem = vectorizer.fit_transform(X_train)
118     vectors_test_stem = vectorizer.transform(X_test)
119
120     return vectors_train_stem, vectors_test_stem
121
122 """5. CountVectorizer with lemma tokenizer
123 1) Use "LemmaTokenizer" to transform text data to vector.
124 2) Normalize your feature vectors
```

```
125 """
126
127 def get_wordnet_pos(word):
128     """Map POS tag to first character lemmatize() accepts"""
129     tag = nltk.pos_tag([word])[0][1][0].upper()
130     tag_dict = {"J": wordnet.ADJ,
131                 "N": wordnet.NOUN,
132                 "V": wordnet.VERB,
133                 "R": wordnet.ADV}
134     return tag_dict.get(tag, wordnet.NOUN)
135
136
137 class LemmaTokenizer:
138      def __init__(self):
139        self.wnl = WordNetLemmatizer()
140      def __call__(self, doc):
141        return [self.wnl.lemmatize(t,pos =get_wordnet_pos(t)) for t in
    word_tokenize(doc) if t.isalpha()]
142
143
144 def count_vec_lemma(X_train, X_test):
145     vectorizer = CountVectorizer(tokenizer=LemmaTokenizer(), binary=True)
146     vectors_train_lemma = vectorizer.fit_transform(X_train)
147     vectors_test_lemma = vectorizer.transform(X_test)
148
149     return vectors_train_lemma, vectors_test_lemma
150
151 """# Bernoulli Naïve Bayes Classifier"""
152
153 # Bernoulli Naïve Bayes
154 class BernoulliNB:
155     '''
156         This is the Bernoulli Naïve Bayes class, containing fit, perdict and
    accu_eval functions,
157         as well as many other useful functions.
158     '''
159
160     def __init__(self, laplace):
161         self.laplace = laplace  # true for performing Laplace smoothing
162         self.le = LabelEncoder()  # encoder for classes
163
164     def fit(self, X, y):
165         '''
166             This function takes the training data X and its corresponding
    labels vector y as input,
167             and execute the model training.
168
169             X - features of traning data
170             y - class labels
171         '''
172         # Laplace smoothing paramerters
173         num = 0
174         den = 0
175         if self.laplace:
176             num += 1
177             den += 2
178
179         # encode the text-based class type to numerical values
180         le = self.le
181         le.fit(y)
182         y_label = le.transform(y)
183         n_k = len(le.classes_)  # number of classes
184         n_j = X.shape[1]  # number of features
```

```
185         N = len(y)  # number of samples
186 
187         theta_k = np.zeros(n_k)  # probability of class k
188         theta_j_k = np.zeros((n_k, n_j))  # probability of feature j given
    class k
189 
190         # compute theta values
191         for k in range(n_k):
192             count_k = (y_label==k).sum()
193             theta_k[k] = count_k / N
194             for j in range(n_j):
195                 theta_j_k[k][j] = (X[y_label==k, j].sum()+num) / (count_k+den)
196 
197         # store the theta values to this instance
198         self.theta_k = theta_k
199         self.theta_j_k = theta_j_k
200         # print("Finished fitting...")
201 
202     def predict(self, X):
203         '''
204             This function takes a set of data as input and outputs predicted
    labels for the input points.
205         '''
206         le = self.le
207         theta_k = self.theta_k
208         theta_j_k = self.theta_j_k
209 
210         # this part works the same as the pseudo-code provided in Lecture 12
211         # but matrix multiplication is much faster than nested loops
212         i_m = np.zeros_like(X)  # identity matrix
213         # predict classes
214         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)
215 
216         # transform back to text-based values
217         y_pred = le.inverse_transform(y_pred)
218         return y_pred
219 
220 """### 1. K-fold validation using CountVectorizer"""
221 
222 tic = time()
223 accuracies = []
224 clf = BernoulliNB(laplace=True)
225 kf = KFold(n_splits=5, shuffle=True)
226 for train_index, test_index in kf.split(X):
227     vectors_train, vectors_test = count_vectorizer(X[train_index], X[
    test_index])
228     clf.fit(vectors_train, y[train_index])
229     a_s = metrics.accuracy_score(y[test_index], clf.predict(vectors_test))
230     print(a_s)
231     accuracies.append(a_s)
232 
233 print("\t- Bernoulli Naïve Bayes + CountVectorizer -\nAccuracy: {}%\tTime
    Spent: {}s".format(np.mean(accuracies), time()-tic))
234 
235 """### 2. K-fold validation using CountVectorizer with stop word, max_features
    =5000"""
236 
237 # with max_features=5000
238 tic = time()
239 accuracies = []
240 clf = BernoulliNB(laplace=True)
241 kf = KFold(n_splits=5, shuffle=True)
```

```
242 for train_index, test_index in kf.split(X):
243     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
    test_index], max_features=True)
244     clf.fit(vectors_train, y[train_index])
245     a_s = metrics.accuracy_score(y[test_index], clf.predict(vectors_test))
246     print(a_s)
247     accuracies.append(a_s)
248 
249 print("\t- Bernoulli Naïve Bayes + CountVectorizer with stop word, Max
    Features = 5000 -\nAccuracy: {}%\tTime Spent: {}s".format(np.mean(accuracies
    ), time()-tic))
250 
251 """### 3. K-fold validation using CountVectorizer with stop word"""
252 
253 tic = time()
254 accuracies = []
255 clf = BernoulliNB(laplace=True)
256 kf = KFold(n_splits=5, shuffle=True)
257 for train_index, test_index in kf.split(X):
258     vectors_train, vectors_test = count_vec_with_sw(X[train_index], X[
    test_index])
259     clf.fit(vectors_train, y[train_index])
260     a_s = metrics.accuracy_score(y[test_index], clf.predict(vectors_test))
261     print(a_s)
262     accuracies.append(a_s)
263 
264 print("\t- Bernoulli Naïve Bayes + CountVectorizer with stop word -\nAccuracy
    : {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
265 
266 """### 4. K-fold validation using CountVectorizer with stem tokenizer"""
267 
268 tic = time()
269 accuracies = []
270 clf = BernoulliNB(laplace=True)
271 kf = KFold(n_splits=5, shuffle=True)
272 for train_index, test_index in kf.split(X):
273     vectors_train, vectors_test = count_vec_stem(X[train_index], X[test_index
    ])
274     clf.fit(vectors_train, y[train_index])
275     a_s = metrics.accuracy_score(y[test_index], clf.predict(vectors_test))
276     print(a_s)
277     accuracies.append(a_s)
278 
279 print("\t- Bernoulli Naïve Bayes + CountVectorizer with stem tokenizer -\n
    Accuracy: {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
280 
281 """### 5. K-fold validation using CountVectorizer with lemma tokenizer"""
282 
283 tic = time()
284 accuracies = []
285 clf = BernoulliNB(laplace=True)
286 kf = KFold(n_splits=5, shuffle=True)
287 for train_index, test_index in kf.split(X):
288     vectors_train, vectors_test = count_vec_lemma(X[train_index], X[test_index
    ])
289     clf.fit(vectors_train, y[train_index])
290     a_s = metrics.accuracy_score(y[test_index], clf.predict(vectors_test))
291     print(a_s)
292     accuracies.append(a_s)
293 
294 print("\t- Bernoulli Naïve Bayes + CountVectorizer with lemma tokenizer -\n
    Accuracy: {}%\tTime Spent: {}s".format(np.mean(accuracies), time()-tic))
```

```
# -*- coding: utf-8 -*-
"""Stacking Classifier.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1OKCClcD9Qf2Z59egC0iTBDwc2xkamhsM

<center><h1>Mini Project 2 - Bernoulli Naïve Bayes</h1>
<h4>The hyperparameters and models used in this file are chosen based on the findings in the testing file.</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 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

!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

"""Additional classifiers:
1. Logistic Regression
2. Multinomial Naïve Bayes
3. Support Vector Machine
4. Random Forest
5. Decision Tree
6. Ada Boost
7. k-Neighbors
8. Neural Network
"""

from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
```

```
from sklearn import svm
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier

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

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

"""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()
    vectors_train = vectorizer.fit_transform(X_train)
    vectors_test = vectorizer.transform(X_test)

    normalizer_train = Normalizer().fit(X=vectors_train)
    vectors_train = normalizer_train.transform(vectors_train)
    vectors_test = normalizer_train.transform(vectors_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):
    stop_words = text.ENGLISH_STOP_WORDS
    vectorizer = CountVectorizer(stop_words=stop_words)
    vectors_train_stop = vectorizer.fit_transform(X_train)
    vectors_test_stop = vectorizer.transform(X_test)

    normalizer_train = Normalizer().fit(X=vectors_train_stop)
    vectors_train_stop= normalizer_train.transform(vectors_train_stop)
    vectors_test_stop = normalizer_train.transform(vectors_test_stop)

    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, binary=False):
    stop_words = text.ENGLISH_STOP_WORDS
    tf_idf_vectorizer = TfidfVectorizer(binary=binary, stop_words=stop_words)
    vectors_train_idf = tf_idf_vectorizer.fit_transform(X_train)
    vectors_test_idf = tf_idf_vectorizer.transform(X_test)

    normalizer_train = Normalizer().fit(X=vectors_train_idf)
    vectors_train_idf= normalizer_train.transform(vectors_train_idf)
    vectors_test_idf = normalizer_train.transform(vectors_test_idf)

    return vectors_train_idf, vectors_test_idf

```

```
126 """4. CountVectorizer with stem tokenizer
127 1) Use "StemTokenizer" to transform text data to vector.
128 2) Normalize your feature vectors
129 """
130
131 class StemTokenizer:
132      def __init__(self):
133        self.wnl =PorterStemmer()
134      def __call__(self, doc):
135        return [self.wnl.stem(t) for t in word_tokenize(doc) if t.isalpha()]
136
137
138 def count_vec_stem(X_train, X_test):
139     vectorizer = CountVectorizer(tokenizer=StemTokenizer())
140     vectors_train_stem = vectorizer.fit_transform(X_train)
141     vectors_test_stem = vectorizer.transform(X_test)
142
143     normalizer_train = Normalizer().fit(X=vectors_train_stem)
144     vectors_train_stem= normalizer_train.transform(vectors_train_stem)
145     vectors_test_stem = normalizer_train.transform(vectors_test_stem)
146
147     return vectors_train_stem, vectors_test_stem
148
149 """5. CountVectorizer with lemma tokenizer
150 1) Use "LemmaTokenizer" to transform text data to vector.
151 2) Normalize your feature vectors
152 """
153
154 def get_wordnet_pos(word):
155     """Map POS tag to first character lemmatize() accepts"""
156     tag = nltk.pos_tag([word])[0][1][0].upper()
157     tag_dict = {"J": wordnet.ADJ,
158                 "N": wordnet.NOUN,
159                 "V": wordnet.VERB,
160                 "R": wordnet.ADV}
161     return tag_dict.get(tag, wordnet.NOUN)
162
163
164 class LemmaTokenizer:
165      def __init__(self):
166        self.wnl = WordNetLemmatizer()
167      def __call__(self, doc):
168        return [self.wnl.lemmatize(t,pos =get_wordnet_pos(t)) for t in
    word_tokenize(doc) if t.isalpha()]
169
170
171 def count_vec_lemma(X_train, X_test):
172     vectorizer = CountVectorizer(tokenizer=LemmaTokenizer())
173     vectors_train_lemma = vectorizer.fit_transform(X_train)
174     vectors_test_lemma = vectorizer.transform(X_test)
175
176     normalizer_train = Normalizer().fit(X=vectors_train_lemma)
177     vectors_train_lemma= normalizer_train.transform(vectors_train_lemma)
178     vectors_test_lemma = normalizer_train.transform(vectors_test_lemma)
179
180     return vectors_train_lemma, vectors_test_lemma
181
182 """## 9. Stacking classifier"""
183
184 from time import time
185
186 from sklearn.ensemble import StackingClassifier
187 tic = time()
```

```
accuracies = []
kf = KFold(n_splits=5, shuffle=True)
for train_index, test_index in kf.split(X):
    vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[test_index], binary=True)
    estimators = [
        ('mlp', MLPClassifier(max_iter=1000, learning_rate="adaptive", learning_rate_init=0.0001)),
        ('svc', svm.SVC(kernel='linear', gamma='auto', C=1, probability=True)),
        ('lr', LogisticRegression(C=40.0, max_iter=1000))
    ]
    clf = StackingClassifier(
        estimators=estimators, final_estimator=LogisticRegression()
    )
    clf.fit(vectors_train, y[train_index])
    y_test = clf.predict(vectors_test)
    accuracies.append(metrics.accuracy_score(y[test_index], y_test))
    print(accuracies[-1])
print("Average accuracy of Stacking Classification = {}".format(np.mean(accuracies)));
toc = time()
print("Time spent for Stacking Classification (with 5 fold validation) = {}".format(toc - tic))

"""## 10. Voting Classifier"""

from sklearn.ensemble import VotingClassifier
tic = time()
accuracies = []
kf = KFold(n_splits=5, shuffle=True)
for train_index, test_index in kf.split(X):
    vectors_train, vectors_test = tfidf_vectorizer(X[train_index], X[test_index], binary=True)
    clf1 = MLPClassifier(max_iter=1000, learning_rate="adaptive", learning_rate_init=0.0001)
    clf2 = LogisticRegression(C=40.0, max_iter=1000)
    clf3 = svm.SVC(kernel='linear', gamma='auto', C=1, probability=True)
    eclf = VotingClassifier(estimators=[('mlp', clf1), ('lr', clf2), ('svc', clf3)],
                            voting='soft', weights=[1,1,1])
    eclf = eclf.fit(vectors_train, y[train_index])
    y_test = eclf.predict(vectors_test)
    accuracies.append(metrics.accuracy_score(y[test_index], y_test))
    print(accuracies[-1])
print("Average accuracy of Volting Classification = {}".format(np.mean(accuracies)));
toc = time()
print("Time spent for Volting Classification (with 5 fold validation) = {}".format(toc - tic))
```

```
# -*- coding: utf-8 -*-
"""Export CSV.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1uNTFaEcHjLGUkGUojaKE8gCcieD9FaLQ

<center><h1>Mini Project 2 - Bernoulli Naïve Bayes</h1>
<h4>This file is for file exporting.</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 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

!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

"""Additional classifiers:
1. Logistic Regression
2. Multinomial Naïve Bayes
3. Support Vector Machine
4. Random Forest
5. Decision Tree
6. Ada Boost
7. k-Neighbors
8. Neural Network
"""

from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn import svm
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier

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)

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()
    vectors_train = vectorizer.fit_transform(X_train)
    vectors_test = vectorizer.transform(X_test)

    # z-score normalization
    normalizer_train = Normalizer().fit(X=vectors_train)
    vectors_train = normalizer_train.transform(vectors_train)
    vectors_test = normalizer_train.transform(vectors_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):
    stop_words = text.ENGLISH_STOP_WORDS
    vectorizer = CountVectorizer(stop_words=stop_words)
    vectors_train_stop = vectorizer.fit_transform(X_train)
    vectors_test_stop = vectorizer.transform(X_test)

    # z-score normalization
    normalizer_train = Normalizer().fit(X=vectors_train_stop)
    vectors_train_stop= normalizer_train.transform(vectors_train_stop)
    vectors_test_stop = normalizer_train.transform(vectors_test_stop)

    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()
    vectors_train_idf = tf_idf_vectorizer.fit_transform(X_train)
    vectors_test_idf = tf_idf_vectorizer.transform(X_test)

    # z-score normalization
    normalizer_train = Normalizer().fit(X=vectors_train_idf)
    vectors_train_idf= normalizer_train.transform(vectors_train_idf)
```

```
127     vectors_test_idf = normalizer_train.transform(vectors_test_idf)
128 
129     return vectors_train_idf, vectors_test_idf
130 
131 """4. CountVectorizer with stem tokenizer
132 1) Use "StemTokenizer" to transform text data to vector.
133 2) Normalize your feature vectors
134 """
135 
136 class StemTokenizer:
137      def __init__(self):
138        self.wnl =PorterStemmer()
139      def __call__(self, doc):
140        return [self.wnl.stem(t) for t in word_tokenize(doc) if t.isalpha()]
141 
142 
143 def count_vec_stem(X_train, X_test):
144     vectorizer = CountVectorizer(tokenizer=StemTokenizer())
145     vectors_train_stem = vectorizer.fit_transform(X_train)
146     vectors_test_stem = vectorizer.transform(X_test)
147 
148     # z-score normalization
149     normalizer_train = Normalizer().fit(X=vectors_train_stem)
150     vectors_train_stem= normalizer_train.transform(vectors_train_stem)
151     vectors_test_stem = normalizer_train.transform(vectors_test_stem)
152 
153     return vectors_train_stem, vectors_test_stem
154 
155 """5. CountVectorizer with lemma tokenizer
156 1) Use "LemmaTokenizer" to transform text data to vector.
157 2) Normalize your feature vectors
158 """
159 
160 def get_wordnet_pos(word):
161     """Map POS tag to first character lemmatize() accepts"""
162     tag = nltk.pos_tag([word])[0][1][0].upper()
163     tag_dict = {"J": wordnet.ADJ,
164                 "N": wordnet.NOUN,
165                 "V": wordnet.VERB,
166                 "R": wordnet.ADV}
167     return tag_dict.get(tag, wordnet.NOUN)
168 
169 
170 class LemmaTokenizer:
171      def __init__(self):
172        self.wnl = WordNetLemmatizer()
173      def __call__(self, doc):
174        return [self.wnl.lemmatize(t,pos =get_wordnet_pos(t)) for t in
    word_tokenize(doc) if t.isalpha()]
175 
176 
177 def count_vec_lemma(X_train, X_test):
178     vectorizer = CountVectorizer(tokenizer=LemmaTokenizer())
179     vectors_train_lemma = vectorizer.fit_transform(X_train)
180     vectors_test_lemma = vectorizer.transform(X_test)
181 
182     # z-score normalization
183     normalizer_train = Normalizer().fit(X=vectors_train_lemma)
184     vectors_train_lemma= normalizer_train.transform(vectors_train_lemma)
185     vectors_test_lemma = normalizer_train.transform(vectors_test_lemma)
186 
187     return vectors_train_lemma, vectors_test_lemma
188 
```

```
"""# Export csv"""

# test set id
X_id = reddit_test['id']
# test set features
X_test = reddit_test['body']

# vectorize the training and testing data
vectors_train, vectors_test = tfidf_vectorizer(X, X_test)
# perform MLP classification
clf = MLPClassifier(random_state=0, max_iter=1000, learning_rate="adaptive",
    learning_rate_init=0.0001).fit(vectors_train, y)

# predict the result
y_test = clf.predict(vectors_test)

# put the result into a pandas dataframe
result = {'id': X_id, 'subreddit': y_test}
df = pd.DataFrame(data=result)

# export to csv
df.to_csv('result.csv', index=False)

# download csv
from google.colab import files
files.download('result.csv')
```