Predicting Reddit Posts Categories Using Multiclass Naïve Bayes and Random Forest

Max Ardito maxwell.ardito@mail.mcgill.ca

Edwin Meriaux edwin.meriaux@mail.mcgill.ca

Ohood Sabr ohood.sabr@mail.mcgill.ca

Abstract

One of the most common machine learning challenges is text classification. In this project, a classifier is developed that can predict posts and comments from the site Reddit.com as belonging to one of four subreddits. A multinomial Naive Bayes classifier and a random forest are two independent algorithms that are employed and experimented with. The presented approach heavily utilizes feature construction to produce an ideal vocabulary for the task at hand. The multinomial Naive Bayes classifier was seen to produce the best results under 10-fold cross-validation. An additional test set was then run through this classifier on Kaggle resulting in a preliminary accuracy of 77.8%.

1 Introduction

The need to classify and arrange the increasing amount of electronic documents being produced around the world makes text classification one of the most important topics in the field of machine learning. Text classification models have been successfully used for a number of different scenarios, including topic detection, spam e-mail filtering, author identification, and so on [7]. The challenge presented in this paper is to classify unlabeled Reddit posts to their corresponding "Subreddits." Reddit is a social media platform and discussion forum based in the United States in which users can post text, images, links, and videos to one of many different subreddits—message boards that group posts around a central topic. The unlabeled Reddit posts in question each belong to one of four possible subreddits: JavaScript, MATLAB, PyTorch, and TensorFlow.

We have used two methods to solve this classification issue. The first method is a multinomial Naïve Bayes (MNB) classifier which assumes that the features are completely independent of each other. The second is a random forest method in which various decision trees are trained in parallel using common bagging and bootstrapping methods. According to the mean test accuracy in 10-fold cross-validation, the Naïve Bayes classifier produced the best results after both carefully preprocessing the compiled training set of Reddit posts into a vectorized vocabulary and choosing the optimal hyper-parameters. Laplace smoothing was used with the classifier to address the issue of zero probability. Additionally, a number of variations on the model's input vocabulary were evaluated over the course of our experimentation.

This report is organized as follows: Section 2 discusses the data set and talks about the preprocessing steps that were taken. Section 3 presents the various models experimented with during the training phase. Section 4 presents the findings, and Section 5 of the report concludes with some possible further research.

2 Datasets

2.1 Overview

The provided training and testing sets for evaluating our model were collected directly from the aforementioned subreddits. The training dataset has 720 samples which represent posts and comments that belong to the JavaScript, MATLAB, PyTorch, and TensorFlow subreddits. Likewise, the test set consists of 380 posts and comments. The classes are distributed equally in the training dataset. In order to construct the most relevant aspects of the input data, individual words will be treated as features. Furthermore, these words will undergo a number of preprocessing steps to filter out irrelevant features.

2.2 Features Extraction

Perhaps the most obvious issue with this classification task is the inherent crossover regarding vocabulary used in posts form these four subreddits. Since the four subreddits all deal with programming, care must be taken not only in constructing a dictionary of common stop words, but also in constructing a potential list of extended stop words that are common to all four programming languages, e.g. *function*, *package*, *variable*, etc. At the very least, the overlapping vocabulary must be weighted differently than vocabulary that is unique to one of the four languages, for instance *npm* for JavaScript, or *keras* for TensorFlow.

To first clear our vocabulary of common words, we considered a 318-word standard English stop-word dictionary from the Natural Language Toolkit Python module.[6] We then created our own list of additional English language stop words and merged it with the previous stop-word dictionary. Our next task was to remove all punctuation, capitalization, and spaces in order to obtain lists of lowercase words. This is a crucial step, as many of the posts in the training set consist of code blocks containing characters that are common to most programming languages such as curly braces , parenthesis *()*, and backslashes *//*. Finally, Lemmatization was applied on the training set to eliminate grammatical variations from the vocabulary by extract a common root from each term.

2.3 Feature Analysis

After cleaning both the training and testing dataset, a vectorization process is performed in order to obtain a final vocabulary of features for our model. SciKit-Learn's `CountVectorizer` was used to vectorize, lemmatize, and filter out the provided dictionary of stop-words. Thus, the output of this process is a vector with each dimension representing a word in the final word list. This vector can be used to analyze the frequency with which each word in the vector appears in a given input post. While training the MNB model (see Section 3.2), it was seen that an increase in accuracy was achieved by integrating 2-grams and 3-grams into the original feature space. This accuracy increase is obtained at the cost of efficiency, as the resulting size of the feature space must increase by at least a factor of 3.

The TF-IDF technique is frequently used to rate each word in a text document according to how unique it is. In other words, the TF-IDF technique captures the relevance between words, text documents, and certain categories [8] . It aims to assess a word's significance to a document inside a corpus (or collection) of documents by assigning a score or weight to each word. Along with the aforementioned preprocessing techniques, TF-IDF was implemented in our feature vector using the SciKit Learn's `TfidfTransformer`[1].

$$IDF(t, Corpus) = \log_2 \frac{(\#Docs\ in\ Corpus)}{(\#Docs\ with\ term\ t) + 1} \tag{1}$$

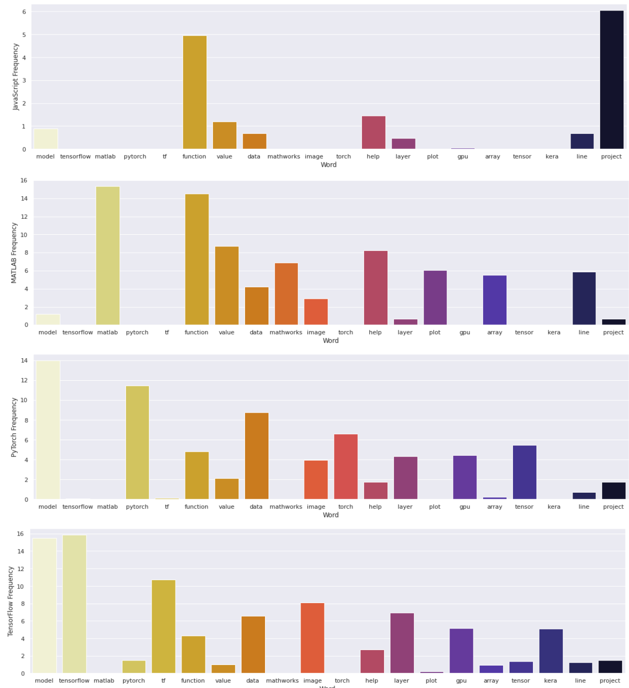

Figure 1: Histograms of representing the frequency of occurrence of each word from the post-processed vocabulary in training set's JavaScript, MATLAB, PyTorch, and TensorFlow posts respectively. Words are organized in descending order according to total number of occurrences in the input data.

Table 1: Top Features and Scores Using Chi-Squared Metric

tensorflow	matlab	pytorch	tf	mathworks	torch	js	keras	model	function
30.28	28.61	21.75	18.96	16.61	15.60	13.08	11.24	11.09	9.37

In Figure 1, the multiclass overlap between the 20 most frequently words is shown in the form of a histogram. In this study, the chi-squared statistic is employed as an analysis tool on our final feature space, as it is a suitable metric for calculating the relevance of certain words in the document corpus[5]. Additionally, Table 1 shows the most important features along with their scores.

3 Proposed Approach

After taking the necessary steps to preprocess and vectorize our feature space, an optimal model must be selected by means of experimentation and evaluation. A multinomial Naïve Bayes classifier was chosen as a base model due to its common use in text classification tasks [3]. Using 10-fold cross validation, the MNB model was evaluated multiple times to confirm and verify the success of the various preprocessing steps that were described in Section 2.3. Hyper-parameters such as N-gram ranges, maximum number of features, and more refined stop-word filtering were experimented with during this cross validation step.

An additional model was also tested which utilized a random forest approach, consisting of bootstrapping the training using N different decision trees and bagging their results. The performance of the random forest approach was compared with the MNB result by once again using 10-fold cross validation.

3.1 Multinomial Naive Bayes

The vectorized data obtained from Section 2 can be used to train a MNB model using a few different methods. For this task, we opted to implement a one-against-all approach, which treats a four-class classification problem as four separate binary classification problems. New posts are thus predicted by running them through each of the four classifiers and choosing the highest resulting sigma value.

The perks of this approach are that it results in conceptual simplicity and easily readable code. However this is at the expense of scalability, since four separate models must be trained. Because the provided training set is relatively small, the authors decided to go ahead with this approach.

In order to classify new data using this MNB model, we first must train the model to obtain parameters for each of the four classifiers. These include...

1. $\boldsymbol{\theta_n}$: The total number of training samples where y belongs to subreddit n over the total number of training samples
2. $\boldsymbol{\theta_{j,n}}$: The total number of training samples where word j in the vectorized vocabulary x exists in a post that also belongs to subreddit n over the total number of posts belonging to subreddit n
3. $\boldsymbol{\theta_{j,\neg n}}$: The total number of training samples where word j in the vectorized vocabulary x exists in a post that does not belong to subreddit n over the total number of posts that do not to subreddit n

Once these values are obtained, a new input post $\hat{x}$ can be vectorized and evaluated using the log-odds decision boundary formulated in (2). This equation is a slightly modified version of the equation found in [7].

$$a_n(\hat{x}) = \log\left(\frac{\theta_n}{1-\theta_n}\right) + \sum_{j=0}^{m}\left(x_j \log\frac{\theta_{j,n}}{\theta_{j,\neg n}} + (1-x_j)\log\frac{(1-\theta_{j,n})}{(1-\theta_{j,\neg n})}\right) \tag{2}$$

The resulting class for $\hat{x}$ can thus be found by evaluating each class' log-odds ratio and finding $\max_n\{\sigma(a_n(x))\}\forall n$, where σ is the sigmoid function.

3.2 Random Forest

A random forest base model was also experimented with on the dataset. The process of building a random forest model starts with constructing a set of B different decision trees that each contain m' features, where $m' < m$ and m is the total number of features in the feature space [2]. These decision trees are organized according to each node's information gain $IG(Y|X)$.

The random forest algorithm is different from a regular decision tree in that it bags multiple different decision trees together to produce a mean result. Bagging referrers to the process of using the same algorithm in n instances, which are each trained on bootstrapped fractions of the data set. The results of the n different instances are evaluated using either a majority vote for classification or a mean average for regression. This results in a model that combats overfitting at the expense of computationally complexity [4]. Bagging compares to stacking as a somewhat similar methods. Stacking is like bagging but with different algorithms rather than just differently trained versions of the same algorithms. These algorithms can be trained on different sets of the algorithm like in bagging.

4 Results

Between the random forest model and the MNB model, the classifier that scored the highest mean accuracy on 10-fold cross validation was the MNB model with the aforementioned vocabulary using lemmatization, TF-IDF, 2-grams, and 3-grams. The optimal number of features used in vectorization was seen to be the first 1000 using the `max_features` argument in SciKit Learn's

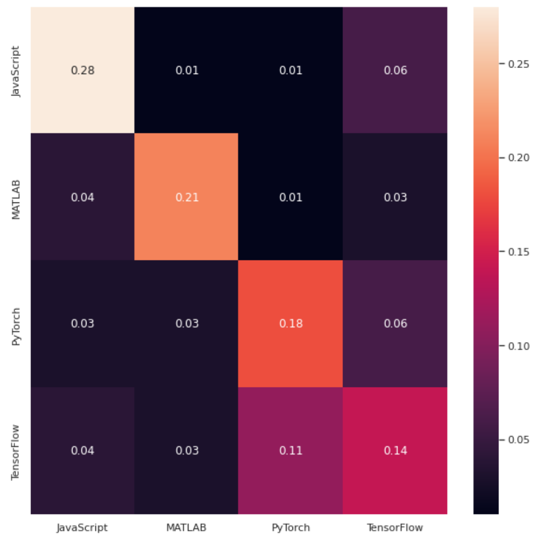

Figure 2: Multiclass confusion matrix showing the performance of the base MNB model in terms of the predicted classes on the Y-axis and the actual classes on the X-axis.

`CountVectorizer`. This model scored 77.18%. The random forest model occasionally performed better and occasionally performed worse, scoring a margin of 2% higher or lower than the MNB.

A subsequent improvement was attempted on the model pertaining to the differences between the PyTorch and TensorFlow vocabulary. These two subreddits contain vocabularies that are relatively similar to each other, since they both consist of Python-related topics and questions. This can be seen in Figure 2, which shows that the bulk of the model's misclassifications occurred in classifying a PyTorch post as a TensorFlow post and vice versa. In an attempt to resolve this issue, the terms *PyTorch*, *TensorFlow*, *torch*, *keras*, *tf*, and *tensor* were manually reweighted in our MNB model. Weights belonging to these elements of the feature space were increased by a factor of 100, resulting in an increased accuracy of 78.31% on the cross validation tests.

A separate testing set was also used on Kaggle.com to test the model when trained on the entirety of its training data. The model that scored the most accuracy on the Kaggle test set was the MNB model, which performed with little difference to the cross-validation tests and scored 77.8% accuracy, displaying its durability regarding variance.

5 Discussion and conclusion

In our final results it was found that the MNB model not only performed best but also held up well regarding overfitting. This can be seen in the similarity between the mean cross-validation score and the test set score. However, this overall accuracy of the model could certainly be improved. Given an even more attentive preprocessing, a larger dataset, and a slightly more complex model, it is possible that the accuracy could improve without sacrificing its low variance.

6 Statement of Contribution

1. Max Ardito: Naive Bayes base model, plots, feature analysis, Pytorch/TF reweighting, report.

2. Ohood Sabr: Feature construction, feature analysis, lemmatization, Chi Squared analysis, TF-IDF, report.

3. Edwin Meriaux: Feature construction, word and symbol filtering and separation, stop words bagging, stacking, random forest, report.

References

[1] Narges Armanfard. Ecse 551 - machine learning for engineers lecture 12 — decision trees (cont'd), feature construction, dimension reduction, September 2022.

[2] Narges Armanfard. Ecse 551 - machine learning for engineers lecture 17 — ensemble methods, September 2022.

[3] Narges Armanfard. Ecse 551 - machine learning for engineers lecture 9 — naive bayes, September 2022.

[4] IBM Cloud Education. Random forest.

[5] Dr. Saptarsi Goswami. Using the chi-squared test for feature selection with implementation, 2020.

[6] omdena.com/blog/machine-learning-classification algorithms. Text pre-processing: Stop words removal using different libraries, February.

[7] Alper Kursat Uysal and Serkan Gunal. The impact of preprocessing on text classification. *Information Processing Management*, 50(1):104–112, 2014.

[8] Zhang Yun-tao, Gong Ling, and Wang Yong-cheng. An improved tf-idf approach for text classification. *Journal of Zhejiang University-Science A*, 6(1):49–55, 2005.

7 Appendix A: Code

```
# -*- coding: utf-8 -*-
"""Reddit_NLP_Predictor_(Final).ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/16ZPRnjjbSfHpP472HiaP-xbSIN5JIm5a

# Subreddit Predictor

Implementation of a model that predicts whether input posts are from 1 of 4
↪  different programming subreddits. Uses a multiclass Bernoulli Naive Bayes
↪  model with stop words and bagging

## Install (Most) Dependencies
"""

# Installing natural language toolkit
!pip install nltk
import nltk
nltk.download('punkt')
nltk.download('wordnet')
nltk.download('averaged_perceptron_tagger')

#import list
import numpy as np
import random as rand
import pandas as pd
import seaborn as sns
import string
from scipy import sparse
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import AdaBoostClassifier
from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB, MultinomialNB
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.svm import LinearSVC
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction import text
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import Normalizer
from nltk.corpus import wordnet
from nltk import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.stem import PorterStemmer
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.feature_selection import SelectKBest, chi2, f_classif, mutual_info_classif, f_regression, mutual_info_regression, SelectPercentile
import math as ma
import scipy as sp
import matplotlib.pyplot as plt
import pandas as pd
import time
print("Finished importing!")

"""## Load The Dataset"""

import pandas as pd
import requests
from io import StringIO
import io
from google.colab import files

#imports train set from google drive
url='https://drive.google.com/file/d/1cqbryoylVGKr9LZ_5PhJFhvkbiw8sCH2/view?usp=sharing'

file_id = url.split('/')[-2]
dwn_url='https://drive.google.com/uc?export=download&id=' + file_id
url2 = requests.get(dwn_url).text
csv_raw = StringIO(url2)


df = pd.read_csv(csv_raw)
df.head()

#imports test set from computer local
uploaded = files.upload()
df2 = pd.read_csv(io.BytesIO(uploaded['test.csv']), encoding='cp1252')
print(df2.iloc[:,1].to_numpy())
test_set = df2.iloc[:,1].to_numpy()

"""## Custom Stop Words"""

#custom word list to improve accuracy to filter out specific words
#partially duplicated in the imported words list
switcher = {
  # All pronouns and associated words
  "i": True,
  "i'll": True,
  "i'd": True,
  "i'm": True,
  "i've": True,
  "ive": True,
  "me": True,
  "myself": True,
```

```
  "you": True,
  "you'll": True,
  "you'd": True,
  "you're": True,
  "you've": True,
  "yourself": True,
  "he": True,
  "he'll": True,
  "he'd": True,
  "he's": True,
  "him": True,
  "she": True,
  "she'll": True,
  "she'd": True,
  "she's": True,
  "her": True,
  "it": True,
  "it'll": True,
  "it'd": True,
  "it's": True,
  "itself": True,
  "oneself": True,
  "we": True,
  "we'll": True,
  "we'd": True,
  "we're": True,
  "we've": True,
  "us": True,
  "ourselves": True,
  "they": True,
  "they'll": True,
  "they'd": True,
  "they're": True,
  "they've": True,
  "them": True,
  "themselves": True,
  "everyone": True,
  "everyone's": True,
  "everybody": True,
  "everybody's": True,
  "someone": True,
  "someone's": True,
  "somebody": True,
  "somebody's": True,
  "nobody": True,
  "nobody's": True,
  "anyone": True,
  "anyone's": True,
  "everything": True,
  "everything's": True,
  "something": True,
  "something's": True,
  "nothing": True,
  "nothing's": True,
  "anything": True,
  "anything's": True,
  # All determiners and associated words
  "a": True,
  "an": True,
  "the": True,
  "this": True,
  "that": True,
  "that's": True,
  "these": True,
```

```
    "those": True,
    "my": True,
    "your": True,
    "yours": True,
    "his": True,
    "hers": True,
    "its": True,
    "our": True,
    "ours": True,
    "own": True,
    "their": True,
    "theirs": True,
    "few": True,
    "much": True,
    "many": True,
    "lot": True,
    "lots": True,
    "some": True,
    "any": True,
    "enough": True,
    "all": True,
    "both": True,
    "half": True,
    "either": True,
    "neither": True,
    "each": True,
    "every": True,
    "certain": True,
    "other": True,
    "another": True,
    "such": True,
    "several": True,
    "multiple": True,
    # "what": True,     #Dealt with later on
    "rather": True,
    "quite": True,
    # All prepositions
    "aboard": True,
    "about": True,
    "above": True,
    "across": True,
    "after": True,
    "against": True,
    "along": True,
    "amid": True,
    "amidst": True,
    "among": True,
    "amongst": True,
    "anti": True,
    "around": True,
    "as": True,
    "at": True,
    "away": True,
    "before": True,
    "behind": True,
    "below": True,
    "beneath": True,
    "beside": True,
    "besides": True,
    "between": True,
    "beyond": True,
    "but": True,
    "by": True,
    "concerning": True,
```

```
    "considering": True,
    "despite": True,
    "down": True,
    "during": True,
    "except": True,
    "excepting": True,
    "excluding": True,
    "far": True,
    "following": True,
    "for": True,
    "from": True,
    "here": True,
    "here's": True,
    "in": True,
    "inside": True,
    "into": True,
    "left": True,
    "like": True,
    "minus": True,
    "near": True,
    "of": True,
    "off": True,
    "on": True,
    "onto": True,
    "opposite": True,
    "out": True,
    "outside": True,
    "over": True,
    "past": True,
    "per": True,
    "plus": True,
    "regarding": True,
    "right": True,
    "since": True,
    "than": True,
    "there": True,
    "there's": True,
    "through": True,
    "to": True,
    "toward": True,
    "towards": True,
    "under": True,
    "underneath": True,
    "unlike": True,
    "until": True,
    "up": True,
    "upon": True,
    "versus": True,
    "via": True,
    "with": True,
    "within": True,
    "without": True,
    # Irrelevant verbs
    "may": True,
    "might": True,
    "will": True,
    "won't": True,
    "would": True,
    "wouldn't": True,
    "can": True,
    "can't": True,
    "cannot": True,
    "could": True,
    "couldn't": True,
```

```
    "should": True,
    "shouldn't": True,
    "must": True,
    "must've": True,
    "be": True,
    "being": True,
    "been": True,
    "am": True,
    "are": True,
    "aren't": True,
    "ain't": True,
    "is": True,
    "isn't": True,
    "was": True,
    "wasn't": True,
    "were": True,
    "weren't": True,
    "do": True,
    "doing": True,
    "don't": True,
    "does": True,
    "doesn't": True,
    "did": True,
    "didn't": True,
    "done": True,
    "have": True,
    "haven't": True,
    "having": True,
    "has": True,
    "hasn't": True,
    "had": True,
    "hadn't": True,
    "get": True,
    "getting": True,
    "gets": True,
    "got": True,
    "gotten": True,
    "go": True,
    "going": True,
    "gonna": True,
    "goes": True,
    "went": True,
    "gone": True,
    "make": True,
    "making": True,
    "makes": True,
    "made": True,
    "take": True,
    "taking": True,
    "takes": True,
    "took": True,
    "taken": True,
    "need": True,
    "needing": True,
    "needs": True,
    "needed": True,
    "use": True,
    "using": True,
    "uses": True,
    "used": True,
    "want": True,
    "wanna": True,
    "wanting": True,
    "wants": True,
```

```
    "let": True,
    "lets": True,
    "letting": True,
    "let's": True,
    "suppose": True,
    "supposing": True,
    "supposes": True,
    "supposed": True,
    "seem": True,
    "seeming": True,
    "seems": True,
    "seemed": True,
    "say": True,
    "saying": True,
    "says": True,
    "said": True,
    "know": True,
    "knowing": True,
    "knows": True,
    "knew": True,
    "known": True,
    "look": True,
    "looking": True,
    "looked": True,
    "think": True,
    "thinking": True,
    "thinks": True,
    "thought": True,
    "feel": True,
    "feels": True,
    "felt": True,
    "based": True,
    "put": True,
    "puts": True,
    "begin": True,
    "began": True,
    "begun": True,
    "begins": True,
    "wanted": True,
    "like": True,
    "feel": True,
    "believe": True,
    "understand": True,
    "shall": True,
    "regard": True,
    "regards": True,
    "regarding": True,
    # Question words and associated words
    "who": True,
    "who's": True,
    "who've": True,
    "who'd": True,
    "whoever": True,
    "whoever's": True,
    "whom": True,
    "whomever": True,
    "whomever's": True,
    "whose": True,
    "whosever": True,
    "whosever's": True,
    "when": True,
    "whenever": True,
    "which": True,
    "whichever": True,
```

```python
    "where": True,
    "where's": True,
    "where'd": True,
    "wherever": True,
    "why": True,
    "why's": True,
    "why'd": True,
    "whyever": True,
    "what": True,
    "what's": True,
    "whatever": True,
    "whence": True,
    "how": True,
    "how's": True,
    "how'd": True,
    "however": True,
    "whether": True,
    "whatsoever": True,
    # Connector words and irrelevant adverbs
    "and": True,
    "or": True,
    "not": True,
    "because": True,
    "also": True,
    "always": True,
    "never": True,
    "only": True,
    "really": True,
    "very": True,
    "greatly": True,
    "extremely": True,
    "somewhat": True,
    "no": True,
    "nope": True,
    "nah": True,
    "yes": True,
    "yep": True,
    "yeh": True,
    "yeah": True,
    "maybe": True,
    "perhaps": True,
    "more": True,
    "most": True,
    "less": True,
    "least": True,
    "good": True,
    "great": True,
    "well": True,
    "better": True,
    "best": True,
    "bad": True,
    "worse": True,
    "worst": True,
    "too": True,
    "thru": True,
    "though": True,
    "although": True,
    "yet": True,
    "already": True,
    "then": True,
    "even": True,
    "now": True,
    "sometimes": True,
    "still": True,
```

```
    "together": True,
    "altogether": True,
    "entirely": True,
    "fully": True,
    "entire": True,
    "whole": True,
    "completely": True,
    "utterly": True,
    "seemingly": True,
    "apparently": True,
    "clearly": True,
    "obviously": True,
    "actually": True,
    "actual": True,
    "usually": True,
    "usual": True,
    "literally": True,
    "honestly": True,
    "absolutely": True,
    "definitely": True,
    "generally": True,
    "totally": True,
    "finally": True,
    "basically": True,
    "essentially": True,
    "fundamentally": True,
    "automatically": True,
    "immediately": True,
    "necessarily": True,
    "primarily": True,
    "normally": True,
    "perfectly": True,
    "constantly": True,
    "particularly": True,
    "eventually": True,
    "hopefully": True,
    "mainly": True,
    "typically": True,
    "specifically": True,
    "differently": True,
    "appropriately": True,
    "plenty": True,
    "certainly": True,
    "unfortunately": True,
    "ultimately": True,
    "unlikely": True,
    "likely": True,
    "potentially": True,
    "fortunately": True,
    "personally": True,
    "directly": True,
    "indirectly": True,
    "nearly": True,
    "closely": True,
    "slightly": True,
    "probably": True,
    "possibly": True,
    "especially": True,
    "frequently": True,
    "thankfully": True,
    "often": True,
    "oftentimes": True,
    "seldom": True,
    "rarely": True,
```

```
    "sure": True,
    "while": True,
    "whilst": True,
    "able": True,
    "unable": True,
    "else": True,
    "ever": True,
    "once": True,
    "twice": True,
    "thrice": True,
    "almost": True,
    "again": True,
    "instead": True,
    "next": True,
    "previous": True,
    "unless": True,
    "somehow": True,
    "anyhow": True,
    "anywhere": True,
    "somewhere": True,
    "everywhere": True,
    "elsewhere": True,
    "anytime": True,
    "nowhere": True,
    "further": True,
    "anymore": True,
    "later": True,
    "ago": True,
    "ahead": True,
    "just": True,
    "same": True,
    "different": True,
    "big": True,
    "small": True,
    "little": True,
    "tiny": True,
    "large": True,
    "huge": True,
    "pretty": True,
    "mostly": True,
    "anyway": True,
    "anyways": True,
    "otherwise": True,
    "regardless": True,
    "needless": True,
    "throughout": True,
    "additionally": True,
    "moreover": True,
    "furthermore": True,
    "therefore": True,
    "thereof": True,
    "meanwhile": True,
    "likewise": True,
    "afterwards": True,
    "nice": True,
    "nicer": True,
    "nicest": True,
    "glad": True,
    "fine": True,
    # Irrelevant nouns
    "thing": True,
    "thing's": True,
    "things": True,
    "stuff": True,
```

```
    "other's": True,
    "others": True,
    "another's": True,
    "total": True,
    "true": True,
    "false": True,
    "none": True,
    "way": True,
    "kind": True,
    # Lettered numbers and order
    "zero": True,
    "zeros": True,
    "zeroes": True,
    "one": True,
    "ones": True,
    "two": True,
    "three": True,
    "four": True,
    "five": True,
    "six": True,
    "seven": True,
    "eight": True,
    "nine": True,
    "ten": True,
    "twenty": True,
    "thirty": True,
    "forty": True,
    "fifty": True,
    "sixty": True,
    "seventy": True,
    "eighty": True,
    "ninety": True,
    "hundred": True,
    "hundreds": True,
    "thousand": True,
    "thousands": True,
    "million": True,
    "millions": True,
    "first": True,
    "last": True,
    "second": True,
    "third": True,
    "fourth": True,
    "fifth": True,
    "sixth": True,
    "seventh": True,
    "eigth": True,
    "ninth": True,
    "tenth": True,
    "firstly": True,
    "secondly": True,
    "thirdly": True,
    "lastly": True,
    # Greetings and slang
    "hello": True,
    "hi": True,
    "hey": True,
    "sup": True,
    "yo": True,
    "greetings": True,
    "please": True,
    "okay": True,
    "ok": True,
    "y'all": True,
```

```
  "lol": True,
  "rofl": True,
  "thank": True,
  "thanks": True,
  "alright": True,
  "kinda": True,
  "dont": True,
  "sorry": True,
  "idk": True,
  "doesnt": True,
  "doesn": True,
  "didn": True,
  "didnt": True,
  "haven": True,
  "havent": True,
  "ugh": True,
    "guess": True,
  "bullshit": True,
  "yup": True,
  "yep": True,
  "haha": True,
  "hahaha": True,
  "hahahaha": True,
  "hehe": True,
  "hehehe": True,
  "till": True,
  "sure": True,
  "soon": True,
  "nah": True,
  "meh": True,
  "imo": True,
  "imho": True,
  "ill": True,
  "hella": True,
  "btw": True,
  "bro": True,

  # Miscellaneous
  "www": True,
  "https": True,
  "http": True,
  "com": True,
  "etc": True,
  "html": True,
  "reddit": True,
  "subreddit": True,
  "subreddits": True,
  "comments": True,
  "reply": True,
  "replies": True,
  "thread": True,
  "threads": True,
  "post": True,
  "posts": True,
  "website": True,
  "websites": True,
  "web site": True,
  "web sites": True
}

"""## Download Stop Words"""

#non custom stop words
#from prewritten imported words
```

```
import nltk
from gensim.parsing.preprocessing import remove_stopwords
from nltk.corpus import words
from nltk.corpus import stopwords
nltk.download('words')
nltk.download('stopwords')
"using" in words.words()
import nltk

print(stopwords.words('english'))
numpy_matrix = df.to_numpy()
print(numpy_matrix.shape)
print(numpy_matrix[1,1])
print("---")

"""## Parse Data

Processes punctuation, code syntax, upper/lowercase...
"""

#filtering out stop words and symbols


apostrophe = 0
y = []
X = []

#custom symbols to be removed
#of these symbols the only one to keep was <> because that is common mainly in
↪  Javascript so it was kept
stop_chars =
↪  [',','.','"','?','!',')','(','[',']','{','}','/','-',':','|','*','^','%','$','#','@','=','+','_','
↪  "\x94",'&']
keep_chars = ['<','>']
keep_no_space = ['\x92',"'"]
apostraphe_remove = 0

#some simple lemmetization is being done here aswell by removed everything
↪  between apostraphes and spaces
#everything is broken down and then reassembled around spaces after symbols
↪  are removed
#those words are then passed through the stop words lists
for val in range(numpy_matrix.shape[0]):
  current_phrase = numpy_matrix[val,0]
  spaced_list = ""

  for i in range(len(current_phrase)):
    letter = current_phrase[i]
    if letter.isdigit():
      pass
    elif letter in keep_no_space:
      apostrophe = 1
    elif apostrophe == 1 and letter == " ":
      apostrophe = 0
    elif apostrophe == 1:
      pass
    elif not letter.isalpha() and letter not in stop_chars and letter not in
    ↪  keep_no_space:
      spaced_list += " "
      spaced_list += letter
      spaced_list += " "
    elif not letter.isalpha() and letter in stop_chars:
      spaced_list += " "
    else:
```

```
      if letter.isalpha():
        spaced_list += letter.lower()
      else:
        spaced_list += letter

  word_list = []

  stop_words = set(stopwords.words('english'))
  word_tokens = word_tokenize(spaced_list)

  filtered_sentence = [w for w in word_tokens if not w.lower() in stop_words]

  filtered_sentence = []
  my_stop_words = text.ENGLISH_STOP_WORDS
  filtered = []
  filtered_string = ""
  for w in spaced_list.split():
    if w not in switcher and w not in stop_words and w not in my_stop_words
    ↪  and len(w) > 1:
      filtered.append(w)
      filtered_string += " "
      filtered_string += w
  #final phrases
  X.append(filtered_string)

#exact same being done just for the test set
y_test = []
x_test = []
apostrophe = 0
for val in range(test_set.shape[0]):
  current_phrase = test_set[val]
  spaced_list = ""
  #stop_chars =
  ↪  [',','.','"','?','!',')','(','[',']','{','}','/','-',':','|','*','^','%','$','#','@','=','+','_'
  ↪  "\x94",'&']
  #keep_chars = ['<','>']
  #keep_no_space = ['\x92',"'"]
  for i in range(len(current_phrase)):
    letter = current_phrase[i]
    if letter.isnumeric():
      pass
    elif letter in keep_no_space:
      apostrophe = 1
    elif apostrophe == 1 and letter == " ":
      apostrophe = 0
    elif apostrophe == 1:
      pass
    elif not letter.isalpha() and letter not in stop_chars and letter not in
    ↪  keep_no_space:
      spaced_list += " "
      spaced_list += letter
      spaced_list += " "
    elif not letter.isalpha() and letter in stop_chars:
      spaced_list += " "
    else:
      if letter.isalpha():
        spaced_list += letter.lower()
      else:
        spaced_list += letter

  word_list = []

  stop_words = set(stopwords.words('english'))
  word_tokens = word_tokenize(spaced_list)
```

```

  filtered_sentence = [w for w in word_tokens if not w.lower() in stop_words]

  filtered_sentence = []
  my_stop_words = text.ENGLISH_STOP_WORDS
  filtered = []
  filtered_string = ""
  for w in spaced_list.split():
    if w not in switcher and w not in stop_words and w not in my_stop_words and len(w) > 1:
      filtered.append(w)
      filtered_string += " "
      filtered_string += w

  x_test.append(filtered_string)
print(X)
print(len(X))

print(stop_words)

"""## Process muticlass y Vector

Converts the y text labels to class numbers from 0 - 3. Train test split is performed at the end
"""

import random

trainingSize = numpy_matrix[:, 0].size
y = np.zeros(trainingSize)

# Convert class labels from values 0 - 3
for i in range(trainingSize):
  if(numpy_matrix[i, 1] == 'Javascript'):
    y[i] = 0
  elif(numpy_matrix[i, 1] == 'Matlab'):
    y[i] = 1
  elif(numpy_matrix[i, 1] == 'Pytorch'):
    y[i] = 2
  elif(numpy_matrix[i, 1] == 'Tensorflow'):
    y[i] = 3

# Shuffle data and classes syncronously
# Code citation: https://www.geeksforgeeks.org/python-shuffle-two-lists-with-same-order/
tmp = list(zip(X, y))
random.shuffle(tmp)
tmp_X, tmp_y= zip(*tmp)
X, y = list(tmp_X), np.asarray(list(tmp_y))

"""# Evaluation

Here we plot histograms of each word, as well as the variance of each word's occurance per class
"""

import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import statistics
"""
    Class for evaluating the performance of a logistic regression model.
    Includes tools for calculating confusion matrix, accuracy, precision,
```

```
    recall, specificity, and false positive rate.

    Assumes that the input data is the result of a binary logistic regression
    (e.g. y && y_hat = {0, 1})
"""


class Evaluation:
    """
        Initializs the evaluation from a vector of predicted binary values (y)
        and a vector of actual values (y_hat). Stores these values in a
        confusion matrix variable (cm) as well as individual cell
        values (tp, tn, fp, fn)
    """

    def __init__(self,
                 X: np.ndarray,
                 y: np.ndarray,
                 word_list: int,
                 num_classes: int):
        self.X = X
        self.y = y
        self.word_list = word_list
        self.num_classes = num_classes
        self.df_trunc = None
        self.df_trunc_long = None

    """
        Prints a heatmap of the confusion matrix
    """

    def confusion_matrix(self, y_hat: np.ndarray):
        size = y_hat.size
        cm = np.ndarray(shape = (self.num_classes, self.num_classes))
        for i in range(size):
         x_index = y_hat[i].astype(int)
         y_index = y[i].astype(int)
         cm[x_index, y_index] += 1

        # Normalize confusion matrix
        cm = np.divide(cm, np.array(size))

        for i in range(self.num_classes):
          for j in range(self.num_classes):
            cm[i, j] = round(cm[i, j], 2)

        labels = ["JavaScript", "MATLAB", "PyTorch", "TensorFlow"]
        df = pd.DataFrame(cm, index = labels, columns = labels)
        fig = plt.figure(figsize = (10, 10))
        cell_labels = cm
        sns.heatmap(df, fmt='', annot=cell_labels)
        plt.savefig("Heatmap.png")
        return cm

    """
        Returns some histograms on word count
    """

    def tables(self, X, y):
        # In this data set, the number of disease patients is equal to the number of helthy patients
        kwargs = dict(line_kws = {'lw': 3})

        # Plot all distributions in the data set
```

```
        X_tmp_JavaScript = []
        X_tmp_MATLAB = []
        X_tmp_PyTorch = []
        X_tmp_TF = []
        X_ord = np.sum(X, axis = 0)

        for i in range(X[:, 0].size):
          if(y[i] == 0):
            X_tmp_JavaScript.append(X[i, :])
          elif(y[i] == 1):
            X_tmp_MATLAB.append(X[i, :])
          elif(y[i] == 2):
            X_tmp_PyTorch.append(X[i, :])
          elif(y[i] == 3):
            X_tmp_TF.append(X[i, :])

        X_tmp_JavaScript = np.sum(X_tmp_JavaScript, axis = 0)
        X_tmp_MATLAB = np.sum(X_tmp_MATLAB, axis = 0)
        X_tmp_PyTorch= np.sum(X_tmp_PyTorch, axis = 0)
        X_tmp_TF = np.sum(X_tmp_TF, axis = 0)

        variance = []
        for i in range(len(self.word_list)):
          values = [X_tmp_JavaScript[i], X_tmp_MATLAB[i], X_tmp_PyTorch[i], X_tmp_TF[i]]
          variance.append(statistics.variance(values))

        table = {'Word': self.word_list,
                 'JavaScript Frequency': X_tmp_JavaScript,
                 'MATLAB Frequency': X_tmp_MATLAB,
                 'PyTorch Frequency': X_tmp_PyTorch,
                 'TensorFlow Frequency': X_tmp_TF,
                 'Total Frequency': X_ord,
                 'Variance': variance}

        df = pd.DataFrame(data = table)

        self.df_trunc = df.iloc[0:20]

        self.df_trunc_long = df.iloc[0:100]


        return df

    def histograms(self):
        fig = plt.figure(figsize = (20, 5))
        plt.savefig("JS.png")

        sns.barplot(data = self.df_trunc,
               x = "Word",
               y = "JavaScript Frequency",
               palette = "CMRmap_r")

        fig = plt.figure(figsize = (20, 5))
        plt.savefig("MATLAB.png")

        sns.barplot(data = self.df_trunc,
               x = "Word",
               y = "MATLAB Frequency",
               palette = "CMRmap_r")

        fig = plt.figure(figsize = (20, 5))
        plt.savefig("PyTorch.png")

```

```
        sns.barplot(data = self.df_trunc,
               x = "Word",
               y = "PyTorch Frequency",
               palette = "CMRmap_r")

        fig = plt.figure(figsize = (20, 5))
        plt.savefig("TF.png")

        sns.barplot(data = self.df_trunc,
               x = "Word",
               y = "TensorFlow Frequency",
               palette = "CMRmap_r")

        self.df_trunc_long = self.df_trunc_long.sort_values(by = 'Variance',
        ↪  ascending = False)

        sns.barplot(data = self.df_trunc_long,
                    x = "Word",
                    y = "Variance")

"""# Naive Bayes Implementation"""

!pip install tqdm

import math
from tqdm import tqdm
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.preprocessing import Normalizer
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords

nltk.download('omw-1.4')

"""
  Implements a MultiClass Naive Bayes Classifier
"""
class NaiveBayes:

    """
      Initializes the model with

      num_classes - Number of classes used in the output data
      stop_words - A set containing stop words to be filtered out by the
↪   vectorizer
    """
    def __init__(self, num_classes, stop_words):
        self.num_classes = num_classes
        self.theta = np.zeros(self.num_classes)
        self.word_list= []
        self.stop_words = stop_words
        self.num_words= 0
        self.theta_j0 = None
        self.theta_j1 = None
        self.eval = None
        self.vocab = None

    """
      Static impelemntation of sigmoid used for predicting final output
    """
    @staticmethod
    def __sigmoid(x):
        return 1 / (1 + math.exp(-x))
```

```python

    @staticmethod
    def __getvocab(self, X, y):
      # First vectorize the input dataset to obtain a histogram of words
      vectorizer = CountVectorizer(max_features = 1200,
                                   binary = False,
                                   ngram_range = (1,3),
                                   stop_words = self.stop_words)

      tf_idf_transformer = TfidfTransformer(smooth_idf = True,
                                            use_idf = True)

      normalizer = Normalizer()

      for i in range(y.size):
          sentences = nltk.sent_tokenize(X[i])
          lemmatizer = WordNetLemmatizer()

          # Lemmatization
          for j in range(len(sentences)):
              words = nltk.word_tokenize(sentences[j])
              words = [lemmatizer.lemmatize(word) for word in words if word not in stop_words]

          sentences = ' '.join(words)

          X[i] = sentences

      vectorized_matrix = vectorizer.fit_transform(X)
      vectorized_idf = tf_idf_transformer.fit_transform(vectorized_matrix)
      normalized_idf = normalizer.fit_transform(vectorized_idf)
      X = normalized_idf.toarray()

      # Obtain the word list, number of words, and empty theta probabilities from
      # the vectorization
      self.word_list = list(vectorizer.get_feature_names_out())
      self.num_words = len(self.word_list)
      self.theta_j0 = np.zeros((self.num_words, self.num_classes))
      self.theta_j1 = np.zeros((self.num_words, self.num_classes))

      # Create Evaluation class for new dictionary based on variance
      e = Evaluation(X, y, self.word_list, 4)

      bayes_data_frame = e.tables(X, y)
      bayes_data_frame = bayes_data_frame.sort_values(by = 'Variance', ascending = False)

      variance_vocab = bayes_data_frame['Word'].iloc[0:130].astype(str)
      variance_vocab = variance_vocab.to_list()
      vocab = {k: v for v, k in enumerate(variance_vocab)}
      return vocab

    """
      Trains the Naive Bayes classifier using an input of strings and corresponding
      numeric class labels. The fit method contains an internal instance of sklearn's
      CountVectorizer, which stores the list of words based on the input data and
      the stop words.

      X: A list of strings containing each block of text in the dataset
      y: A numpy array of numeric class labels corresponding to the items in X
```

```python
    """
    def fit(self, X, y, plot_flag):

      vocab = self.__getvocab(self, X, y)

      print(vocab)

      # First vectorize the input dataset to obtain a histogram of words
      vectorizer = CountVectorizer(max_features = 800,
                                   binary = False,
                                   ngram_range = (1,3),
                                   vocabulary = vocab,
                                   stop_words = self.stop_words)

      tf_idf_transformer = TfidfTransformer(smooth_idf = True,
                                            use_idf = True)

      normalizer = Normalizer()

      for i in range(y.size):
          sentences = nltk.sent_tokenize(X[i])
          lemmatizer = WordNetLemmatizer()

          # Lemmatization
          for j in range(len(sentences)):
              words = nltk.word_tokenize(sentences[j])
              words = [lemmatizer.lemmatize(word) for word in words if word not in stop_words]

          sentences = ' '.join(words)

          X[i] = sentences

      vectorized_matrix = vectorizer.fit_transform(X)
      vectorized_idf = tf_idf_transformer.fit_transform(vectorized_matrix)

      # Obtain the word list, number of words, and empty theta probabilities from
      # the vectorization
      self.word_list = list(vectorizer.get_feature_names_out())
      self.num_words = len(self.word_list)

      normalized_idf = normalizer.fit_transform(vectorized_idf)
      X = normalized_idf.toarray()

      self.theta_j0 = np.zeros((self.num_words, self.num_classes))
      self.theta_j1 = np.zeros((self.num_words, self.num_classes))


      # Transform y into an n-dimensional array where n = num_classes, and
      # convert each dimension values into binary class values (one class against all others)
      y_multiclass = np.zeros((self.num_classes, y.size))
      for n in range(self.num_classes):
        for i in range(y.size):
          y_multiclass[n, i] = 1 if (y[i] == n) else 0
      specials = ["tf","tensorflow","pytorch","torch"]
      weightfactor = 100
      # Train each class' binary theta feature values
      for n in range(self.num_classes):
        self.theta[n] = y_multiclass[n, :].sum() / y.size
        for i in tqdm(range(self.num_words),
                      desc="Class: " + str(n + 1) + " / " + str(self.num_classes),
```

```
                        ascii=False,
                        ncols=75):
            for j in range(y.size):
              if(y_multiclass[n, j] == 1):
                  self.theta_j1[i, n] += X[j, i]
              else:
                  self.theta_j0[i, n] += X[j, i]

            # Dividing by total number of items of each class + Laplace
            ↪  Smoothing
            self.theta_j1[i, n] = (self.theta_j1[i, n] + 1) / (y_multiclass[n,
            ↪  :].sum() + 2)
            self.theta_j0[i, n] = (self.theta_j0[i, n] + 1) / ((1 -
            ↪  y_multiclass[n, :]).sum() + 2)

        if(plot_flag):
          eval = Evaluation(X, y, self.word_list, 4)
          eval.tables(X, y)
          eval.histograms()


    """
      Classify a new input x as one of the n classes. Predicted class is based
      on the values derived from the fit function above

      x - String containing an unlabeled block of text
    """
    def predict(self, x):
        bias = np.zeros(self.num_classes)
        xTw = np.zeros(self.num_classes)
        result = np.zeros(self.num_classes)

        # Calculate weights and bias for each class
        for n in range(self.num_classes):
          bias[n] = math.log(self.theta[n] / (1 - self.theta[n]))
          hist = np.vstack((self.theta_j0[:, n], self.theta_j1[:, n]))

          # Check each word in the class' word list and see if it
          # exists in the input block of text
          for i in range(self.num_words):
              currentWord = self.word_list[i]
              if(x.count(currentWord)):
                  xTw[n] += math.log(hist[1, i] / hist[0, i])
              else:
                  xTw[n] += math.log((1 - hist[1, i]) / (1 - hist[0, i]))

          # Pass the resulting bias an weights into the sigmoid function
          result[n] = self.__sigmoid(bias[n] + xTw[n])

        # Choose the max result
        result_max = np.where(result == max(result))[0][0]

        return result_max

import math
from tqdm import tqdm
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.preprocessing import Normalizer
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords

nltk.download('omw-1.4')
```

```python

"""
  Implements a MultiClass vectorizer
  This is effectively the same code as the naive bayes minus the actual naive bayes
  Just the data processing is included for the vectorization so that
"""
class simple_vectorizer():

    """
      Initializes the model with

      num_classes - Number of classes used in the output data
      stop_words - A set containing stop words to be filtered out by the vectorizer
    """
    def __init__(self, num_classes, stop_words):
        self.num_classes = num_classes
        self.theta = np.zeros(self.num_classes)
        self.word_list= []
        self.stop_words = stop_words
        self.num_words= 0
        self.theta_j0 = None
        self.theta_j1 = None
        self.eval = None
        self.vocab = None

    """
      Static impelemntation of sigmoid used for predicting final output
    """
    @staticmethod
    def __sigmoid(x):
        return 1 / (1 + math.exp(-x))


    """
      Trains the Naive Bayes classifier using an input of strings and corresponding
      numeric class labels. The fit method contains an internal instance of sklearn's
      CountVectorizer, which stores the list of words based on the input data and
      the stop words.

      X: A list of strings containing each block of text in the dataset
      y: A numpy array of numeric class labels corresponding to the items in X
    """
    def train_vector(self, X, y):
      # First vectorize the input dataset to obtain a histogram of words
      vectorizer = CountVectorizer(max_features = 800,
                                   binary = False,
                                   ngram_range = (1,3),
                                   stop_words = self.stop_words)

      tf_idf_transformer = TfidfTransformer(smooth_idf = True,
                                            use_idf = True)

      normalizer = Normalizer()

      for i in range(y.size):
          sentences = nltk.sent_tokenize(X[i])
          lemmatizer = WordNetLemmatizer()

          # Lemmatization
```

```
          for j in range(len(sentences)):
              words = nltk.word_tokenize(sentences[j])
              words = [lemmatizer.lemmatize(word) for word in words if word not in stop_words]

          sentences = ' '.join(words)

          X[i] = sentences

      vectorized_matrix = vectorizer.fit_transform(X)
      vectorized_idf = tf_idf_transformer.fit_transform(vectorized_matrix)
      normalized_idf = normalizer.fit_transform(vectorized_idf)
      X = normalized_idf.toarray()

      # Obtain the word list, number of words, and empty theta probabilities from
      # the vectorization
      self.word_list = list(vectorizer.get_feature_names_out())
      self.num_words = len(self.word_list)
      return X

    def test_vector(self, X):
      # First vectorize the input dataset to obtain a histogram of words
      vectorizer = CountVectorizer(max_features = 800,
                                   binary = False,
                                   ngram_range = (1,3),
                                   vocabulary = self.word_list,
                                   stop_words = self.stop_words)

      tf_idf_transformer = TfidfTransformer(smooth_idf = True,
                                            use_idf = True)

      normalizer = Normalizer()

      for i in range(len(X)):
          sentences = nltk.sent_tokenize(X[i])
          lemmatizer = WordNetLemmatizer()

          # Lemmatization
          for j in range(len(sentences)):
              words = nltk.word_tokenize(sentences[j])
              words = [lemmatizer.lemmatize(word) for word in words if word not in stop_words]

          sentences = ' '.join(words)

          X[i] = sentences

      vectorized_matrix = vectorizer.fit_transform(X)
      vectorized_idf = tf_idf_transformer.fit_transform(vectorized_matrix)
      normalized_idf = normalizer.fit_transform(vectorized_idf)
      X = normalized_idf.toarray()
      return X

"""#Train A K-Fold Classifier"""

import pandas as pd, numpy as np
from sklearn.preprocessing import StandardScaler
"""
        Class for performing K-fold cross validation and
        returning its mean error. Ideally used for comparing
        the performance of multiple logistic regression models.
"""
class KFold:
```

```
    """
        Initializes class by shuffling the input data and obtaining
        a validation set size based on the input dimensions and specified
        K value.

    """
    def __init__(self,
                 X: np.ndarray,
                 y: np.ndarray,
                 k: int,
                 stop_words: set):
        # Initialized data and classes
        self.X = X
        self.y = y

        self.k = k

        self.stop_words = stop_words

        # Validation size set = number of rows / k
        self.validation_set_size = int(len(self.X) / k)


    @staticmethod
    def __evaluate(model, X_val, Y_val):
        accuracy = 0
        total = 0
        #print(X_val)
        for i in tqdm(range(len(X_val)),
                              desc="Predicting Test Data: ",
                              ascii=False,
                              ncols=75):

          prediction = model.predict(X_val[i])
          if prediction == Y_val[i]:
              accuracy += 1
          total += 1

        accuracy_percent = (accuracy / total) * 100
        accuracy_percent = float(f'{accuracy_percent:.2f}')

        print("\n\n ================== \n\n")
        print("Accuracy: ", accuracy_percent, "%")

        return (accuracy / total)

    """
      Performs the cross validation on K iterations of the input data.
      The cross validation is performed by taking the first validation
      set from the top of the input data and then subsequently shifting
      (rolling) the input data N elements, where N = validation set size.

      Note that this method of cycling and partitioning will automatically
      throw any remainder of input data into the validation set if the data
      set cannot be evenly divided into K sets.
    """
    def cross_validation(self):
        # Initialize error count
        self.test_acc = 0

        for i in range(self.k):
            print("Fold - ", i + 1, " / ", self.k)

```

```
            # Training set = all rows where index is larger than validation set size
            X_train_fold = self.X[self.validation_set_size:]
            Y_train_fold = self.y[self.validation_set_size:]

            # Test set = all rows where index is below validation set size
            X_validation_fold = self.X[:self.validation_set_size]
            Y_validation_fold = self.y[:self.validation_set_size]

            # Implement the Naive Bayes model
            bayes = NaiveBayes(4, stop_words)

            bayes.fit(X_train_fold, Y_train_fold, 0)

            # Evaluate the predicted Y with the actual Y from the test data
            self.test_eval = self.__evaluate(bayes, X_validation_fold, Y_validation_fold)

            # Accumulate the error
            self.test_acc += self.test_eval

            # Shift the X data over a validation set size to ensure a new
            # validation set data for the next training iteration
            self.X = np.roll(self.X, -self.validation_set_size, axis = 0)
            self.y = np.roll(self.y, -self.validation_set_size, axis = 0)

        # Normalize error of all iterations ofself,  the validation set
        self.test_acc /= self.k

        print("--------------------------------------------")

        total_acc = self.test_acc * 100

        print("Total Accuracy: ",
              float(f'{total_acc:.2f}'), "%")

        return(total_acc)

import time
class KFold_bag:
    """
        Initializes class by shuffling the input data and obtaining
        a validation set size based on the input dimensions and specified
        K value.

    """
    def __init__(self,
                 X: np.ndarray,
                 y: np.ndarray,
                 k: int,
                 stop_words: set):
        # Initialized data and classes
        self.X = X
        self.y = y

        self.k = k

        self.stop_words = stop_words
        self.bags = []
        self.bag_size = 10

        # Validation size set = number of rows / k
        self.validation_set_size = int(len(self.X) / k)

```

```

    @staticmethod
    def __evaluate(model, X_val, Y_val):
        accuracy = 0
        total = 0
        #print(X_val)
        for i in tqdm(range(len(X_val)),
                              desc="Predicting Test Data: ",
                              ascii=False,
                              ncols=75):
          blanks = []

          for j in range(10):
            val = model[j].predict(X_val[i])
            blanks.append(val)
          print("blanks",blanks,Y_val[i])
          top = 0
          top_instance = 0
          for j in range(10):
            if top_instance > blanks.count(j):
              pass
            else:
              top_instance = blanks.count(j)
              top = j

          if top == Y_val[i]:
              accuracy += 1

          total += 1

        accuracy_percent = (accuracy / total) * 100
        accuracy_percent = float(f'{accuracy_percent:.2f}')

        print("\n\n ================== \n\n")
        print("Accuracy: ", accuracy_percent, "%")

        return (accuracy / total)

    """
      Performs the cross validation on K iterations of the input data.
      The cross validation is performed by taking the first validation
      set from the top of the input data and then subsequently shifting
      (rolling) the input data N elements, where N = validation set size.

      Note that this method of cycling and partitioning will automatically
      throw any remainder of input data into the validation set if the data
      set cannot be evenly divided into K sets.
    """
    def cross_validation(self):
        # Initialize error count
        self.test_acc = 0

        for i in range(self.k):
            print("Fold - ", i + 1, " / ", self.k)

            # Training set = all rows where index is larger than validation set size
            X_train_fold = self.X[self.validation_set_size:]
            Y_train_fold = self.y[self.validation_set_size:]

            # Test set = all rows where index is below validation set size
            X_validation_fold = self.X[:self.validation_set_size]
            Y_validation_fold = self.y[:self.validation_set_size]
            #print(X_validation_fold)
```

```python
            for j in range(self.bag_size):

              #percentage = random.uniform(0.75, 0.97)
              percentage = 0.99
              percentage = int(percentage*100)/100
              #print(i,percentage)
              bag_X_train, bag_X_test, bag_y_train, bag_y_test = train_test_split(X_validation_fold, Y_validation_fold, test_size=1-percentage, random_state=30)
              #print(bag_X_train)
              #print(bag_X_train.shape)
              #print(bag_y_train.shape)
              bag_bayes = NaiveBayes(4, stop_words)
              bag_bayes.fit(bag_X_train, bag_y_train, 0)
              self.bags.append(bag_bayes)
            #time.sleep(10)
            #print(self.bags)
            # Evaluate the predicted Y with the actual Y from the test data
            self.test_eval = self.__evaluate(self.bags, X_validation_fold, Y_validation_fold)

            # Accumulate the error
            self.test_acc += self.test_eval

            # Shift the X data over a validation set size to ensure a new
            # validation set data for the next training iteration
            self.X = np.roll(self.X, -self.validation_set_size, axis = 0)
            self.y = np.roll(self.y, -self.validation_set_size, axis = 0)

        # Normalize error of all iterations ofself,  the validation set
        self.test_acc /= self.k

        print("--------------------------------------------")

        total_acc = self.test_acc * 100

        print("Total Accuracy: ",
              float(f'{total_acc:.2f}'), "%")

        return(total_acc)

"""# Test A K-Fold Classifier

This section of the code just takes the kfold with naive bayes to test different presets to determine accuracy.
"""

import nltk
nltk.download('wordnet')
kf = KFold(X, y, 10, stop_words)
kf.cross_validation()

print(X)

bayes = NaiveBayes(4, stop_words)
bayes.fit(X, y, 1)

y_pred = []
for i in x_test:
  y_pred.append(bayes.predict(i))

output = []

step = 0
```

```python
for i in y_pred:
    if i == 0:
        output.append([step+1,"Javascript"])
    elif i == 1:
        output.append([step+1,"Matlab"])
    elif i == 2:
        output.append([step+1,"Pytorch"])
    elif i == 3:
        output.append([step+1,"Tensorflow"])
    step += 1
print(output)
df = pd.DataFrame(output)
df.to_csv('output.csv', index=False)

print(x_test)

"""#Other ML test

Vectorization for train and test sets
"""

bayes = simple_vectorizer(4, stop_words)
train_vector = bayes.train_vector(X, y)
test_vector = bayes.test_vector(x_test)

print(train_vector)
print(len(train_vector))
print(len(train_vector[0]))

print(test_vector)

"""Random Forest Test"""

from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score
rf_X_train, rf_X_test, rf_y_train, rf_y_test = train_test_split(train_vector,
↪  y, test_size=0.2)
clf = RandomForestClassifier(max_depth=20, random_state=0)
clf.fit(rf_X_train, rf_y_train)
y_pred = clf.predict(rf_X_test)
print(y_pred)
print(rf_y_test)
print(accuracy_score(rf_y_test, y_pred))

"""Support Vector Machine"""

from sklearn import svm
rf_X_train, rf_X_test, rf_y_train, rf_y_test = train_test_split(train_vector,
↪  y, test_size=0.2)
clf = svm.SVC()
clf.fit(rf_X_train, rf_y_train)
y_pred = clf.predict(rf_X_test)
print(y_pred)
print(rf_y_test)
print(accuracy_score(rf_y_test, y_pred))

"""Adaboost"""

from sklearn.ensemble import AdaBoostClassifier
rf_X_train, rf_X_test, rf_y_train, rf_y_test = train_test_split(train_vector,
↪  y, test_size=0.2)
clf = AdaBoostClassifier(n_estimators=200, random_state=0)
clf.fit(rf_X_train, rf_y_train)
```

```
y_pred = clf.predict(rf_X_test)
print(y_pred)
print(rf_y_test)
print(accuracy_score(rf_y_test, y_pred))

"""Gaussian Naive Bayes"""

from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
rf_X_train, rf_X_test, rf_y_train, rf_y_test = train_test_split(train_vector, y, test_size=0.25)
clf = GaussianNB()
clf.fit(rf_X_train, rf_y_train)
y_pred = clf.predict(rf_X_test)
print(y_pred)
print(rf_y_test)
print(accuracy_score(rf_y_test, y_pred))
print(confusion_matrix(rf_y_test, y_pred))

"""#Stacking

tested with:
Random Forest,

Support Vector Machine,

Adaboost
"""

rf_X_train, rf_X_test, rf_y_train, rf_y_test = train_test_split(train_vector, y, test_size=0.2, random_state=42)

stack = []

rf = RandomForestClassifier(max_depth=20, random_state=0)
rf.fit(rf_X_train, rf_y_train)
rf_y_pred = rf.predict(rf_X_test)
stack.append(rf_y_pred)

ada = AdaBoostClassifier(n_estimators=200, random_state=0)
ada.fit(rf_X_train, rf_y_train)
y_pred = ada.predict(rf_X_test)
#stack.append(y_pred)

sv = svm.SVC()
sv.fit(rf_X_train, rf_y_train)
y_pred = sv.predict(rf_X_test)
#stack.append(y_pred)

pred = []
for i in range(len(y_pred)):
  top = 0
  current = 0
  current_list = []
  for j in range(len(stack)):
    current_list.append(stack[j][i])
  for j in range(len(stack)):
    if top < current_list.count(j):
      top = current_list.count(j)
      current = j

  pred.append(current)

```

```
print(pred)

print(accuracy_score(rf_y_test, pred))

print(accuracy_score(rf_y_test, rf_y_pred))

```