ECSE 551 Group 16 Mini-Project 2

Jonathan Arsenault
260585289

Hung-Yang Chang
260899468

Anan Lu
260467054

Abstract

Text classification is a well-known machine learning problem. In this project, a classifier is built that identifies the subreddit from which a comment originated. Two separate algorithms are used, a multinomial Naïve Bayes classifier and a support-vector machine classifier. To obtain optimal performance from these classifiers, the raw text comments must be converted to features. The majority of this report presents the methods used to construct the features which yielded the best results. Through 5-fold cross-validation, the best results were found using the multinomial Naïve Bayes classifier with a tuned Laplace smoothing parameter, and assuming a uniform distribution of each class. Results were further improved by selecting features which were determined to be the most relevant using a chi-squared statistical test. This classifier was then applied to a test set, yielding a preliminary accuracy of 93.1% according to Kaggle.

1 Introduction

Text classification has many applications from email filtering to algorithmic social media recommendations. The problem posed in this project consists of classifying Reddit comments according to the subreddit in which they were posted. Reddit is an American social platform and discussion website. Members submit content such as text posts, images, and links to subreddits which organize all related discussion themes. Two methods must be used to solve this classification problem. The first method to be implemented is a multinomial Naïve Bayes (MNB) classifier, a classification algorithm that assumes the features are completely independent of each other. The second is a one-versus-all support vector machine (SVM) classifier, which uses a hyperplane to divide the feature space into two classes. The distance of each sample to the hyperplane is used to make a prediction.

After carefully selecting model parameters and features, the MNB classifier provided the best results, as defined by the mean test accuracy in 5-fold cross-validation. The key to achieving optimal performance was to assume a uniform prior on the class distributions, selecting only the most relevant features, and properly implementing Laplace smoothing. This classifier provided an accuracy of 93% when applied to the reserved testing data.

The remainder of this report is structured as follows. In Section 2, the data set is explored and any steps taken to preproceess the data are discussed. The decisions made when designing the classifiers are shown in Section 3. Results are presented in Section 4, and the report is brought to a close with a discussion and conclusion in Section 5.

2 Data

Two data sets are provided, a training set and a testing set. The training set consists of 11382 samples with the comment and corresponding subreddit. The distribution of each class in the training set is uneven, ranging from 20.6% for the `datascience` class to 3.7% for the `computers` class. On average, the comments contain 104 words, with a maximum of 552 words and a minimum of 1 word. The testing set contains 2860 unlabelled sample comments, and the number of words in each comment follows a similar distribution as the training set.

2.1 Feature Construction From Raw Text

Text classification is only possible if meaningful numerical features are extracted from the raw text data. Commonly used Natural Language Processing (NLP) techniques are described here. This is not an exhaustive list, but summarizes the more common approaches, and importantly the approaches used in this report.

2.1.1 Vectorization

In NLP, a token is any sequence of characters separated by a separator, typically defined as white space or punctuation. *Vectorization* is a method of converting a document into a meaningful vector of numbers. The standard way of doing this is to use a bag of words approach, in which a corpus is represented as the set of all the words contained therein. This method cannot capture the position of the token in the text, co-occurrences in different documents, and it also fails to capture semantics and grammar compared to topic models such as word embeddings [1]. However, it is easy to compute, and the similarities between multiple documents can be captured rapidly.

The simplest form of vectorization is count vectorization, in which the number of times each token appears in the document is recorded and converted to a vector. A binary variant is also sometimes applicable, where a token is assigned a value of 1 if it is present in the document and 0 otherwise.

However, one issue with count vectorization is that some words may appear many times without providing any meaningful information. *Term frequency-inverse document frequency* (TF-IDF), on the other hand, provides term weighting in an attempt to filter out terms that commonly occur in the corpus. Term frequency is the count of the word divided by the length of the document. For a given term t, the inverse document frequency is

$$IDF(t, corpus) = \log \frac{\#\text{ documents in corpus}}{\#\text{ documents with term } t}, \tag{1}$$

A feature with a high TF occurs frequently within a document, while a term with a high IDF appears infrequently in the entire corpus. TF-IDF is then the product of TF and IDF. It captures not only word frequency but also relative importance between words.

Another way of improving the performance of vectorized features is normalization. Here, row normalization is used, which scales each feature vector to have unit norm. Implementing normalization is common in machine learning [2][Chapter 3], and it can greatly improve the results, as shown in section 4.

A simple vectorization of the training set resulted in 41,033 features. As mentioned, standard vectorization simply creates a token based on the presence of white space or punctuation. However, more advanced processing is required to create meaningful features that may improve the results of the classification algorithm. Several text processing techniques were tested including removing stop words, lemmatization/stemming, and removing non-alphabetic characters.

2.1.2 Stop Words

Stop words are usually defined as common words that appear frequently in all documents. A standard English stop-word dictionary from the Natural Language Toolkit python library [3] that contains 318 words was first considered. These words include “the", “and", and “it", for example. Words could potentially be added to this stop word dictionary, so word clouds were created for the training and testing sets. These word clouds helped expand the dictionary of stop words. Two additional stop word dictionaries were created, a broader stop word dictionary with 347 words and a narrower one with 342 words. The choice of stop word dictionary was treated as a hyperparameter, for which the results are presented in Section 4.

2.1.3 Lemmatization and Stemming

Both lemmatization and stemming function as methods to reduce feature numbers by grouping together the different inflected forms of a word. Lemmatization is a more accurate method, as it keeps the readable form of the root word. It does so by factoring in the context of the word. Stemming does not factor in the context, and simply attempts to cut down a word to its root. The two methods would remove slightly different numbers of words depending on specific texts. Another factor that could impact prediction accuracy is redundant strings like non-alphabetic characters. Most of these characters may seem to provide little predictive power, but considering the nature of the categories analyzed here, some of the numbers and symbols could actually aid in realizing the correct prediction. Although lemmatizing/stemming and the removal of non-alphabetic characters could reduce feature numbers by almost 50 %, it sacrifices a substantial amount of computational cost with questionable improvement in accuracy [4]. The results of preliminary tests with these methods will be discussed in section 4.

2.1.4 N-gram

N-gram is a commonly used NLP technique to include the sequence of words. It generates features of a continuous sequence of n items from the given texts. Adding these features could potentially improve the accuracy of prediction; however, the computational cost would grow significantly due to the considerable increase in feature size. In order to compare its effectiveness, including bi-gram, tri-gram, and four-gram features was considered and reported in section 4.1.1

2.2 Feature selection

Performing text classification on a large corpus of text often yields an enormous amount of features. However, many of these features often offer little to no predictive value. Removing these specific features may even improve accuracy, with the added benefit of reducing the computational cost of fitting the model. Determining the *importance* of a feature can be done through a variety of metrics, including information gain, odds ratio, chi-squared statistic, linear regression test, etc.

For the purposes of this report, the chi-squared statistic is used, as it is easy to compute and does not significantly change the time required to fit the models [2]. The preferred method, mutual information gain, was abandoned due to lengthy computation times. The top 10 words with their corresponding chi-squared statistic are summarized in Table 1.This list seems correct based on the knowledge of the classes. For example, the “anime" should be important considering one of the classes is anime. The number of words to be chosen, denoted k is a hyperparameter. The effect of k on accuracy and computational cost will be discussed in Section 4.

Word	anime	data	car	game	gamedev	science	cars	players	unity	nvidia
Statistic	1149	849	559	437	331	302	287	228	212	211

Table 1: Ten most important words and their corresponding chi-squared statistic.

3 Proposed Approach

Having now discussed the different techniques which could be used to extract and select features, the models used to perform the classifications are now discussed, beginning with MNB and closing with the SVM.

3.1 Multinomial Naïve Bayes

MNB classification is a popular generative learning method that makes strong independence assumptions to significantly simplify the model. Namely, the naïve assumption is to assume that the features $x_j,\ j = 1, \ldots, m$ are conditionally independent given a class label y. This leads to one parameter per class per feature to be trained. Training is done using a simple counting method.

Laplace, or additive, smoothing is used to avoid having features with zero probability for some classes. Define the parameter $\theta_{j,k} = P(x_j = 1|y = k)$. In MNB classification, the maximum likelihood estimate of $\theta_{j,k}$ is

$$\theta_{j,k} = \frac{N_{j,k}}{N_k}, \tag{2}$$

where $N_{j,k}$ is the number of samples in the training set with $x_j = 1$ and $y = k$ and N_k is the number of samples in the training set with $y = k$. However, as each vectorized sample is normalized here, this is method is no longer valid. Instead,

$$\theta_{j,k} = \frac{S_{j,k} + \alpha}{\sum_{j=1}^{m} S_{j,k} + N\alpha}, \tag{3}$$

where α is a smoothing parameter and $S_{j,k} = \sum_{i=1}^{N} x_{j,k}^{(i)}$ is the sum over all training samples of x_j with $y = k$ [6]. The selection of the α parameter is elucidated in Section 4.

An important parameter in MNB classification is the estimation of the prior class distribution. The most common method is to observe the distribution of classes in the training set, and assume this distribution will hold in the testing set. Therefore, for the class k, $P(y = k) = N_k/N$. However, it is possible that the class distribution does not hold. In such a case, it may be advantageous to assume a uniform prior for the distribution of class, mean $P(y = k) = 1/K$, where K is the number of classes [6]. Both these approaches are tested, with the results shown in Section 4.

3.2 Support-Vector Machines

The MNB classifier presented above is to be compared to another popular supervised learning method, namely an SVM. An SVM was chosen because it scales well to higher dimensional data compared to other classification techniques [7].

The goal of an SVM is to build a hyperplane that best divides the feature space into two sub-spaces. One subspace contains vectors that belong to one class, and the other subspace belongs to all other classes. This scheme, known as "one-versus-all" is chosen as it is the implementation used in `sklearn` [8]. Moreover, since it is reasonable to assume training data is too complicated to be linearly separable, a radial basis function (RBF) kernel is chosen to separate data into eight classes. To obtain good performace, two important parameters, the regularization parameter C and the RBF scaling parameter γ, must be tuned. More precisely, the RBF kernal is given by $K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)$ where γ adjusts the curvature of the decision boundary. The larger the γ, the narrower the Gaussian bell is. Therefore, large γ leads to high bias and low variance models. The grid search to determine these parameters will be discussed in section 4.1.2.

4 Results

It is impossible to present every experiment performed that lead to these results. Only the most relevant ones will be presented and discussed in this section. The classifiers were evaluated based on their mean accuracy and fit time in 5-fold cross validation and fit time.

4.1 Choosing Model Parameters and Preprocessing Methods

4.1.1 Multinomial Naïve Bayes

To begin, tests were performed to select α and the prior distribution. Features were created using a simple count vectorization. TF-IDF vectorization was tried, but did not perform better. It was determined that $\alpha = 0.01$ and a uniform prior distribution yielded the best results. Further discussion of the choice of α can be found in Section 4.2. This baseline model had a mean 5-fold cross-validation accuracy of 89.45%. Fitting all folds took 120 seconds. Adding more complex feature processing was then tested, for which the results are shown in Table 2. For example, simply adding normalization increased the accuracy to 90.56%, while decreasing the fit time. This notable increase led to the decision to include normalization for all other methods tested here. The method lemma refers to lemmatization and *isalpha* refers to the removal of non-alphabetic characters. For testing n-grams, a max feature number of 40,000 was used due to limit on computational power.

It was noticed that removing stop-words is the only method that improves accuracy over a simple normalization. Adding n-grams led to decreasing accuracy with increasing computational cost. However, this could be due to limiting the number of features. Thus, further tests are necessary to see if chi-squared feature selection could improve the accuracy. Therefore, the impact of selected feature numbers k is also discussed in section 4.2. while a massive grid search was constructed to select the best parameters, which will be presented in section 4.3.

Method	Normalize	Stop Words	Lemma	*isalpha*	2-gram	3-gram	4-gram
Accuracy	90.56 %	90.66 %	90.58 %	89.86 %	88.84 %	87.26 %	86.54 %
Time	101 s	101 s	255 s	242 s	127 s	154 s	166 s

Table 2: Comparison of parameters and pre-processing methods for Multinomial Naïve Bayes

4.1.2 Support Vector Machine

Contrary to the MNB classifier, the SVM classifer performed better when using a TF-IDF vectorization. Normalization was also consistently used, as it also yielded better results. Lastly, selecting only the k best features did not improve the results, and this method was omitted. A grid search was performed to determine the best stop words, n-gram range, C, and γ. The final value of C and γ are chosen to be 10 and 0.1 respectively, along with using the expanded stop words dictionary and including 2-grams in the feature list. This resulted in mean 5-fold cross-validation accuracy of 89.8 % with a 400 second execution time. The submitted Kaggle competition accuracy is 91.7%, lower than that of the MNB classifier to be presented in Section 4.3.

4.2 Isolating the Impact of α and k

Two hyperparameters of the MNB classifier, α and k, merit further discussion. Separate grid searches are performed for each parameter, with the other parameters set to the best values found in Section 4.1. Figure 1 shows the mean test score and fit time for 4 different values of α. Simply using add-one smoothing, corresponding to $\alpha = 1$, is clearly the worst-performing option. This confirms that the method of Laplace smoothing implemented here is the best option, with optimal results found at $\alpha = 0.01$. Interestingly, larger values of α provide notably faster fit times than $\alpha = 10^{-3}$. Similarly, Figure 2 shows the mean test score and fit time for 7 different values of k. The general trend sees the mean test score increasing with k. This result is expected, as increasing the amount of features used will typically improve the results. However, the gains become marginal for $k > 15000$, and the increased training time seems to offset gains made in performance.

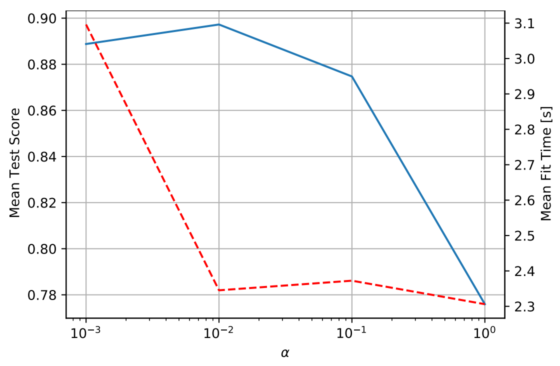

Figure 1: Mean test score as a function of Laplace smoothing parameter α.

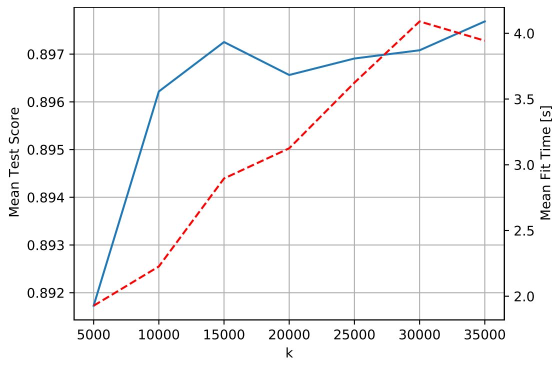

Figure 2: Mean test score as a function of number of features chosen.

4.3 Final Best Result

A final grid search was performed to determine the best combination of parameters for the MNB classifier. The search space included various stop-word dictionaries, n-gram ranges, α, k, and class probability prior. The best result used the broadest stop-word dictionary, $k = 15000$, $\alpha = 0.01$, and using a uniform prior on the class probability. Higher k may have yielded better performance, but to improve fit time, a smaller k was chosen with little impact on the result. This combination yields a preliminary accuracy of 93.1% according to Kaggle.

5 Discussion and Conclusion

In this report, supervised text classification is done using both a multinomial naïve Bayes classifier and a SVM classifier. Through careful testing of various model hyperparameters and feature selection methods, the Naïve Bayes classifier was deemed to offer better performance. This was especially true once Laplace smoothing was implemented and a uniform prior was placed on the class probabilities. Several feature selection methods, such as TF-IDF vectorization, lemmatization, and stemming, which in theory should improve performance, did not in fact yield better results.

Future investigation could be centered around different methods of selecting the features to use. One promising method is to use maximum information gain to determine importance instead of the chi-squared statistic. Using a combination of count vectorization and TF-IDF vectorization also seemed to yield promising results, but time constraints prevented further investigation.

6 Contribution Statement

1. Jonathan Arsenault: Data pre-processing, implementing of algorithm, write-up contribution.
2. Hung-Yang Chang: Data pre-processing, implementing of algorithm, write-up contribution.
3. Anan Lu: Data pre-processing, implementing of algorithm, write-up contribution.

References

[1] E. Rudkowsky, M. Haselmayer, M. Wastian, M. Jenny, S. Emrich & Michael Sedlmair "More than bags of words: Sentiment analysis with word embeddings." (2018) Communication Methods and Measures 12.2-3, 140-157.

[2] Ozdemir, S., & Susarla, D. (2018). Feature Engineering Made Easy. Packt Publishing.

[3] Scikit-Learn, "sklearn.feature_extraction.text.english_stop_words," [Online]. Available: https://scikit-learn.org/stable/modules/feature_extraction.html [Accessed: 06-Nov-2020].

[4] M. Toman, T. Roman, and J. Karel. "Influence of word normalization on text classification." (2006) Proceedings of InSciT 4: 354-358.

[5] G. Forman. "An extensive empirical study of feature selection metrics for text classification." (2003) Journal of machine learning research 3.Mar: 1289-1305.

[6] Scikit-Learn, "Naive Bayes," [Online]. Available: https://scikit-learn.org/stable/modules/naive_bayes.html [Accessed: 06-Nov-2020].

[7] Li, I-Jing. Wu, Jiunn-Lin & Yeh, Chih-Hung. . "A fast classification strategy for SVM on the large-scale high-dimensional datasets. Pattern Analysis and Applications." (2017) Pattern Analysis and Applications 21 4

[8] Scikit-Learn, "Support Vector Machines," [Online]. Available: https://scikit-learn.org/stable/modules/svm.html. [Accessed: 06-Nov-2020].

7 Appendix

Code1:miniproject2.py

```
# -*- coding: utf-8 -*-
"""miniproject2.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1KqDF3_FSXwR9Sz1fuZQzj7fxBbrNZhgA

# Mini-Project 2
"""

# Import relevant modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.preprocessing import Normalizer
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction import text
from sklearn import model_selection
from sklearn import svm
import time
import nltk

"""# Load and Prepare Data"""

# Load testing and training data
url = "https://raw.githubusercontent.com/jonarsenault/ecse551data/master/train.csv"
train_data = pd.read_csv(url)

url = "https://raw.githubusercontent.com/jonarsenault/ecse551data/master/test.csv"
test_data = pd.read_csv(url)

# Parameters
number_of_samples = None # Set to None to test entire data set
stop_words = text.ENGLISH_STOP_WORDS
np.random.seed(10)

# For some reason, need to shuffle even if using all data
if number_of_samples is None:
    number_of_samples = len(train_data)

train_data = train_data.sample(number_of_samples).reset_index(drop=True)

X = train_data["body"]
y = train_data["subreddit"]

X_test = test_data["body"]

# Parameters

# Add some stop words
more_stop_words = [
    "u",
    "just",
    "think",
    "https",
    "www",
    "don't",
    "like",
    "need",
    "it",
    "you're",
    "use",
    "reddit",
    "thing",
    "I'm",
    "things",
    "good",
    "really",
    "want",
    "maybe",
    "imgur",
    "com",
    "don",
    "actually",
    "that",
    "make",
    "lot",
    "different",
    "doing",
    "that",
    "better",
    "going",
    "great",
]

fewer_stop_words = [
    "u",
    "just",
    "think",
    "don't",
    "like",
    "need",
    "it",
    "you're",
    "use",
    "thing",
    "I'm",
    "things",
    "good",
    "really",
    "want",
    "maybe",
    "don",
    "actually",
    "that",
    "make",
    "lot",
    "different",
    "doing",
    "that",
    "better",
    "going",
    "great",
]
almost_all_stop_words = stop_words.union(fewer_stop_words)
all_stop_words = stop_words.union(more_stop_words)

"""# Define our own Naive Bayes class"""

class NaiveBayes:
    def __init__(self, alpha=0.01, prior="learn"):
        """Constructor"""
        self.alpha = alpha
        self.prior = prior
    def fit(self, X, y):
        """Obtain naive bayes parameters from training data. X is input data,
        y are class labels"""

        # Convert sparse array to dense array
        X = X.toarray()

        # Compute each class probability
        class_counts = y.value_counts()

        num_labels = len(class_counts)

        if self.prior == "learn":
          # Learn the class probabilities from the training data
          self.class_probabilities = class_counts / len(y)
        elif self.prior == "uniform":
          # Assume a uniform prior
          self.class_probabilities = pd.Series(np.repeat(1/num_labels, num_labels),
                                             index = class_counts.index)

        # Sort in alphabetical order
        self.class_probabilities.sort_index(inplace=True)
        class_counts.sort_index(inplace=True)

        # Compute parameters
        features_count = np.empty((num_labels, X.shape[1]))

        y_numpy = y.to_numpy()
        for i in range(num_labels):

            label = self.class_probabilities.index[i]
            X_this_label = X[np.nonzero(y_numpy == label), :]

            features_count[i,:] = np.sum(X_this_label, axis=1)

        # Laplace smoothing
        smoothed_numerator = features_count + self.alpha
        smoothed_denominator = np.sum(smoothed_numerator,axis=1).reshape(-1,1)

        self.parameters = pd.DataFrame(smoothed_numerator / smoothed_denominator,
            index=self.class_probabilities.index)

    def predict(self, X):
        """Predict class of text"""

        X = X.toarray()

        delta = pd.DataFrame(columns=self.class_probabilities.index)
        for label in self.class_probabilities.index:

            # Get probability of currect class P(y=k)
            class_probability = self.class_probabilities[label]

            # Get theta_j for currect class
            theta_j_class = self.parameters.loc[label, :].to_numpy()

            # Compute P(x_j | y = k)
            prob_features_given_y = (theta_j_class ** X) * (1 - theta_j_class) ** (
                1 - X
            )

            # Compute P(x | y = k)
            prob_sample_given_y = np.prod(prob_features_given_y, axis=1)

            # Compute P(y) * P(x | y = k)
            term1 = np.log(class_probability)
            term2 = np.sum(X * np.log(theta_j_class), axis=1)
            term3 = np.sum((1 - X) * np.log(1 - theta_j_class), axis=1)
            delta_k = term1 + term2 + term3

            # Append
            delta[label] = delta_k

        predicted_class = delta.idxmax(axis=1)

        return predicted_class.to_list()

    def score(self, X, y):
        """Compute accuracy of naive bayes model"""

        y_pred = self.predict(X)

        accuracy = np.count_nonzero(y == y_pred) / len(y_pred)

        return accuracy

    def get_params(self, deep=True):
        """Getter for parameters"""

        params = {"alpha": self.alpha,
                  "prior": self.prior}

        return params

    def set_params(self, **parameters):
        "Setter for parameters"
        for parameter, value in parameters.items():
            setattr(self, parameter, value)

        return self

"""# Run grid search to determine best parameters for Naive Bayes model"""

# After extensive offline testing, it was determined that these are the parameters
# that should be tuned.

# final best result:

# count vectorzier vs TFIDF vectorzier --> choose count vectorzier
# vectorzier stop words --> all stop words
# Lemmatizer or Stemming --> Neither!
# removing stop word --> add more stop words
# N-gram --> None (1,1)
# Chi2 Best K --> k = 30000
# normalizer --> Yes!
# Naive bayes alpha --> 0.01
# Class prior --> uniform!



pipe_params = {
    "vect__stop_words": [None, stop_words, all_stop_words],
    "vect__ngram_range": [(1,1), (1,2), (1,3)],
    "selecter__k":[5000, 15000, 25000],
    "classify__alpha" : [0.001, 0.01, 0.1],
    "classify__prior" : ["uniform", "learn"],
}
```

```
vectorizer = CountVectorizer()
selecter = SelectKBest(chi2)
normalizer = Normalizer()
naive_bayes_model = NaiveBayes()

pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("selecter", selecter), ("classify",
        NaiveBayes())]
)

grid = model_selection.GridSearchCV(pipe, pipe_params, verbose=1, n_jobs=-1)

grid.fit(X, y)

print(f"The best accuracy is {grid.best_score_}.")
print(f"The winning parameters are {grid.best_params_}")

"""# Generate plot of grid search for smoothing parameter"""

# Using the best parameters from the more general grid search, a more specific
# grid search is used to determine the effect of alpha

# Define search parameters
alpha_grid = np.logspace(-3, 0, 4)
pipe_params = {
    "classify__alpha" : alpha_grid,
}

# Create pipeline
vectorizer = CountVectorizer(stop_words=all_stop_words)
selecter = SelectKBest(chi2, k=15000)
normalizer = Normalizer()
naive_bayes_model = NaiveBayes(prior="uniform")

pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("selecter", selecter), ("classify",
        NaiveBayes())]
)

# Run grid search
grid = model_selection.GridSearchCV(pipe, pipe_params, verbose=1, n_jobs=-1)
grid.fit(X, y)

# Plot results
save_fig = True
fig, ax = plt.subplots()
ax.plot(alpha_grid, grid.cv_results_["mean_test_score"])
ax.set_xscale("log")
ax.set_xlabel(r"$\alpha$")
ax.set_ylabel("Mean Test Score")
ax.grid()

ax2 = ax.twinx()
ax2.plot(alpha_grid, grid.cv_results_["mean_fit_time"], color="r", linestyle="--")
ax2.set_ylabel("Mean Fit Time [s]")
if save_fig:
  plt.tight_layout()
  plt.savefig("alpha_grid_search.pdf", bbox_inches="tight", pad_inches=0)

"""# Generate plot of grid search for number of features to select"""

# Define search parameters
k_grid = np.linspace(5000, 35000, 7, dtype=int)
pipe_params = {
```

```python
    "selecter__k" : k_grid,
}

# Create pipeline
vectorizer = CountVectorizer(stop_words=all_stop_words)
normalizer = Normalizer()
selecter = SelectKBest(chi2)
naive_bayes_model = NaiveBayes(alpha = 0.01, prior="uniform")

pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("selecter", selecter), ("classify",
        NaiveBayes())]
)

# Run grid search
grid = model_selection.GridSearchCV(pipe, pipe_params, verbose=1, n_jobs=-1)
grid.fit(X, y)

# Plot results
save_fig = True
fig, ax = plt.subplots()
ax.plot(k_grid, grid.cv_results_["mean_test_score"])
ax.set_xlabel("k")
ax.set_ylabel("Mean Test Score")
ax.grid()

ax2 = ax.twinx()
ax2.plot(k_grid, grid.cv_results_["mean_fit_time"], color="r", linestyle="--")
ax2.set_ylabel("Mean Fit Time [s]")
if save_fig:
  plt.tight_layout()
  plt.savefig("k_grid_search.pdf", bbox_inches="tight", pad_inches=0)

"""# Some results on selecting the best features"""

# Instantiate models
vectorizer = CountVectorizer()
normalizer = Normalizer()
selecter = SelectKBest(chi2)

# Fit and transform variables
X_new = vectorizer.fit_transform(X)
X_new = normalizer.fit_transform(X_new)
X_new = selecter.fit_transform(X_new, y)

# Print the 10 most important words
word_list = np.array(vectorizer.get_feature_names())

selected_words = pd.DataFrame(data={"words" :word_list[selecter.get_support()]})
selected_words["scores"] = selecter.scores_[selecter.get_support()]

print(selected_words.sort_values(by="scores", ascending=False)[0:10])


# Plot p-value of each feature
save_fig = False
fig, ax = plt.subplots()
ax.plot(np.arange(len(selecter.scores_)), sorted(selecter.pvalues_, reverse=True))
ax.set_xlabel("Rank")
ax.set_ylabel("p-value")
ax.grid()
if save_fig:
  plt.tight_layout()
  plt.savefig("p_value.pdf", bbox_inches="tight", pad_inches=0)
```

```
save_fig = False
fig, ax = plt.subplots()
ax.plot(np.arange(len(selecter.scores_)), sorted(selecter.scores_, reverse=True))
ax.set_xlabel("Rank")
ax.set_yscale("log")
ax.set_ylabel("Chi-squared Statistic")
ax.grid()
if save_fig:
  plt.tight_layout()
  plt.savefig("chi_squared_stat.pdf", bbox_inches="tight", pad_inches=0)

"""# Fit best Naive Bayes model"""

vectorizer = CountVectorizer(stop_words=all_stop_words)
selecter = SelectKBest(chi2, k=25000)
normalizer = Normalizer()
naive_bayes_model = NaiveBayes(alpha = 0.01, prior="uniform")
pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("selecter", selecter), ("classify",
        naive_bayes_model)]
)

cross_val_accuracy = model_selection.cross_val_score(pipe, X, y, n_jobs=-1)

print(f"The 5-fold cross-validation accuracy is: {np.mean(cross_val_accuracy):.5f}")

def store_csv(data, outfile_name):
  """Save file for submission to kaggle"""

  rawdata= {'subreddit':data}
  a = pd.DataFrame(rawdata, columns = ['id','subreddit'])
  a.to_csv(outfile_name,index=True, header=True)
  print ("File saved.")


# Run on test set and save output
pipe.fit(X, y)
print(f"The training accuracy is: {pipe.score(X, y):.5f}")

y_pred = pipe.predict(X_test)

store_csv(y_pred,"naive_bayes.csv")

"""# Run grid search to determine best parameters for a Support Vector Machine, chosen as the
    "other" classifier"""

pipe_params = {
    "vectorizer__stop_words": [None, all_stop_words],
    "vectorizer__ngram_range": [(1,1), (1,2), (1,3)],
    "classify__gamma" : np.logspace(-3, 1, 5),
    "classify__C" : np.logspace(0, 4, 5)
}

vectorizer = TfidfVectorizer()
normalizer = Normalizer()
svm_model = svm.SVC(kernel="rbf")

pipe = Pipeline(
    [
        ("vect", vectorizer),
        ("norm", Normalizer()),
        ("classify", svm_model()),
    ]
)
```

```
grid = model_selection.GridSearchCV(pipe, pipe_params, verbose=2, n_jobs=7)

grid.fit(X, y)

print(f"The best accuracy is {grid.best_score_}.")
print(f"The winning parameters are {grid.best_params_}")

"""# Fit best SVM model"""

vectorizer = TfidfVectorizer(stop_words=all_stop_words, ngram_range=(1,2))
normalizer = Normalizer()
svm_model = svm.SVC(kernel="rbf", gamma=0.1, C=10)
pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("classify", svm_model)]
)

cross_val_accuracy = model_selection.cross_val_score(pipe, X, y, n_jobs=-1)

print(np.mean(cross_val_accuracy))

"""# Bagging and Stacking (Not reported in result)"""

## Ensemble version: Bagging
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.datasets import make_classification

start = time.time()

# four

# clf_svm = svm.SVC(kernel="rbf", gamma=0.1, C=10)
# The 5-fold cross-validation accuracy is: 0.86

# clf = MultinomialNB(alpha=0.01, fit_prior = False)
# The 5-fold cross-validation accuracy is: 0.90667

clf = RandomForestClassifier(max_depth= None, n_estimators=75, random_state=1)
# The 5-fold cross-validation accuracy is: 0.80228

clf = BaggingClassifier(base_estimator = clf,
                      n_estimators=100, random_state=13)

vectorizer = CountVectorizer(stop_words=all_stop_words)
normalizer = Normalizer()
selecter = SelectKBest(chi2, k=15000)

pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("selecter", selecter), ("classify", clf)]
)
cross_val_accuracy = model_selection.cross_val_score(pipe, X, y, n_jobs=-1)

print(f"The 5-fold cross-validation accuracy is: {np.mean(cross_val_accuracy):.5f}")

end = time.time()
print(f"Execution Time: {end-start:.2f} s")

## Ensemble version: Stacking
from sklearn.ensemble import RandomForestClassifier,VotingClassifier
from sklearn.naive_bayes import MultinomialNB

start = time.time()
clf_nb = MultinomialNB(alpha=0.01, fit_prior = False)
clf_svm = svm.SVC(kernel="rbf", gamma=0.1, C=10)
```

```python
clf_rf = RandomForestClassifier(n_estimators=50, random_state=1)

eclf1 = VotingClassifier(estimators=[
       ('clf_nb',clf_nb), ('clf_svm', clf_svm), ('clf_rf',clf_rf)], voting='hard')


vectorizer = CountVectorizer(stop_words=all_stop_words)
normalizer = Normalizer()
selecter = SelectKBest(chi2, k=15000)

pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("selecter", selecter), ("classify", eclf1)]
)
cross_val_accuracy = model_selection.cross_val_score(pipe, X, y, n_jobs=-1)

print(f"The 5-fold cross-validation accuracy is: {np.mean(cross_val_accuracy):.5f}")

end = time.time()
print(f"Execution Time: {end-start:.2f} s")
```

Code2:miniproject_2supplemental.py

```
# -*- coding: utf-8 -*-
"""miniproject2_supplemental.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/14CYONUuhLPM1noivLRckkQobWgQzcwhw

# Mini-Project 2 Supplemental Results
"""

# Import relevant modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.preprocessing import Normalizer
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.feature_extraction import text
from sklearn import model_selection
from sklearn import svm
import time
import nltk

"""# Load and Prepare Data"""

# Load testing and training data
url = "https://raw.githubusercontent.com/jonarsenault/ecse551data/master/train.csv"
train_data = pd.read_csv(url)

url = "https://raw.githubusercontent.com/jonarsenault/ecse551data/master/test.csv"
test_data = pd.read_csv(url)

# Parameters
number_of_samples = None # Set to None to test entire data set
stop_words = text.ENGLISH_STOP_WORDS
np.random.seed(10)

# For some reason, need to shuffle even if using all data
if number_of_samples is None:
    number_of_samples = len(train_data)

train_data = train_data.sample(number_of_samples).reset_index(drop=True)

X = train_data["body"]
y = train_data["subreddit"]

X_test = test_data["body"]

# Parameters

# Add some stop words
more_stop_words = [
    "u",
    "just",
    "think",
    "https",
    "www",
    "don't",
    "like",
    "need",
    "it",
    "you're",
    "use",
    "reddit",
    "thing",
    "I'm",
    "things",
    "good",
    "really",
    "want",
    "maybe",
    "imgur",
    "com",
    "don",
    "actually",
    "that",
    "make",
    "lot",
    "different",
    "doing",
    "that",
    "better",
    "going",
    "great",
]

fewer_stop_words = [
    "u",
    "just",
    "think",
    "don't",
    "like",
    "need",
    "it",
    "you're",
    "use",
    "thing",
    "I'm",
    "things",
    "good",
    "really",
    "want",
    "maybe",
    "don",
    "actually",
    "that",
    "make",
    "lot",
    "different",
    "doing",
    "that",
    "better",
    "going",
    "great",
]
almost_all_stop_words = stop_words.union(fewer_stop_words)
all_stop_words = stop_words.union(more_stop_words)

"""# Define Naive Bayes class"""

class NaiveBayes:
    def __init__(self, alpha=0.01, prior="learn"):
        """Constructor"""

        self.alpha = alpha
        self.prior = prior
```

```
def fit(self, X, y):
    """Obtain naive bayes parameters from training data. X is input data,
    y are class labels"""

    # Convert sparse array to dense array
    X = X.toarray()

    # Compute each class probability
    class_counts = y.value_counts()

    num_labels = len(class_counts)

    if self.prior == "learn":
      # Learn the class probabilities from the training data
      self.class_probabilities = class_counts / len(y)
    elif self.prior == "uniform":
      # Assume a uniform prior
      self.class_probabilities = pd.Series(np.repeat(1/num_labels, num_labels),
                                  index = class_counts.index)

    # Sort in alphabetical order
    self.class_probabilities.sort_index(inplace=True)
    class_counts.sort_index(inplace=True)

    # Compute parameters
    features_count = np.empty((num_labels, X.shape[1]))

    y_numpy = y.to_numpy()
    for i in range(num_labels):

        label = self.class_probabilities.index[i]
        X_this_label = X[np.nonzero(y_numpy == label), :]

        features_count[i,:] = np.sum(X_this_label, axis=1)

    # Laplace smoothing
    smoothed_numerator = features_count + self.alpha
    smoothed_denominator = np.sum(smoothed_numerator,axis=1).reshape(-1,1)

    self.parameters = pd.DataFrame(smoothed_numerator / smoothed_denominator,
        index=self.class_probabilities.index)

def predict(self, X):
    """Predict class of text"""

    X = X.toarray()

    delta = pd.DataFrame(columns=self.class_probabilities.index)
    for label in self.class_probabilities.index:

        # Get probability of currect class P(y=k)
        class_probability = self.class_probabilities[label]

        # Get theta_j for currect class
        theta_j_class = self.parameters.loc[label, :].to_numpy()

        # Compute P(x_j | y = k)
        prob_features_given_y = (theta_j_class ** X) * (1 - theta_j_class) ** (
            1 - X
        )

        # Compute P(x | y = k)
        prob_sample_given_y = np.prod(prob_features_given_y, axis=1)

        # Compute P(y) * P(x | y = k)
```

```
            term1 = np.log(class_probability)
            term2 = np.sum(X * np.log(theta_j_class), axis=1)
            term3 = np.sum((1 - X) * np.log(1 - theta_j_class), axis=1)
            delta_k = term1 + term2 + term3

            # Append
            delta[label] = delta_k

        predicted_class = delta.idxmax(axis=1)

        return predicted_class.to_list()

    def score(self, X, y):
        """Compute accuracy of naive bayes model"""

        y_pred = self.predict(X)

        accuracy = np.count_nonzero(y == y_pred) / len(y_pred)

        return accuracy

    def get_params(self, deep=True):
        """Getter for parameters"""

        params = {"alpha": self.alpha,
                  "prior": self.prior}

        return params

    def set_params(self, **parameters):
        "Setter for parameters"
        for parameter, value in parameters.items():
            setattr(self, parameter, value)

        return self

"""# Choosing parameters and preprocessing methods for Multinomial Naive Bayes model"""

#Define functions of lemmatization and stemming

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

class LemmaTokenizer_1:
     def __init__(self):
       self.wnl = WordNetLemmatizer()
     def __call__(self, doc):
       return [self.wnl.lemmatize(t,pos ="v") for t in word_tokenize(doc)]

class LemmaTokenizer_2:
     def __init__(self):
       self.wnl = WordNetLemmatizer()
     def __call__(self, doc):
       return [self.wnl.lemmatize(t,pos ="v") for t in word_tokenize(doc) if t.isalpha()]

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()]

"""## No preprocessing"""

## Test with no preprocessing
```

```
t_start = time.time()

vectorizer = CountVectorizer()
naive_bayes_model = NaiveBayes(alpha = 0.01, prior="uniform")

pipe = Pipeline(
    [("vect", vectorizer), ("classify", naive_bayes_model)]
)

cross_val_accuracy = model_selection.cross_val_score(pipe, X, y, n_jobs=-1)

t_end = time.time()

print(f"The 5-fold cross-validation accuracy is: {np.mean(cross_val_accuracy):.5f}")
print(f"Run time: {t_end-t_start: .3f} seconds")

"""## Normalization"""

## Test: normalize

t_start = time.time()

vectorizer = CountVectorizer()
normalizer = Normalizer()
naive_bayes_model = NaiveBayes(alpha = 0.01, prior="uniform")

pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("classify", naive_bayes_model)]
)

cross_val_accuracy = model_selection.cross_val_score(pipe, X, y, n_jobs=-1)

t_end = time.time()

print(f"The 5-fold cross-validation accuracy is: {np.mean(cross_val_accuracy):.5f}")
print(f"Run time: {t_end-t_start: .3f} seconds")

"""## Remove stop words"""

##Test: remove stop-words

t_start = time.time()

vectorizer = CountVectorizer(stop_words=all_stop_words)
normalizer = Normalizer()
naive_bayes_model = NaiveBayes(alpha = 0.01, prior="uniform")

pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("classify", naive_bayes_model)]
)

cross_val_accuracy = model_selection.cross_val_score(pipe, X, y, n_jobs=-1)

t_end = time.time()

print(f"The 5-fold cross-validation accuracy is: {np.mean(cross_val_accuracy):.5f}")
print(f"Run time: {t_end-t_start: .3f} seconds")

"""## Lemmatization"""

##Test: lemmatization that works on colab

t_start = time.time()
```

```
from nltk import word_tokenize
from nltk.stem import WordNetLemmatizer
class LemmaTokenizer:
     def __init__(self):
       self.wnl = WordNetLemmatizer()
     def __call__(self, doc):
       return [self.wnl.lemmatize(t,pos ="v") for t in word_tokenize(doc) if t.isalpha()]
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')

from nltk.corpus import wordnet
from nltk.tokenize.treebank import TreebankWordDetokenizer

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

lemmatizer = WordNetLemmatizer()

X1 = X.copy()

for i in range(X1.shape[0]):
    X1[i] = X1[i].lower()
    X1[i] = [lemmatizer.lemmatize(w, get_wordnet_pos(w)) for w in nltk.word_tokenize(X1[i])]
    X1[i] = TreebankWordDetokenizer().detokenize(X1[i]) #detokenize
print(X1[0])
#Although it was defined in lemmarizer function to remove isalpha, this print shows that it
    doesn't remove all non-alphabetic characters.
#A second step is taken to ensure is alpha in the next test

vectorizer = CountVectorizer()
normalizer = Normalizer()
naive_bayes_model = NaiveBayes(alpha = 0.01, prior="uniform")

pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("classify", naive_bayes_model)]
)

cross_val_accuracy = model_selection.cross_val_score(pipe, X1, y, n_jobs=-1)

t_end = time.time()

print(f"The 5-fold cross-validation accuracy is: {np.mean(cross_val_accuracy):.5f}")
print(f"Run time: {t_end-t_start: .3f} seconds")

"""## Lemmatization and remove non-alphabetic characters"""

#Tests: lemmatization with isalpha that works on colab

t_start = time.time()

X1 = X.copy()
for i in range(X1.shape[0]):
    X1[i] = X1[i].lower()
    X1[i] = [lemmatizer.lemmatize(w, get_wordnet_pos(w)) for w in nltk.word_tokenize(X1[i])]
    X1[i] = [w for w in X1[i] if w.isalpha()] #Remove non-alphabetic words
    X1[i] = TreebankWordDetokenizer().detokenize(X1[i]) #detokenize

vectorizer = CountVectorizer()
```

```
normalizer = Normalizer()
naive_bayes_model = NaiveBayes(alpha = 0.01, prior="uniform")

pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("classify", naive_bayes_model)]
)

cross_val_accuracy = model_selection.cross_val_score(pipe, X1, y, n_jobs=-1)

t_end = time.time()

print(f"The 5-fold cross-validation accuracy is: {np.mean(cross_val_accuracy):.5f}")
print(f"Run time: {t_end-t_start: .3f} seconds")

"""## ngram (1,2) (1,3) (1,4) with max_features=40000"""

##Tests: ngram (1,2)

t_start = time.time()

vectorizer = CountVectorizer(ngram_range=(1, 2), max_features = 40000)
normalizer = Normalizer()
naive_bayes_model = NaiveBayes(alpha = 0.01, prior="uniform")

pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("classify", naive_bayes_model)]
)

cross_val_accuracy = model_selection.cross_val_score(pipe, X, y, n_jobs=-1)

t_end = time.time()

print(f"The 5-fold cross-validation accuracy is: {np.mean(cross_val_accuracy):.5f}")
print(f"Run time: {t_end-t_start: .3f} seconds")

##Tests: ngram (1,3)

t_start = time.time()

vectorizer = CountVectorizer(ngram_range=(1,3), max_features = 40000)
normalizer = Normalizer()
naive_bayes_model = NaiveBayes(alpha = 0.01, prior="uniform")

pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("classify", naive_bayes_model)]
)

cross_val_accuracy = model_selection.cross_val_score(pipe, X, y, n_jobs=-1)

t_end = time.time()

print(f"The 5-fold cross-validation accuracy is: {np.mean(cross_val_accuracy):.5f}")
print(f"Run time: {t_end-t_start: .3f} seconds")

##Tests: ngram (1,4)

t_start = time.time()

vectorizer = CountVectorizer(ngram_range=(1,4), max_features = 40000)
normalizer = Normalizer()
naive_bayes_model = NaiveBayes(alpha = 0.01, prior="uniform")

pipe = Pipeline(
    [("vect", vectorizer), ("norm", normalizer), ("classify", naive_bayes_model)]
)
```

```
cross_val_accuracy = model_selection.cross_val_score(pipe, X, y, n_jobs=-1)

t_end = time.time()

print(f"The 5-fold cross-validation accuracy is: {np.mean(cross_val_accuracy):.5f}")
print(f"Run time: {t_end-t_start: .3f} seconds")
```