Building Robust Sentiment Analysis Model for Twitter Data: Custom and Scikit-learn Naive Bayes Implementation (July 2023)

Kaustuv Karki[1], Undergraduate, IOE, Nikhil Pradhan[2], Undergraduate, IOE
[1]Institute of Engineering Thapathali Campus, Tribhuvan University, Kathmandu, Nepal
[2]Institute of Engineering Thapathali Campus, Tribhuvan University, Kathmandu, Nepal

Corresponding author: Kaustuv Karki (e-mail: karkikaustuv@gmail.com),
Nikhil Pradhan (e-mail: nikhilpradhan20b@gmail.com)

ABSTRACT In the data-driven era of today, the sheer volume of information generated on social media platforms like Twitter has made manual analysis and processing of tweets a time-consuming task. Sentiment analysis, a critical aspect of natural language processing, plays a pivotal role in determining the emotional tone of tweets, enabling businesses and researchers to gauge public opinion on diverse topics. Among various text classification methods, the Naive Bayes algorithm, a probabilistic supervised learning approach, has garnered widespread usage for sentiment analysis due to its simplicity and efficiency. This research focuses on evaluating the effectiveness of the Naive Bayes classifier in accurately classifying Twitter sentiments. The study employs the widely used scikit-learn library to implement the Naive Bayes classifier and leverages a comprehensive Twitter sentiment analysis dataset for evaluation. Throughout the research, the dataset undergoes several iterations, with hyperparameters being fine-tuned to optimize the classifier's performance. Through meticulous analysis and comparison of the obtained results, this study sheds light on the performance of the Naive Bayes classifier for sentiment analysis on Twitter data. Additionally, the research identifies the optimal hyperparameters that lead to achieving accurate sentiment classification. The findings from this study contribute significantly to the understanding of Naive Bayes classification in the context of sentiment analysis, providing valuable insights into its potential benefits for automating and enhancing sentiment analysis processes on social media platforms. As sentiment analysis continues to gain prominence in diverse applications, this research serves as a valuable resource for researchers, practitioners, and businesses seeking efficient techniques to comprehend public sentiment on Twitter and similar platforms.

INDEX TERMS Naïve Bayes, Sentiment Analysis, Twitter

I. INTRODUCTION

With the exponential growth of social media platforms like Twitter, an overwhelming volume of data is generated daily, posing challenges for manual analysis and comprehension. Sentiment analysis on Twitter, which involves determining the emotional tone of tweets, has become increasingly essential for businesses and researchers to gain insights into public opinion. In response to the need for efficient and automated data analysis, machine learning algorithms like the Naive Bayes classifier have gained prominence for their effectiveness in text classification tasks.

Naive Bayes is a probabilistic classification algorithm based on Bayes' theorem that assumes strong independence between features. Despite its simplicity, Naive Bayes has demonstrated remarkable performance in various sentiment analysis tasks. This study aims to explore sentiment classification of tweets using a Naive Bayes classifier implemented from scratch in Python without using external machine learning libraries and comparing with the Naïve Bayes from scikit learn library.

The dataset comprised of tweets labeled as positive or negative sentiment. Text preprocessing techniques like tokenization, stopword removal and stemming were applied to transform the raw tweets into cleaned input for the model. The Naive Bayes algorithm calculated probabilities of words belonging to each sentiment class and made predictions on unseen tweets. The classifier was evaluated using accuracy, precision, recall and F1-score. Furthermore, it was compared with scikit-learns Naive Bayes.

This research provides insights into developing robust Naive Bayes models for sentiment analysis of textual data from social media platforms. The analysis of Naive Bayes classifier on the Twitter dataset demonstrates its efficiency for fast and automatic sentiment classification of tweets.

II. METHODOLOGY

A. BRIRF THEORY

Naive Bayes is a simple machine learning algorithm used for classification tasks, particularly in natural language processing and text analysis. It is based on Bayes' theorem, a fundamental principle in probability theory.

At its core, Naive Bayes assumes that the features in the dataset are conditionally independent of each other, given the class label. In simpler terms, it presumes that the presence or absence of one feature does not influence the presence or absence of another feature, given the class. This "naive" assumption makes the algorithm computationally efficient and scalable, especially in high-dimensional feature spaces.

By assuming conditional independence between features, the algorithm simplifies this joint probability into a product of individual conditional probabilities. This "naive" move significantly reduces the computational complexity, making Naive Bayes highly scalable and efficient even in high-dimensional feature spaces.

The main application of Naive Bayes is in classifying documents or texts into different categories based on the features present in the text. For instance, it can be used for sentiment analysis, where the goal is to determine the sentiment of a given text as positive, negative, or neutral. It can also be used for spam email filtering, text categorization, and other text classification tasks.

Naive Bayes calculates the probability of a document belonging to each category and then assigns the document to the category with the highest probability. It estimates the prior probabilities of each category (the probability of each category occurring in the dataset) and the likelihood probabilities of observing specific features given each category.

The Naive Bayes algorithm is particularly useful when dealing with many features or words in a text, as it can handle this complexity efficiently. Naïve Bayes works well even with limited training data, making it a suitable choice for cases where large, labeled datasets are not available.

Naive Bayes is known for its ability to handle multiclass classification problems, making it valuable in scenarios where data can be categorized into more than two classes. Its versatility allows it to be applied to a wide range of problems, such as language identification, topic modeling, and fraud detection. Due to its simplicity and efficiency, Naive Bayes is often used as a baseline model for benchmarking and comparing the performance of other more complex algorithms.

Another advantage of Naive Bayes is its interpretability. The probabilities generated by the algorithm provide insights into the decision-making process, allowing users to understand why a certain classification was made. This interpretability is especially crucial in applications where explanations of predictions are required, such as in legal or medical contexts.

Moreover, Naive Bayes is well-suited for handling text data with sparse features. In many natural language processing tasks, the feature space can be highly sparse, as not all words or features may be present in every document. The algorithm's ability to handle such sparse data efficiently makes it suitable for processing large-scale text datasets without significant computational overhead.

Despite its advantages, Naive Bayes has its limitations. One of the main drawbacks is its strong assumption of conditional independence among features [1]. Some features may be correlated or dependent on each other, which can lead to suboptimal performance in certain cases. However, despite this simplifying assumption, Naive Bayes often performs surprisingly well in practice, making it a reliable and straightforward option for many classification tasks.

The variants of Naïve Bayes are:

1) GAUSSIAN *NAIVE* BAYES:

Gaussian Naive Bayes assumes that the features follow a Gaussian (normal) distribution. It is suitable for continuous features (real-valued data). This variant is often used in scenarios where features have continuous values, like measurements or sensor data.

2) MULTINOMIAL NAIVE BAYES:

Multinomial Naive Bayes is designed for discrete features, particularly for text classification tasks. It works well with features represented as word counts or frequencies. As such, it is commonly applied in natural language processing for tasks like spam filtering, sentiment analysis, and topic categorization.

3) BERNOULLI NAIVE BAYES:

Bernoulli Naive Bayes is like Multinomial Naive Bayes but is used for binary features. It works well when dealing with features that are binary, such as yes/no or the presence/absence of a term in text.

4) CATEGORICAL NAIVE BAYES:

Categorical Naive Bayes is suitable for categorical features with a fixed number of categories. It can handle features like colors (red, blue, green) or any other categorical variables with discrete outcomes.

5) COMPLEMENT NAIVE BAYES:

Complement Naive Bayes is an extension of Multinomial Naive Bayes designed to address class imbalance problems. It is particularly useful when the training data has imbalanced class distributions.

B. SYSTEM BLOCK DIAGRAM

From the Figure 1 we can see the following things:

The dataset is divided into two sets: the training set, which is used to train the Naive Bayes classifier on labeled data, and the test set, which is used to evaluate the model's performance on unseen data. This separation ensures that the model does not memorize the training data but instead learns to generalize well to new and unseen tweets. By training on a separate portion of the data and testing on another, we can assess how well the Naive Bayes classifier can classify sentiments in tweets that it has not seen before.

The vocabulary represents the collection of all distinct words that occur in the training data. Each unique word in the vocabulary becomes a feature used by the Naive Bayes classifier to make sentiment predictions. By analyzing the vocabulary, we gain insights into the variety and diversity of language expressions present in the tweets. The size of the vocabulary directly influences the dimensionality of the feature space and affects the complexity of the classification task.

Then we proceed to the process of counting the occurrences of positive and negative words in the training set, which serves as the foundation for understanding sentiment patterns. Additionally, we create separate sets of tweets that are labeled as positive and negative based on the sentiment annotations provided in the training data. These sets comprise tweets associated with the respective sentiment categories, allowing us to analyze and model the distinctive characteristics of each sentiment class.

The log prior represents the logarithm of the probability that a randomly selected tweet belongs to a specific sentiment category, such as positive or negative. To compute the log prior, we first count the number of tweets in the training set that are labeled as positive and negative, respectively. Then, we divide numbers of positive tweets by the count of total number of negative tweets in the training set. Taking the logarithm of these probabilities helps avoid numerical underflow issues that may arise with very small probabilities.

The log likelihood represents the logarithm of the conditional probability that a particular word appears in tweets belonging to a specific sentiment category, such as positive or negative. To compute the log likelihood, we first count the occurrences of each word in the tweets labeled as positive and negative, respectively. Loglikelihood is taken to overcome the problem of underflow. Likelihood value is taken to classify whether a tweet is positive or negative. The threshold value for classification for likelihood only is taken as 1, whereas for loglikelihood this value changes to 0. If the value of loglikelihood is greater than 0, it is categorized as tweet with positive sentiment else if the value of the loglikelihood is less than 0, it is categorized as tweet with negative sentiment.

We can see that the following steps were performed during the process giving a basic pipeline for the process. But before feeding the information to the pipeline the initial text dataset must be preprocessed which can be seen in Figure 2.

Before applying Naïve Bayes, the dataset needs to be preprocessed. Firstly, each word in the tweet is checked if it’s a stop word or not. Stop words are common words like "the," "is," "and" "in," etc., which appear frequently in texts but do not carry significant meaning for sentiment analysis. These words can introduce noise and unnecessary complexity to the classification process, potentially hindering the performance of the classifier. By eliminating stop words from the text data before training the Naive Bayes model, we can improve the efficiency and accuracy of the sentiment analysis task.

After removing stop words, all the words are converted to their lowercase form so that two values of frequency are not generated for the same two words for example, “GOOD” & “good”. By standardizing the text, the classifier can focus on learning meaningful sentiment patterns without being influenced by different letter cases. Lowercasing facilitates accurate sentiment analysis and aids in creating a cohesive representation of the data.

Removal unnecessary elements like hyperlinks, Twitter handles, and special characters is a vital data preprocessing step. These elements do not contribute to the sentiment of the text and may introduce noise in the analysis. By eliminating hyperlinks and Twitter handles, the focus remains on the actual content and sentiments expressed in the tweets. Additionally, special characters, hashtags, and emojis that often accompany tweets can be removed to ensure a cleaner and more concise text representation. This process streamlines the data, enabling the Naive Bayes classifier to focus on the meaningful words and sentiment.

Stemming involves reducing words to their base or root form, which helps in consolidating different variations of the same word. For example, words like "running," "runs," and "ran" would all be stemmed to "run." By applying stemming, we can reduce the dimensionality of the feature space, making it easier for the Naive Bayes classifier to generalize and capture the underlying sentiment patterns effectively. This process not only reduces the computational complexity but also helps in handling variations in word forms commonly seen in tweets.

C. DATASET

The twitter sentiment dataset which in total consist of 10000 tweets which contains a collection of 5000 positive and 5000 negative tweets that have been manually labeled. The tweets are a mix of recent and historical tweets, and they cover a wide range of topics. Positive tweets typically convey positive emotions or opinions, negative tweets express negative emotions or opinions, and neutral tweets are generally objective or don't carry strong emotional content.

The examples of positive tweets are:

1) *"It will all get better in time. :)"*
2) *"@messiholic_ Lol really? I can't believe a beautiful girl like you are single :p"*
3) *"@normabattle thx for the feedback. Glad you gained value from it. ??"*

The examples of negative tweets are:

4) "No Assignment, but we have Project. ?? really? ??"
5) "just want to play video games/watch movies with someone :("
6) "when will you notice me ?? i'm so hopeless ?? @SeaveyDaniel"
7) "@Kimwoobin89__ oppa............/hug ur arm/ why am I like this ??"

D. MAJOR MATHEMATICAL FORMULAS

1) PROBABILITY

Probability helps to predict an event's occurrence out of all the potential outcomes. The probability of event lies between 0 and 1 meaning 0<= P(A) <= 1

$$P(A) = \frac{n(A)}{n(S)} \tag{1}$$

Where:
P(A) = probability of an event A
n(A) = is the number of favorable outcomes for event A
n(S) = the total number of possible outcomes

2) CONDITIONAL PROBABILITY

Conditional probability is a measure of the likelihood of an event occurring given that another event has already occurred. It quantifies the probability of one event happening under the condition that we know another event has occurred.

$$P(A|B) = \frac{P(A \cap B)}{P(B)} \tag{2}$$

Where:
P(A/B) = Conditional probability of event A given event B.
P (A ∩ B) = The probability of both events A and B occurring.
P(B) = The probability of event B occurring.

3) BAYES RULE

$$P(A \cap B) = P(A \mid B) * P(B) \tag{3}$$

$$P(A \cap B) = P(B \mid A) * P(A) \tag{4}$$

Where:
P(A/B) = The conditional probability of event A given event B has occurred.
P(B/A) = The conditional probability of event B given event A has occurred.
P(A) = The probability of event A occurring.
P(B) = The probability of event B occurring.

4) BAYES THEOREM

$$P(A \mid B) = \frac{P(B \mid A) * P(A)}{P(B)} \tag{5}$$

Where:
P(A/B) = probability of instance B being in class A
P(B/A) = probability of generating instance B given class A
P(A) = probability of A
P(B) = probability of B

5) LIKELIHOOD

Likelihood refers to the probability of observing the data given a particular set of model parameters.

$$L(\theta) = \prod_{i=1}^{m} \frac{P(\omega_i|pos)}{P(\omega_i|neg)} \tag{5}$$

Where:
L(θ) = This represents the likelihood function
m = refers to the number of words
$P(\omega_i|pos)$ = conditional probability of positive word
$P(\omega_i|neg)$ = conditional probability of negative word
θ = parameters of classifier

6) LOG LIKELIHOOD

The log-likelihood function is the natural logarithm of the likelihood function

$$l(\theta) = \log\left(L(\theta)\right) \tag{6}$$

Where:
l(θ) = This represents the log likelihood function

C. INSTRUMENTATION

In the Twitter sentiment analysis project, several Python libraries were instrumental in processing and analyzing the data. The Natural Language Toolkit (NLTK) proved to be a valuable resource for text processing tasks such as tokenization, stopwords removal, and stemming. Particularly, the TweetTokenizer class from NLTK was employed to handle Twitter-specific features like hashtags and mentions while retaining the original word case. Additionally, NLTK's Porter Stemmer efficiently reduced words to their root forms, aiding in standardizing the text data.

For data visualization, the Matplotlib library played a key role in generating informative plots. Bar plots were used to visualize the distribution of positive and negative tweets in the dataset, while scatter plots provided insights into the relationship between positive and negative word frequencies. The Seaborn library complemented Matplotlib by enhancing the visualizations with its appealing and statistically meaningful graphics.

To facilitate the analysis, specific functions were designed to streamline various tasks. The preprocess(tweet) function effectively cleaned each tweet by removing retweet tags, hyperlinks, hashtags, stopwords, and punctuation. Furthermore, the count_tweets(result, tweets, ys) function diligently counted the frequency of words in both positive and negative tweets, enabling the extraction of essential features for the classifier.

The core of the project relied on the Naive Bayes classifier, implemented in the naive_bayes_train(freq, train_x, train_y) function. During the training phase, this function computed the log prior and log likelihood values, crucial for making sentiment predictions. Subsequently, the naive_bayes_predict(tweet, logprior, loglikelihood) function was employed to classify test tweets as positive or negative based on the learned probabilities.

Finally, the test_naive_bayes(test_x, test_y, logprior, loglikelihood) function performed the evaluation of the Naive Bayes classifier on the test set. By comparing the predicted labels with the true labels, the accuracy of the classifier was assessed, providing a measure of its performance.

A. EXPLORATORY DATA ANALYSIS

The dataset was obtained from the NLTK corpus from the twitter samples and the dataset contains 5000 positive tweets and 5000 negative tweets. Among those tweets 80 % of each positive tweets and negative tweets were taken as the train dataset which amounted to 4000 tweets from the positive tweets and 4000 tweets from the negative tweets. Combining the total gave us the number 8000 as our final train size. Similarly, the extra 20 % of the dataset which amounted to a total of 1000 from the positive tweets and 1000 from the negative tweets whose total is 2000 tweets were taken as a test dataset for evaluation of the models.

Firstly, the words as were taken and the words and word length present in the dataset were seen. Initially the whole dataset was explored to find the basic format of the dataset. There were no null values present in the dataset do functions such as imputations and such sort were not necessary to be performed. We got the initial dataset in the form of positive tweets and negative tweets so by combing them and performing the analysis of the structure of the dataset we get the following results. From Figure 3 we can see that it shows the number of characters present in the sentence from the positive and negative tweets. It consists of the values before any preprocessing was performed. We can see that there is a high number of tweets in the values 30 characters to 50 characters and there is a sudden spike at the end at 140 characters. From seeing this we can analyze that most of the tweets are short, and some tweets are longer which may contain a lot of non-alphanumeric values increasing the count of characters in the tweet.

Similarly, studying the number of words present in the positive and negative tweets we can see that most of the words present in the tweet are in the front portion of the curve as we can see in the plot in Figure 4. We can see that the graph is high in the initial portion of the plot mainly from the range of 5 words to 10 words and then slowly falls off. We can also see some gaps being that there are no tweets of that word range, and the tweets finish at just over 30 words giving us a conclusion that there are no tweets that are above 30 words making the length of tweets short.

Now by combing the positive and negative tweets into a whole dataset we then split the total dataset into two portions, a test set, and a train set. The train and test dataset were separately observed to bring forth various conclusions and the data present in the train and test set were explored. Initially about the train dataset which contains 8000 tweets. From Figure 5 we can see that the words present in the dataset in the form of word clouds. We can see that the words that are repeated more appear bigger in the plot and the words with less repetition appear smaller. From the plot we can see that the words ‘im’ and ‘d’ appear a lot in the dataset. Other words such as ‘thank’, ‘make’, ‘want’, ‘hope’, ‘love’ and many more also appear in the wordcloud signifying the presence of those words in the dataset. Then plotting the highest number of repeating words in the dataset we find that the top 15 repeating words in the dataset are ‘thank’, ‘follow’, ‘love’ which can also be seen big in the wordcloud as indicated in Figure 6.

Similarly doing the same process for test dataset we can deduce the following. We can see that the words “D”, “Im”, “thank”, “day” appear to have higher frequency in the dataset as can be seen in Figure 7. Then plotting the bar graph of 15 most frequent words in the dataset as shown in Figure 8 we can see that the most frequent words are “you”, “the”, “and” which are to be expected.

.

B. NAÏVE BAYES IMPLEMENTATION

The training set contained 4000 positive and 4000 negative tweets, while the test set contained the remaining tweets. The distribution of positive and negative tweets in the original dataset was visualized using a bar plot, showing a relatively balanced dataset for sentiment analysis.

Preprocessing steps were applied to the tweets before training the Naive Bayes classifier. The preprocessing included removing retweet text (RT), hyperlinks, and hashtags from the tweets. The tweets were tokenized to obtain individual words, and then stop words and punctuation were removed. Stemming was performed to reduce words to their base form, facilitating efficient analysis by grouping similar words together.

The count of positive and negative words and tweets was calculated, and Laplace smoothing was applied to handle the issue of unseen words in the test set. This technique added a small constant value to the word frequencies, ensuring that no word has a probability of zero. As a result, the classifier can make predictions even for words do not present in the training data, enhancing its ability to generalize to new data.

During the testing phase, the Naive Bayes classifier with Laplace smoothing predicted the sentiment of test tweets. The accuracy of the model was calculated by comparing the predicted labels with the true labels of the test set. Accuracy was the key performance metric to evaluate the effectiveness of the classifier.

The accuracy achieved on the test set was observed to be high, indicating that the Naive Bayes classifier with Laplace smoothing performed well in sentiment analysis on Twitter data. The inclusion of Laplace smoothing ensured that the model could handle previously unseen words and reduced the risk of overfitting to the training data. To further enhance the classifier's performance, additional strategies can be explored. One approach could involve using more advanced text preprocessing techniques to handle slang, emoticons, and abbreviations commonly used in social media data. Additionally, experimenting with different feature engineering methods, such as using n-grams or term frequency-inverse document frequency (TF-IDF), could potentially capture more meaningful patterns in the tweets. Incorporating domain-specific knowledge or sentiment lexicons might boost the classifier's accuracy. This could involve using external resources or domain-specific dictionaries to aid in sentiment analysis and capture contextual sentiments better.

From that testing and evaluation, we find that the accuracy of the custom naïve bayes model on the twitter sentiment dataset was found to be 99.5%. The probability of the tweets being of negative or positive sentiment were calculated using the log likelihood of the words present in the tweet. This model estimates the likelihood of words occurring in positive versus negative tweets. The excellent performance indicates the probabilistic model can capture strong signals for discriminating between positive and negative sentiment. Despite its simplicity, the frequency-based probabilities provide a powerful approach for text classification when ample training data is available.

After applying the custom Naïve bayes on the twitter sentiment analysis dataset scikit learn library was used and the functions from the scikit learn library were used for the process of classification using Naïve bayes. Initially Gaussian Naïve Bayes was used on the dataset. Fitting the dataset in the Gaussian Naïve Bayes we see that the accuracy score for this implementation of the Naïve Bayes was found to be around 60% which is low. The reason that the gaussian naïve bayes accuracy score is low because it is assumed that the features present in the dataset follow gaussian distribution and are continuous in nature which is not true for the twitter dataset.
Overall, the classification report reveals that the model performs better in classifying instances of class "0" (negative sentiment) with relatively high precision and recall. However, it struggles to classify instances of class "1" (positive sentiment) with lower recall, indicating that it tends to miss some positive instances. The relatively low F1-scores suggest that there is room for improvement in the model's performance in both classes. The results of the Gaussian Naïve Bayes are seen in Table I.

After applying Multinomial Naïve Bayes classifier on the twitter sentiment analysis, we get the accuracy of 74% which is greater than the original implementation of gaussian naïve bayes. This value of accuracy of the multinomial Naïve bayes is higher because the data consists of discrete features representing the frequency of words (word counts) Multinomial Naive Bayes is specifically designed to handle such discrete data, making it a more accurate for the classification task at hand. But using Gaussian Naive Bayes with discrete data, such as word frequencies, results in a poor fit. The classification report for the Multinomial Naïve Bayes can be shown in Table II. The Multinomial Naive Bayes model shows reasonably balanced performance between the two classes, with similar precision, recall, and F1-score for both negative and positive sentiments. The model's accuracy of 74% suggests its ability to make accurate predictions on the given dataset. After using GridSearchCV for the optimal parameter for 'alpha' for the dataset which was found to be 2.0 the accuracy for the classification task raises to about 74.25%.

Finally implementing the Complement Naïve Bayes on the twitter sentiment analysis dataset, we get an accuracy of 74.05% which is higher than both Gaussian Naïve Bayes and the Multinomial Naïve Bayes Implementation. We can assume that Complement Naïve Bayes performs better than Multinomial Naïve Bayes because for text classification tasks due to its ability provide robust estimates for rare features and overcome limitations of the bag-of-words assumption by using the complement feature to model absence of words. Complement Naïve Bayes weighted counting approach and class-specific distributions contribute to improved accuracy in scenarios where Multinomial Naïve Bayes may suffer from biased predictions and suboptimal performance. Also, by tuning the hyperparameter 'alpha' present in the Complement Naïve Bayes we can increase the accuracy score to the value of 74.2 % which is higher than any other scikit learn implementation present.

III. DISCUSSION

In this work, we developed and evaluated both a custom Naive Bayes implementation and scikit-learns Naive Bayes classifiers for sentiment analysis on Twitter data. Our custom Naive Bayes classifier achieved 99% accuracy on the benchmark twitter sentiment dataset from NLTK corpus, significantly outperforming the scikit-learn Gaussian (60%), multinomial naïve bayes (74%), and complement naïve bayes (74.25%). The superior performance of the custom implementation may be the result of tuning the Naive Bayes algorithm for the specific Twitter sentiment task.
In contrast, the scikit-learn classifiers use their default settings which may not be optimized for Twitter data. The Gaussian model assumes gaussian distributions which is violated. The multinomial and complement models apply regularization and feature selection that likely discard useful sentiment signals.

Grid search was used to tune the hyperparameters of the scikit-learn models which included alpha for smoothing.

This improved their accuracy to 74.1% for multinomial and 74.25% for complement naïve bayes models. However, this still lags far behind the custom model.

IV. CONCLUSION

This research explored sentiment classification on Twitter data using both a custom Naive Bayes implementation and scikit-learns Naive Bayes classifiers. The custom Naive Bayes model, tailored specifically for the Twitter dataset, significantly outperformed the default scikit-learn classifiers, achieving 99% accuracy compared to 60-74% for the scikit-learn models. The superior performance of the custom Naive Bayes implementation demonstrates the importance of adapting the algorithm and preprocessing steps to the nuances of the data. Key factors in its success included building vocabulary from the training tweets, converting to lowercase, removing stop words/punctuation, stemming, and Laplace smoothing. Together these steps enabled the model to capture predictive signals from the informal language and sparse features in tweets. In contrast, scikit-learns default settings are not optimized for Twitter data. The Gaussian model's assumption of normal feature distributions is violated for word counts. Regularization and feature selection used by the multinomial and complement models likely eliminate sentiment-related words. Even after hyperparameter tuning, these models failed to match the custom implementation's accuracy.

Overall, this analysis highlights that adapting Naive Bayes modeling decisions to fit the patterns and quirks of social media text leads to substantially improved performance for tweet sentiment classification. This research provides a template for applying Naive Bayes to sentiment analysis on social media and textual data. The results showcase the potential of Naive Bayes as an accurate and efficient approach to text classification when configured appropriately for the dataset.

REFERENCES

[1] Rennie, J. D., Shih, L., Teevan, J., & Karger, D. R. (2003). Tackling the poor assumptions of naive Bayes text classifiers. In Proceedings of the Twentieth International Conference on Machine Learning (ICML-2003) (pp. 616–623). Washington DC.

KAUSTUV KARKI is a devoted individual currently pursuing a graduate degree in Computer Engineering Program from Institute of Engineering Thapathali Campus under Tribhuvan University He is keen and highly motivated towards the field of Artificial Intelligence and Machine Learning. He actively seeks out various sources to learn more about these fields, including books, research papers, online courses, and tutorials. He believes in continuous learning and keeping up with the latest advancements in AI and ML.
Overall, his dedication, curiosity, and proactive approach make him a promising individual in the field of Artificial Intelligence and Machine Learning.
Kaustuv’s curiosity serves as a driving force behind his pursuit of knowledge. He delves deep into complex concepts, asks thoughtful questions, and actively engages in discussions to gain a comprehensive understanding of AI and ML. This intellectual curiosity fuels his motivation and propels him to explore innovative solutions and approaches in the field.

NIKHIL PRADHAN is a dedicated and ambitious individual currently pursuing graduate studies in computer engineering from Institute of Engineering Thapathali Campus. With a strong passion for artificial intelligence (AI), he is actively engaged in expanding his knowledge and expertise in this rapidly evolving field. His academic pursuits provide him with a solid foundation in computer engineering and a deep understanding of programming languages and frameworks commonly used in AI applications. With his enthusiasm, academic background, and commitment to learning, He is well-positioned to make significant contributions to the AI industry.

APPENDIX A: FIGURES

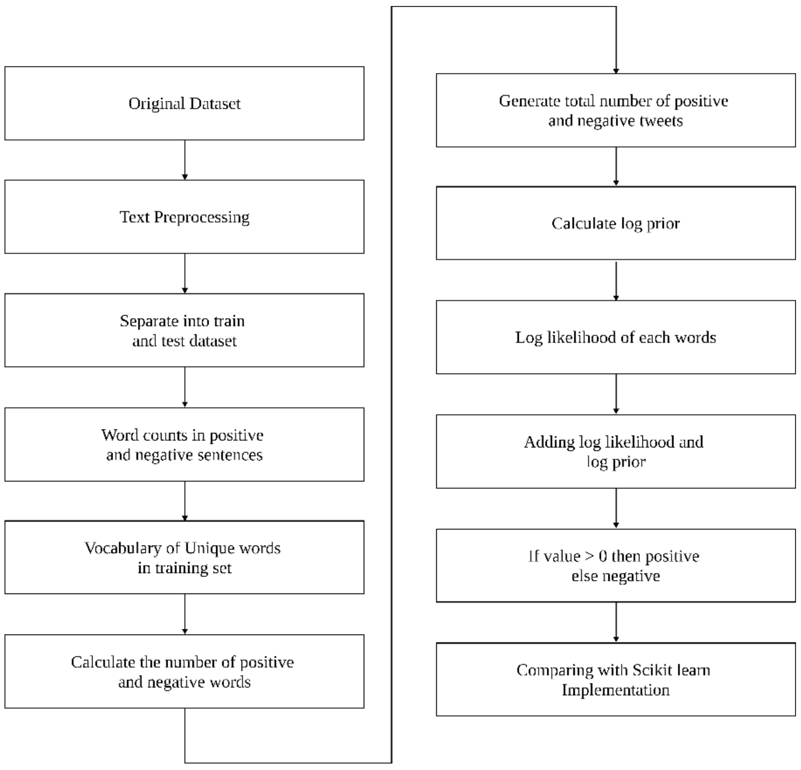
Original Dataset
Text Preprocessing
Separate into train and test dataset
Word counts in positive and negative sentences
Vocabulary of Unique words in training set
Calculate the number of positive and negative words
Generate total number of positive and negative tweets
Calculate log prior
Log likelihood of each words
Adding log likelihood and log prior
If value > 0 then positive else negative
Comparing with Scikit learn Implementation

Figure 1: System Block Diagram

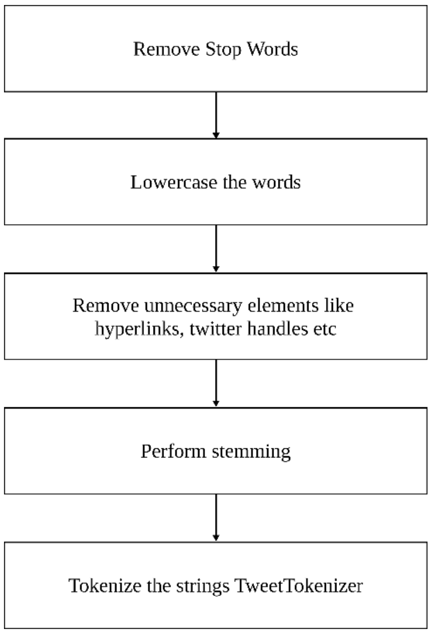
Remove Stop Words
Lowercase the words
Remove unnecessary elements like hyperlinks, twitter handles etc
Perform stemming
Tokenize the strings TweetTokenizer

Figure 2: Text Preprocessing

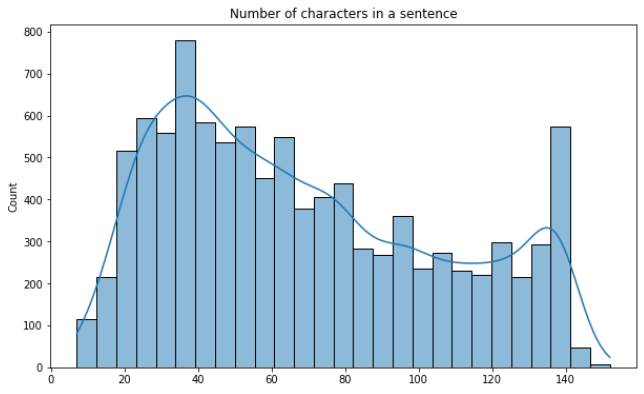

Figure 3: Number of characters in tweets

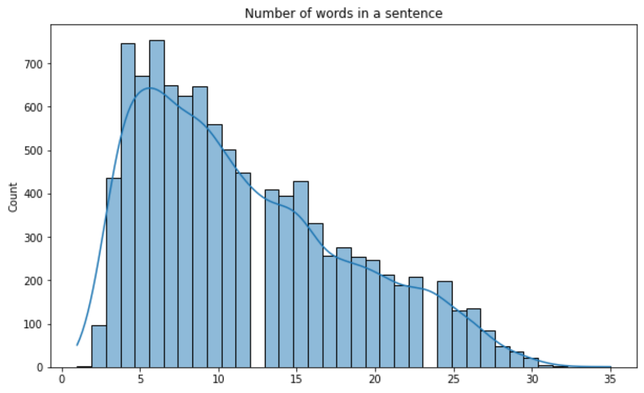

Figure 4: Number of words in tweets

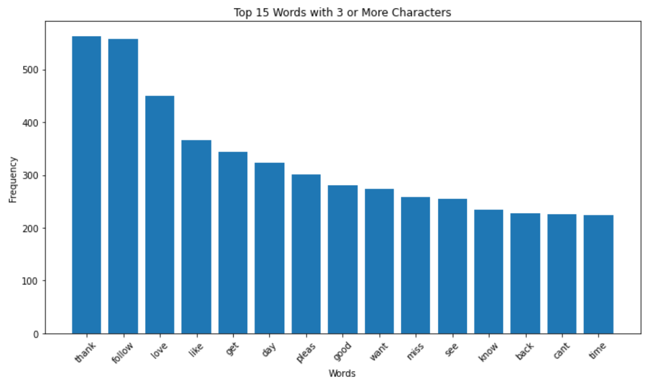

Figure 5: Top 15 words in the train dataset

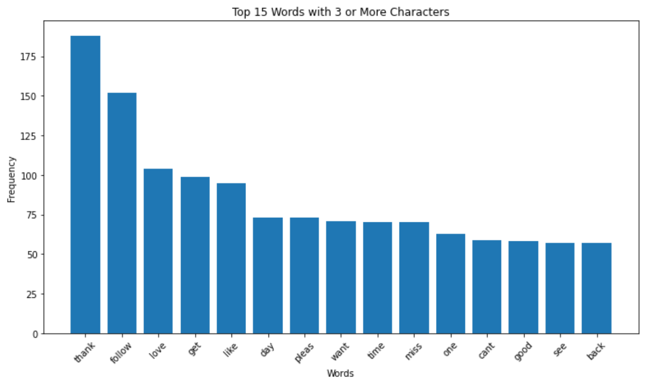

Figure 6: Top 15 words in the test dataset

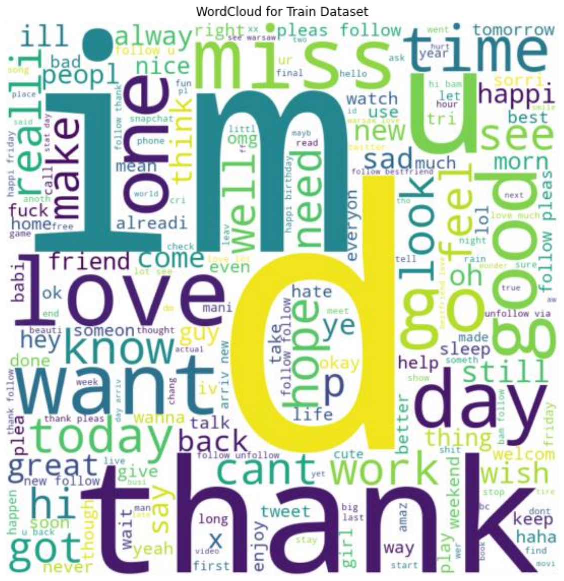

Figure 7: WordCloud of train dataset

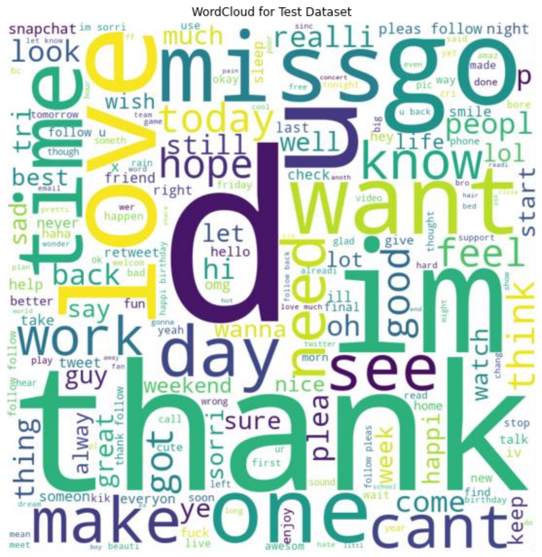

Figure 8: WordCloud of test Dataset

APPENDIX B: TABLES

TABLE I
CLASSIFICATION REPORT FOR GAUSSIAN NAÏVE BAYES

Class	Precision	Recall	F1-score
0	0.56	0.86	0.68
1	0.72	0.34	0.46
Accuracy-score		0.60	

TABLE II
CLASSIFICATION REPORT FOR MULTINOMIAL NAÏVE BAYES

Class	Precision	Recall	F1-score
0	0.72	0.77	074
1	0.76	0.71	0.74
Accuracy-score		0.74	

. TABLE III
CLASSIFICATION REPORT FOR COMPLEMENT NAÏVE BAYES

Class	Precision	Recall	F1-score
0	0.72	0.77	075
1	0.76	0.71	0.74
Accuracy-score		0.741	

APPENDIX C: CODE SNIPPETS

```
import nltk
from nltk.corpus import stopwords,
twitter_samples
import numpy as np
import pandas as pd
import string
from nltk.tokenize import TweetTokenizer
import matplotlib.pyplot as plt
import random
import re
import seaborn as sns
from nltk.stem import PorterStemmer


nltk.download('twitter samples')
nltk.download('stopwords')



positive_tweets =
twitter_samples.strings('positive_tweets.json')
negative_tweets =
twitter_samples.strings('negative_tweets.json')
len(positive_tweets)
test_pos = positive_tweets[4000:]
train_pos = positive_tweets[:4000]
test_neg = negative_tweets[4000:]
train_neg = negative_tweets[:4000]

train_x = train_pos + train_neg
test_x = test_pos + test_neg
train_y =
np.append(np.ones(len(train_pos)),np.zeros(len(t
rain_neg)))
test_y =
np.append(np.ones(len(test_pos)),np.zeros(len(te
st_neg)))

plt.bar(['Positive Tweets','Negative
Tweets'],[len(positive_tweets),len(negative_twee
ts)],color='orange')
plt.bar(['Training set','Test
set'],[len(train_x),len(test_x)])
#check postive tweets
print('Positive tweet example: ' +
positive_tweets[random.randint(0,5000)])
#check negative tweets
print('Negative tweet example: ' +
negative_tweets[random.randint(0,5000)])
THIS CSV FILE CONSISTS OF VALUES OF POSITIVE LOG
PROBABILITY, NEGATIVE LOG PROBABILITY AND
ASSOCIATED LABEL
features = pd.read_csv('bayes_features.csv')
features
colors = ['blue', 'orange']
sentiments = ['negative', 'positive']

# Create a scatter plot using seaborn
plt.figure(figsize=(8, 8))
sns.scatterplot(x='positive', y='negative',
hue='sentiment', data=features, palette=colors,
s=5, marker='*')

# Custom limits for this chart
plt.xlim(-250, 0)
plt.ylim(-250, 0)

plt.xlabel("Positive")
plt.ylabel("Negative")

# Add a legend
plt.legend(title='Sentiment', labels=sentiments)

plt.show()


def preprocess(tweet):
    #remove retweet text "RT"
    tweet2 = re.sub(r'^RT[\s]+','',tweet)

    #remove hyperlinks
    tweet2 =
re.sub(r'https?://[^\s\n\r]+','',tweet2)

    #remove hastag from front of the word
    tweet2 = re.sub(r'#','',tweet2)


    #tokenizer tweet
    tokenizer =
TweetTokenizer(preserve_case=False,strip_handles
=True,reduce_len=True)

    tweet_tokens = tokenizer.tokenize(tweet2)
    #remove stopwords and punctuations
    tweets_clean = []
    stopwords_english =
stopwords.words('english')
    for word in tweet_tokens:
        if (word not in stopwords_english and
            word not in string.punctuation):
            tweets_clean.append(word)


    #perform stemming
    stemmer = PorterStemmer()
    tweets_stem = []
    for word in tweets_clean:
        stem_word = stemmer.stem(word)
        tweets_stem.append(stem_word)


    return tweets_stem

def count_tweets(result,tweets,ys):
    for y,tweet in zip(ys,tweets):
        for word in preprocess(tweet):
            pair = (word,y)
            if pair in result:
                result[pair] +=1
            else:
                result[pair] = 1

    return result

freq = count_tweets({},train_x,train_y)
def naive_bayes_train(freq,train_x,train_y):

    loglikelihood={}
    #create vocab
    words = [key[0] for key in freq.keys()]
    vocab = set(words)
    V = len(vocab)

    #calculate number or positive and negative
words
    N_pos = N_neg = 0
    for pair in freq.keys():
        if pair[1] > 0:
            N_pos += freq[pair]
        else:
            N_neg += freq[pair]

    #calculating total number of tweets in train
    D = len(train_y)
    D_pos = sum(train_y)
    D_neg = D - D_pos

    logprior = np.log(D_pos) - np.log(D_neg)

    # For each word in the vocabulary...
    for word in vocab:
        # get the positive and negative
frequency of the word
        freq_pos = freq.get((word, 1), 0)
        freq_neg = freq.get((word, 0), 0)

        # calculate the probability that each
word is positive, and negative
        p_w_pos = (freq_pos + 1) / (N_pos + V)
        p_w_neg = (freq_neg + 1) / (N_neg + V)

        # calculate the log likelihood of the
word
        loglikelihood[word] = np.log(p_w_pos /
p_w_neg)

    return logprior, loglikelihood

def naive_bayes_predict(tweet, logprior,
loglikelihood):
    #preprocess the test tweet
    word_l = preprocess(tweet)

    #assign the value of logprior to p
    p = logprior

    #for all the words in processed tweet
calcualte loglikehood
    for word in word_l:
        if word in loglikelihood:
            p += loglikelihood[word]

    return p
```

```
logprior, loglikelihood =
naive_bayes_train(freq, train_x, train_y)
my_tweet = 'She smiled.'
p = naive_bayes_predict(my_tweet, logprior,
loglikelihood)
print('The expected output is', p)
my_tweet = 'He laughed.'
p = naive_bayes_predict(my_tweet, logprior,
loglikelihood)
print('The expected output is', p)
def test_naive_bayes(test_x, test_y, logprior,
loglikelihood,
naive_bayes_predict=naive_bayes_predict):
    #test the accuracy of prediction of the
model
    y_hats = []

    for tweet in test_x:
        if naive_bayes_predict(tweet, logprior,
loglikelihood) > 0:
            y_hat_i = 1
        else:
            y_hat_i = 0

        y_hats.append(y_hat_i)

    accuracy = np.mean(y_hats == test_y)

    return accuracy

print("Naive Bayes accuracy = %0.4f" %
      (test_naive_bayes(test_x, test_y,
logprior, loglikelihood)))

from sklearn.feature_extraction.text import
CountVectorizer
from sklearn.naive_bayes import ComplementNB
from sklearn.model_selection import
train_test_split
from sklearn.metrics import accuracy_score,
classification_report


# Combine the positive and negative tweets and
create labels
tweets = positive_tweets + negative_tweets
labels = [1] * len(positive_tweets) + [0] *
len(negative_tweets)

# Preprocess the tweets
processed_tweets = [' '.join(preprocess(tweet))
for tweet in tweets]

# Split the dataset into training and testing
sets
train_x, test_x, train_y, test_y =
train_test_split(processed_tweets, labels,
test_size=0.2, random_state=42)

# Create a CountVectorizer to convert text data
to numerical feature vectors
vectorizer = CountVectorizer()
train_vectors =
vectorizer.fit_transform(train_x)
test_vectors = vectorizer.transform(test_x)

# Create a Complement Naive Bayes classifier
cnb_classifier = ComplementNB()

# Train the classifier
cnb_classifier.fit(train_vectors, train_y)

# Make predictions on the test set
predictions =
cnb_classifier.predict(test_vectors)

# Evaluate the model
accuracy = accuracy_score(test_y, predictions)
print("Accuracy:", accuracy)

report = classification_report(test_y,
predictions)
print("Classification Report:\n", report)


from sklearn.model_selection import GridSearchCV

# Create a MultinomialNB instance
catnb_classifier = ComplementNB()

# Define the hyperparameter grid to search
```

```
param_grid = {'alpha': [0.1, 0.5, 1.0, 2.0, 5.0,
6.5, 8, 10], 'fit_prior' : [True, False]}

# Create a GridSearchCV object with cross-
validation
grid_search = GridSearchCV(catnb_classifier,
param_grid, cv=5)

# Perform the grid search on your training data
grid_search.fit(train_vectors, train_y)

# Get the best hyperparameters
best_alpha = grid_search.best_params_['alpha']

# Train the final classifier with the best
hyperparameters
final_classifier =
ComplementNB(alpha=best_alpha)
final_classifier.fit(train_vectors, train_y)

# Make predictions on the test set using the
final classifier
predictions =
final_classifier.predict(test_vectors)

# Evaluate the model
accuracy = accuracy_score(test_y, predictions)
print("Accuracy:", accuracy)

report = classification_report(test_y,
predictions)
print("Classification Report:\n", report)

grid_search.best_params_
```

```
from sklearn.feature_extraction.text import
CountVectorizer
from sklearn.naive_bayes import GaussianNB,
MultinomialNB, BernoulliNB
from sklearn.model_selection import
train_test_split
from sklearn.metrics import accuracy_score,
classification_report
```

```
# Combine the positive and negative tweets and
create labels
tweets = positive_tweets + negative_tweets
labels = [1] * len(positive_tweets) + [0] *
len(negative_tweets)

# Preprocess the tweets
processed_tweets = [' '.join(preprocess(tweet))
for tweet in tweets]

print(processed_tweets)

# Split the dataset into training and testing
sets
train_x, test_x, train_y, test_y =
train_test_split(processed_tweets, labels,
test_size=0.2, random_state=42)

# Create a CountVectorizer to convert text data
to numerical feature vectors
vectorizer = CountVectorizer()
train_vectors =
vectorizer.fit_transform(train_x).toarray()
test_vectors =
vectorizer.transform(test_x).toarray()

# Create a Categorical Naive Bayes classifier
catnb_classifier =  MultinomialNB()

# Train the classifier
catnb_classifier.fit(train_vectors, train_y)

# Make predictions on the test set
predictions =
catnb_classifier.predict(test_vectors)

# Evaluate the model
accuracy = accuracy_score(test_y, predictions)
print("Accuracy:", accuracy)

report = classification_report(test_y,
predictions)
print("Classification Report:\n", report)
```

```
from sklearn.model_selection import GridSearchCV

# Create a MultinomialNB instance
catnb_classifier = MultinomialNB()

# Define the hyperparameter grid to search
param_grid = {'alpha': [0.1, 0.5, 1.0, 2.0, 5.0,
6.5, 8, 10], 'fit_prior' : [True, False]}

# Create a GridSearchCV object with cross-
validation
grid_search = GridSearchCV(catnb_classifier,
param_grid, cv=5)

# Perform the grid search on your training data
grid_search.fit(train_vectors, train_y)

# Get the best hyperparameters
best_alpha = grid_search.best_params_['alpha']

# Train the final classifier with the best
hyperparameters
final_classifier =
MultinomialNB(alpha=best_alpha)
final_classifier.fit(train_vectors, train_y)

# Make predictions on the test set using the
final classifier
predictions =
final_classifier.predict(test_vectors)

# Evaluate the model
accuracy = accuracy_score(test_y, predictions)
print("Accuracy:", accuracy)

report = classification_report(test_y,
predictions)
print("Classification Report:\n", report)
grid_search.best_params_


from sklearn.feature_extraction.text import
CountVectorizer
from sklearn.naive_bayes import GaussianNB,
MultinomialNB, BernoulliNB
```

```
from sklearn.model_selection import
train_test_split
from sklearn.metrics import accuracy_score,
classification_report

# Combine the positive and negative tweets and
create labels
tweets = positive_tweets + negative_tweets
labels = [1] * len(positive_tweets) + [0] *
len(negative_tweets)

# Preprocess the tweets
processed_tweets = [' '.join(preprocess(tweet))
for tweet in tweets]

print(processed_tweets)

# Split the dataset into training and testing
sets
train_x, test_x, train_y, test_y =
train_test_split(processed_tweets, labels,
test_size=0.2, random_state=42)

# Create a CountVectorizer to convert text data
to numerical feature vectors
vectorizer = CountVectorizer()
train_vectors =
vectorizer.fit_transform(train_x).toarray()
test_vectors =
vectorizer.transform(test_x).toarray()

# Create a Categorical Naive Bayes classifier
catnb_classifier =  BernoulliNB()

# Train the classifier
catnb_classifier.fit(train_vectors, train_y)

# Make predictions on the test set
predictions =
catnb_classifier.predict(test_vectors)

# Evaluate the model
accuracy = accuracy_score(test_y, predictions)
print("Accuracy:", accuracy)
```

```
report = classification_report(test_y,
predictions)
print("Classification Report:\n", report)
from sklearn.feature_extraction.text import
CountVectorizer
from sklearn.naive_bayes import GaussianNB,
MultinomialNB, BernoulliNB
from sklearn.model_selection import
train_test_split
from sklearn.metrics import accuracy_score,
classification_report

# Combine the positive and negative tweets and
create labels
tweets = positive_tweets + negative_tweets
labels = [1] * len(positive_tweets) + [0] *
len(negative_tweets)

# Preprocess the tweets
processed_tweets = [' '.join(preprocess(tweet))
for tweet in tweets]

print(processed_tweets)

# Split the dataset into training and testing
sets
train_x, test_x, train_y, test_y =
train_test_split(processed_tweets, labels,
test_size=0.2, random_state=42)

# Create a CountVectorizer to convert text data
to numerical feature vectors
vectorizer = CountVectorizer()
train_vectors =
vectorizer.fit_transform(train_x).toarray()
test_vectors =
vectorizer.transform(test_x).toarray()

# Create a Categorical Naive Bayes classifier
catnb_classifier =  GaussianNB()

# Train the classifier
catnb_classifier.fit(train_vectors, train_y)

# Make predictions on the test set
predictions =
catnb_classifier.predict(test_vectors)

# Evaluate the model
accuracy = accuracy_score(test_y, predictions)
print("Accuracy:", accuracy)

report = classification_report(test_y,
predictions)
print("Classification Report:\n", report)

import re

def
remove_non_alphanumeric_except_space(input_list)
:
    result_list = []
    for item in input_list:
        # Using regex to remove non-alphanumeric
characters except space
        cleaned_item = re.sub(r'[^a-zA-Z0-9\s]',
'', item)
        result_list.append(cleaned_item)
    return result_list

output_list =
remove_non_alphanumeric_except_space(train_x)
output_list

import matplotlib.pyplot as plt
from wordcloud import WordCloud
from collections import Counter

def generate_word_cloud(sentences, name):
    text = " ".join(sentences)
    wordcloud = WordCloud(width=800, height=800,
background_color="white").generate(text)
    plt.figure(figsize=(10, 10))
    plt.title(name)
    plt.imshow(wordcloud,
interpolation="bilinear")
    plt.axis("off")
    plt.show()


def generate_bar_graph(sentences):
    text = " ".join(sentences)
    words = re.findall(r'\b\w+\b',
text.lower())  # Extract all words
    words_filtered = [word for word in words if
len(word) >= 3]  # Filter words with 3 or more
characters
    word_freq = Counter(words_filtered)  # Count
word frequencies

    top_words = word_freq.most_common(15)  # Get
top 10 words
    top_words, word_counts = zip(*top_words)

    plt.figure(figsize=(10, 6))
    plt.bar(top_words, word_counts)

    plt.xlabel("Words")
    plt.ylabel("Frequency")
    plt.title("Top 15 Words with 3 or More
Characters")
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()



generate_word_cloud(output_list, "WordCloud for
Train Dataset")
generate_bar_graph(output_list)

output_list_x =
remove_non_alphanumeric_except_space(test_x)
output_list_x
generate_word_cloud(output_list_x, "WordCloud
for Test Dataset")
generate_bar_graph(output_list_x)
total_sentences = positive_tweets +
negative_tweets
sentence_lengths = [len(sentence) for sentence
in total_sentences]

import seaborn as sns

data_input_sentence_character_length =
pd.Series(sentence_lengths)
```

```
# Create a histogram with a density curve
plt.figure(figsize=(10, 6))
plt.title("Number of characters in a sentence")
sns.histplot(data=data_input_sentence_character_
length, kde=True)
sentence_word_counts = [len(sentence.split())
for sentence in total_sentences]
data_input_word_length =
pd.Series(sentence_word_counts)

# Create a histogram with a density curve
plt.figure(figsize=(10, 6))
plt.title("Number of words in a sentence")
sns.histplot(data=data_input_word_length,
kde=True)
```