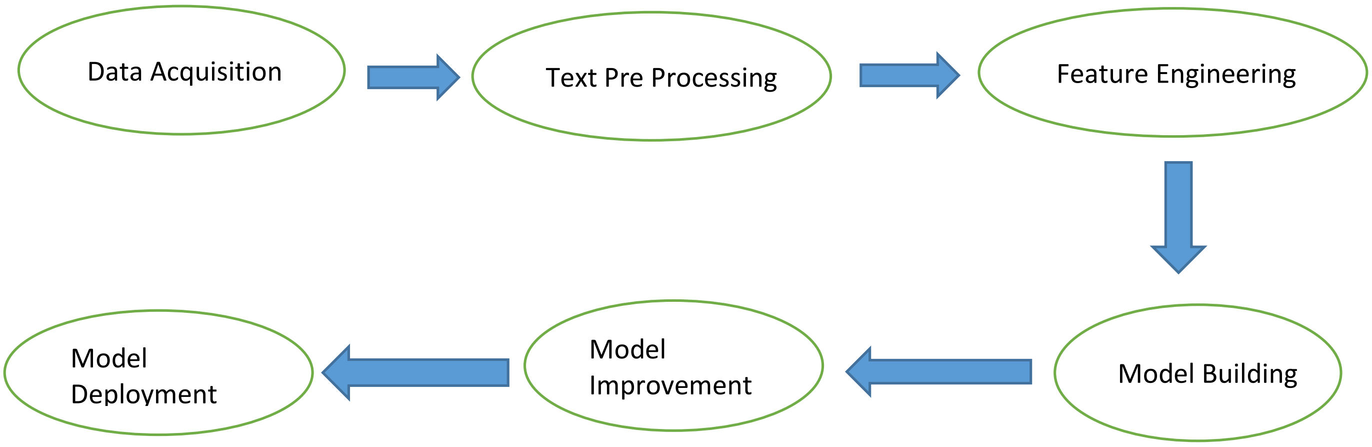
NAME: **SUSANTA DHURUA,**
LOCATION: **WEST BENGAL**
EMAIL: **sd202524@gmail.com**
MOB: **9078317624**

SMS- SPAM CLASSIFIER

Abstract: The project revolves around Natural Language Processing (NLP), where we are building an email classifier model. This model aims to categorize incoming messages as either spam or not spam (ham). For the dataset, we have collected it from a public source and utilized it to train our model. Since I am not affiliated with any organization, I opted to work with publicly available data.

In this project, I thoroughly explain each step of the NLP process, following the NLP pipeline.

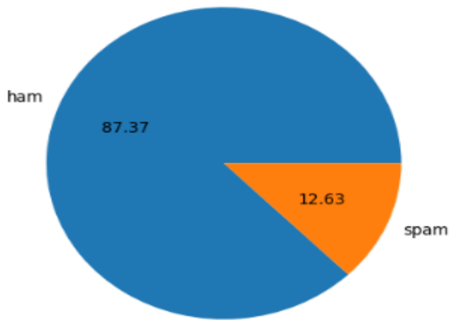


Introduction: Our data was collected from the public dataset Kaggle. After loading the dataset, we observed that its shape is 5572x5, which translates to 5572 rows and 5 columns. To ensure data quality, we conducted data cleaning and identified an unknown column alongside our main working columns. Consequently, we dropped the unknown column and renamed our working columns as "target" and "text" respectively.

Next, we employed a label encoder to convert the "target" column into binary values, representing "ham" as 0 and "spam" as 1. Throughout this process, we carefully checked for missing values, and fortunately, our dataset was devoid of any missing data.

However, we did find 403 duplicate values, but we resolved this issue by retaining only the first occurrence of each duplicate text and discarding the rest. After successfully implementing these steps, our data's new shape became 5169x2, signifying 5169 rows and 2 columns.

Now, during the Exploratory Data Analysis (EDA), our initial step involved Examining the percentage of texts classified as spam and ham.



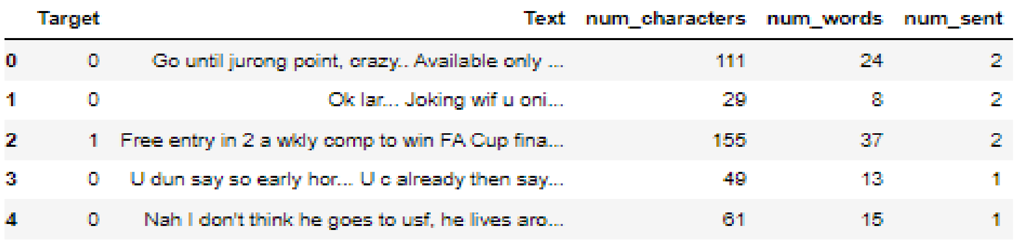
We represented this distribution using a pie chart, which revealed that 87.37% of texts were classified as ham (not spam) and 12.63% were classified as spam. This pie chart also led us to the conclusion that our data is imbalanced.

Due to the data imbalance, we decided to construct some custom features that would aid in the model-building process. These features include:

- The number of characters in each text.
- The number of words in each text.
- The number of sentences in each text.

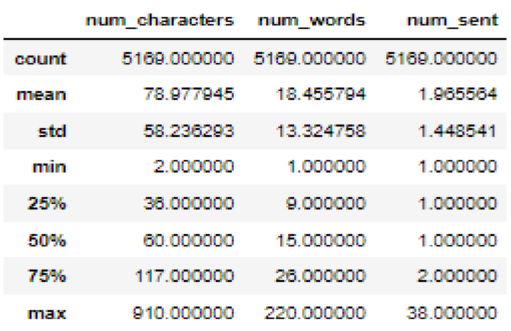
To develop the first feature, we utilized the "Len" function in Python. For the remaining two features, we relied on the NLTK library. Specifically, we imported the "word tokenize" and "sent tokenize" functions from NLTK.

After successfully creating these features, we appended them to our main dataset. Here is an overview of our dataset, now enriched with these additional features.



	Target	Text	num_characters	num_words	num_sent
0	0	Go until jurong point, crazy.. Available only ...	111	24	2
1	0	Ok lar... Joking wif u oni...	29	8	2
2	1	Free entry in 2 a wkly comp to win FA Cup fina...	155	37	2
3	0	U dun say so early hor... U c already then say...	49	13	1
4	0	Nah I don't think he goes to usf, he lives aro...	61	15	1

After appending the features, we proceeded to use the "describe" function to analyse the "num_characters," "num_words," and "num_sent" data. Here is an overview of the "describe" function's output:



	num_characters	num_words	num_sent
count	5169.000000	5169.000000	5169.000000
mean	78.977945	18.455794	1.965564
std	58.236293	13.324758	1.448541
min	2.000000	1.000000	1.000000
25%	36.000000	9.000000	1.000000
50%	60.000000	15.000000	1.000000
75%	117.000000	26.000000	2.000000
max	910.000000	220.000000	38.000000

Interpretation:

num characters:

- The count (number of data points) for this feature is 4516.
- The mean value is approximately 70.46, indicating that, on average, the texts contain about 70 characters.
- The standard deviation is approximately 56.36, indicating that the values tend to vary from the mean by around 56 characters.
- The minimum value is 2, suggesting that the shortest text in the dataset consists of only 2 characters.
- The 25th percentile value is 34, meaning that 25% of the texts have 34 characters or fewer.
- The median (50th percentile) value is 52, implying that half of the texts have 52 characters or less.
- The 75th percentile value is 90, indicating that 75% of the texts have 90 characters or fewer.
- The maximum value is 910, revealing that the longest text in the dataset contains 910 characters.

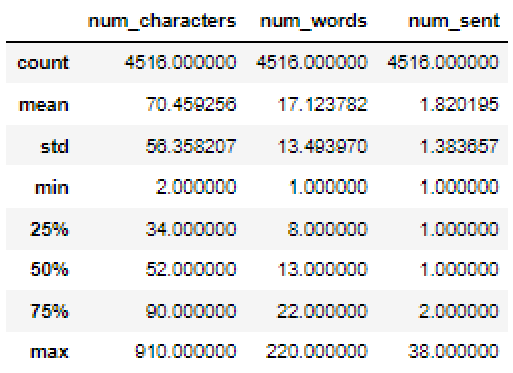
num words:

- The count for this feature is also 4516, meaning there are 4516 data points.
- The mean value is approximately 17.12, indicating that, on average, the texts have about 17 words.
- The standard deviation is approximately 13.49, suggesting that the number of words tends to vary from the mean by around 13 words.
- The minimum value is 1, meaning that the shortest text in the dataset consists of only 1 word.
- The 25th percentile value is 8, indicating that 25% of the texts have 8 words or fewer.
- The median (50th percentile) value is 13, implying that half of the texts have 13 words or less.
- The 75th percentile value is 22, indicating that 75% of the texts have 22 words or fewer.
- The maximum value is 220, revealing that the longest text in the dataset contains 220 words.

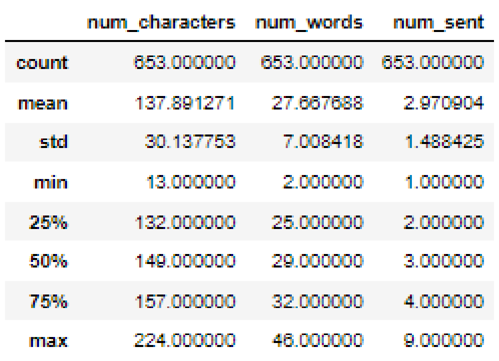
num sent:

- The count for this feature is also 4516, meaning there are 4516 data points.
- The mean value is approximately 1.82, indicating that, on average, the texts have about 1.82 sentences.
- The standard deviation is approximately 1.38, suggesting that the number of sentences tends to vary from the mean by around 1.38 sentences.
- The minimum value is 1, meaning that the shortest text in the dataset consists of only 1 sentence.
- The 25th percentile value is 1, indicating that 25% of the texts have only 1 sentence.
- The median (50th percentile) value is also 1, implying that half of the texts have 1 sentence.
- The 75th percentile value is 2, indicating that 75% of the texts have 2 sentences or fewer.
- The maximum value is 38, revealing that the longest text in the dataset contains 38 sentences.

The above describe function is for both the classes, what if we could see the separate describe function for both of the classes separately, Ham and Spam



	num_characters	num_words	num_sent
count	4516.000000	4516.000000	4516.000000
mean	70.459256	17.123782	1.820195
std	56.358207	13.493970	1.383657
min	2.000000	1.000000	1.000000
25%	34.000000	8.000000	1.000000
50%	52.000000	13.000000	1.000000
75%	90.000000	22.000000	2.000000
max	910.000000	220.000000	38.000000



	num_characters	num_words	num_sent
count	653.000000	653.000000	653.000000
mean	137.891271	27.667688	2.970904
std	30.137753	7.008418	1.488425
min	13.000000	2.000000	1.000000
25%	132.000000	25.000000	2.000000
50%	149.000000	29.000000	3.000000
75%	157.000000	32.000000	4.000000
max	224.000000	46.000000	9.000000

Ham Describe Function

Spam Describe Function

HAM Message **Interpretation**: SPAM Message

Summary for "num characters":

Count: 4516
Mean: ~70.46 characters
Std. Deviation: ~56.36 characters
Min: 2 characters
25th percentile: 34 characters or fewer
Median: 52 characters or less
75th percentile: 90 characters or fewer
Max: 910 characters

Summary for "num words":

Count: 4516
Mean: ~17.12 words
Std. Deviation: ~13.49 words
Min: 1 word
25th percentile: 8 words or fewer
Median: 13 words or less
75th percentile: 22 words or fewer
Max: 220 words

Summary for "num sent":

Count: 4516
Mean: ~1.82 sentences
Std. Deviation: ~1.38 sentences
Min: 1 sentence
25th percentile: 1 sentence
Median: 1 sentence

Summary for "num characters":

Count: 653
Mean: ~137.89 characters
Std. Deviation: ~30.14 characters
Min: 13 characters
25th percentile: 132 characters or fewer
Median: 149 characters or less
75th percentile: 157 characters or fewer
Max: 224 characters

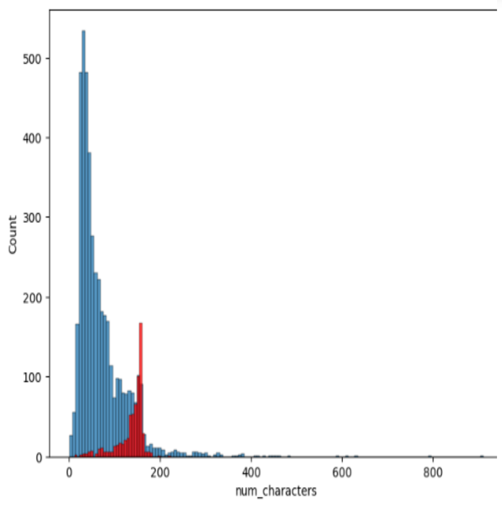
Summary for "num words":

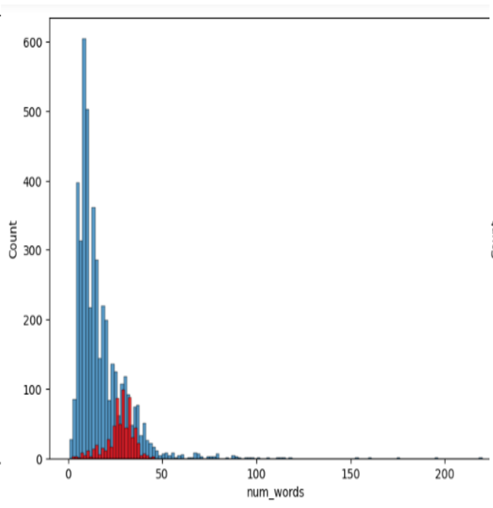
Count: 653
Mean: ~27.67 words
Std. Deviation: ~7.01 words
Min: 2 words
25th percentile: 25 words or fewer
Median: 29 words or less
75th percentile: 32 words or fewer
Max: 46 words

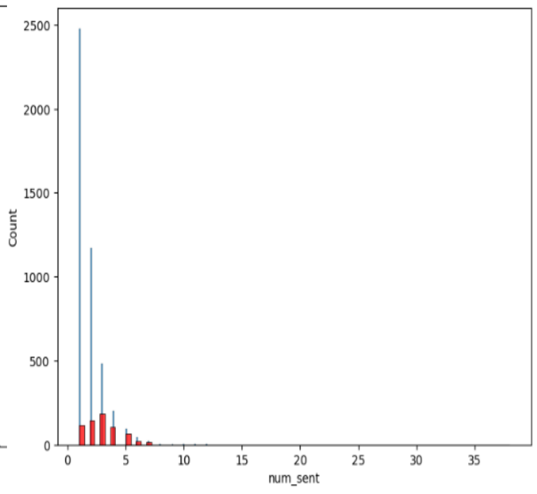
Summary for "num sent":

Count: 653
Mean: ~2.97 sentences
Std. Deviation: ~1.49 sentences
Min: 1 sentence
25th percentile: 2 sentences
Median: 3 sentences

Some of the visualizations, to describe the 3 features for both the class.

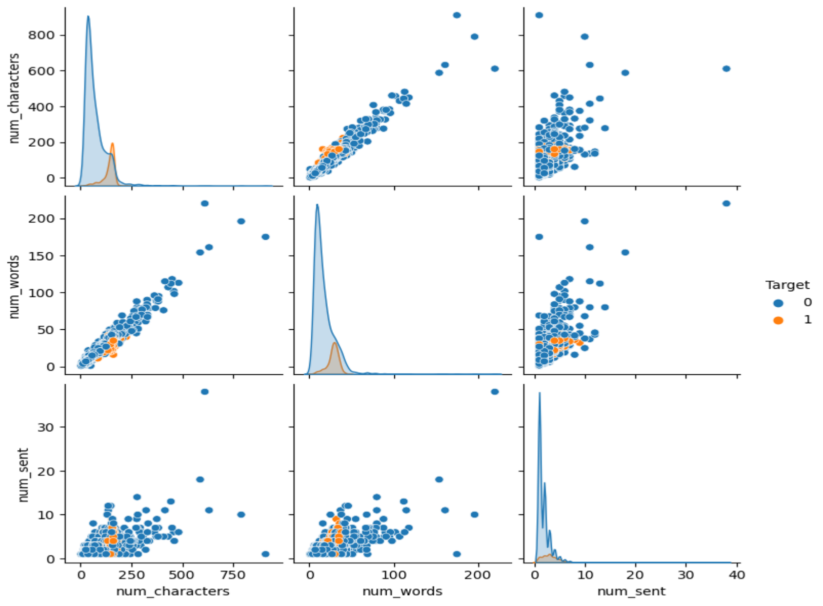






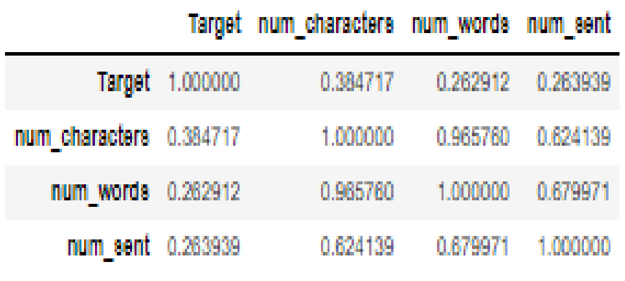
We have visualized that spam texts use a lower number of characters, words, and sentences compared to non-spam texts.

We have constructed a pair plot to observe the patterns of feature behaviour with each other.

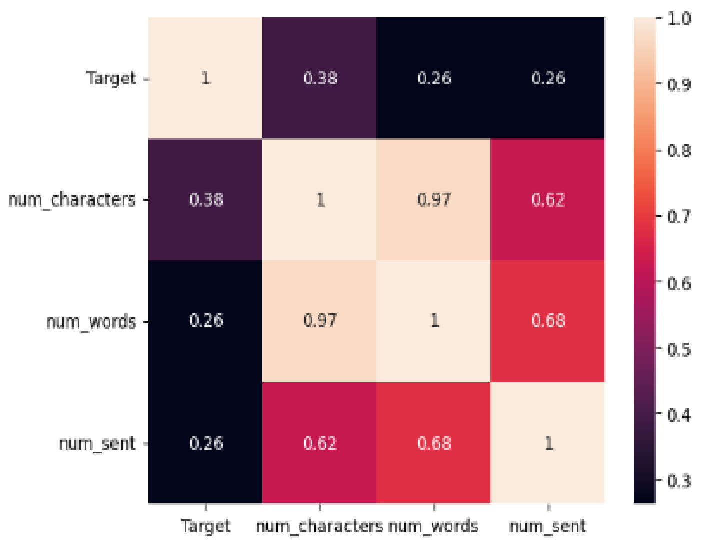


Upon examining the pair plot, we noticed that the relationship between the number of characters and the number of sentences is not linear, and the same holds true for the number of words. Furthermore, we observed the presence of outliers in the data.

To understand the relationship between the independent variable and the dependent variable, we examined their correlation. The correlation ranges from -1 to +1, where values close to -1 indicate a negative correlation, and values close to +1 indicate a positive correlation. To facilitate a clear understanding of these relationships, I have presented the correlation table as a heat map.



	Target	num_characters	num_words	num_sent
Target	1.000000	0.384717	0.262912	0.263939
num_characters	0.384717	1.000000	0.965760	0.624139
num_words	0.262912	0.965760	1.000000	0.679971
num_sent	0.263939	0.624139	0.679971	1.000000



Interpretation: The "Target" variable shows a moderate Positive correlation with"num characters" (0.384717), "num words" (0.262912), and "num sent" (0.263939).

Additionally, "num characters" has a moderate positive Correlation with "num words" (0.965760) and a relatively Weaker positive correlation with "num sent" (0.624139).

Furthermore, "num words" demonstrates a strong Positive correlation with "num sent" (0.679971).

TEXT PREPROCESSING: In text pre-processing, we applied several transformations to our text. The transformations we performed include: # lowercasing, #tokenization, #removing special characters, #removing stop words, #removing punctuation, #stemming.

Here is a brief summary of the transformations applied to the text:

# lowercasing: This transformation converts all the text into lowercase, ensuring uniformity throughout.

# Tokenization: In the context of NLP, tokenization involves breaking down a text or sentence into individual units called tokens. These tokens can be words, phrases, or even characters, depending on the level of analysis required.

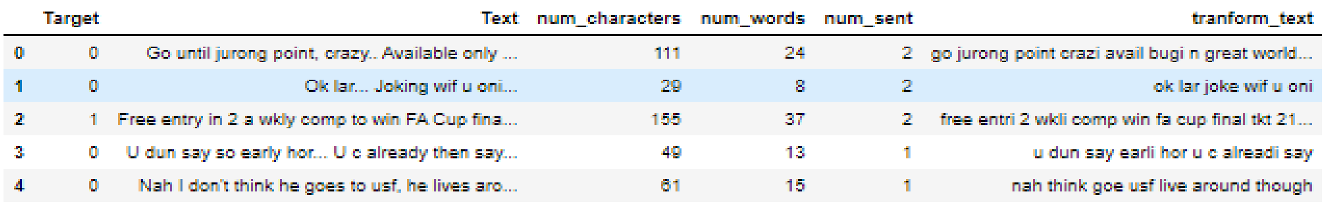
# removing special characters: Special characters encompass non-alphanumeric symbols, punctuation marks, emoji's, and other non-standard characters.

# removing stop words: Stop words are common words frequently found in the language that do not contribute significantly to the meaning of the text. These include articles (e.g., "a," "an," "the"), prepositions (e.g., "in," "on," "at"), conjunctions (e.g., "and," "but," "or"), and other high-frequency words.

# removing punctuation: The removal of punctuation symbols such as "!"#$ %&'()*+,-./:;<=>?@[]^_`{|} ~" from the text.

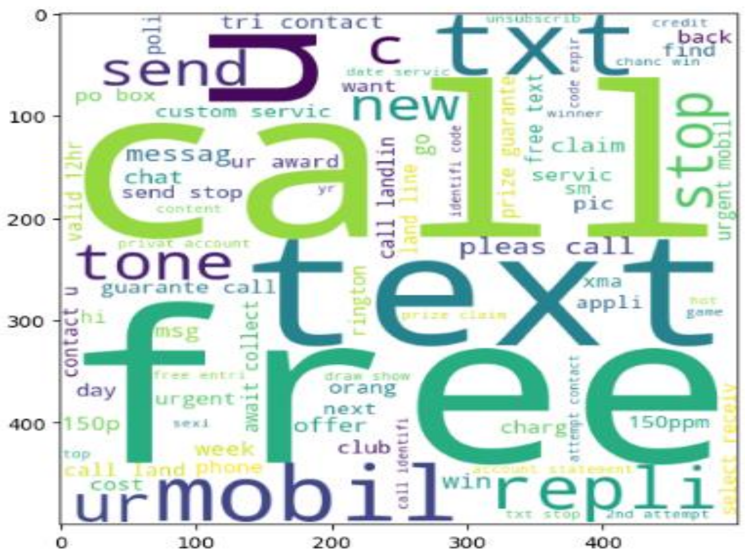
# stemming: This process brings words to their root form, facilitating the consolidation of variations of the same word.

After successfully completing the text processing, we created a new column named "Transformed Text" in which the text underwent the aforementioned transformations of text processing. Here is a sample of the result

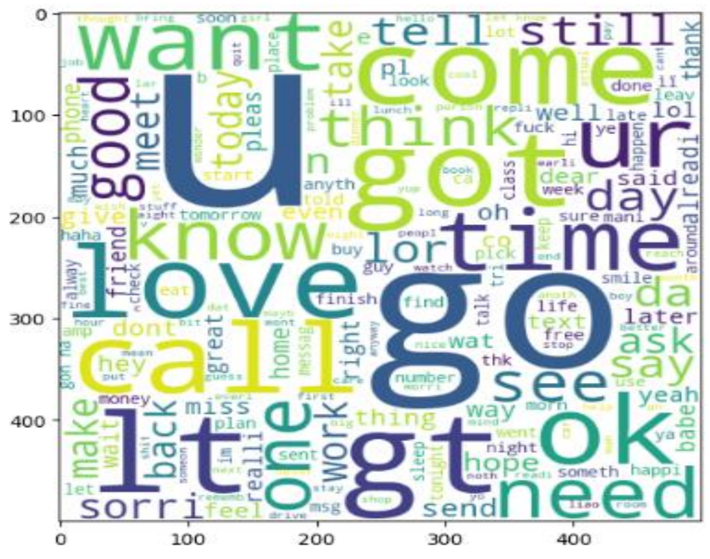


	Target	Text	num_characters	num_words	num_sent	tranform_text
0	0	Go until jurong point, crazy.. Available only ...	111	24	2	go jurong point crazi avail bugi n great world...
1	0	Ok lar... Joking wif u oni...	29	8	2	ok lar joke wif u oni
2	1	Free entry in 2 a wkly comp to win FA Cup fina...	155	37	2	free entri 2 wkli comp win fa cup final tkt 21...
3	0	U dun say so early hor... U c already then say...	49	13	1	u dun say earli hor u c alreadi say
4	0	Nah I don't think he goes to usf, he lives aro...	61	15	1	nah think goe usf live around though

A sudden query arises about the frequent words present in both spam and non-spam messages. To resolve this doubt, I utilized the word cloud function, which allows us to visually identify the most common words appearing in both classes.

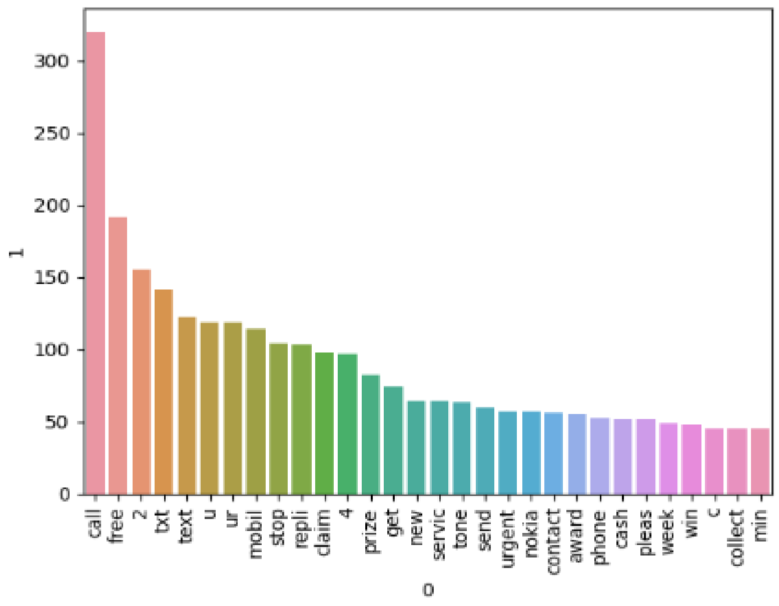


Spam messages

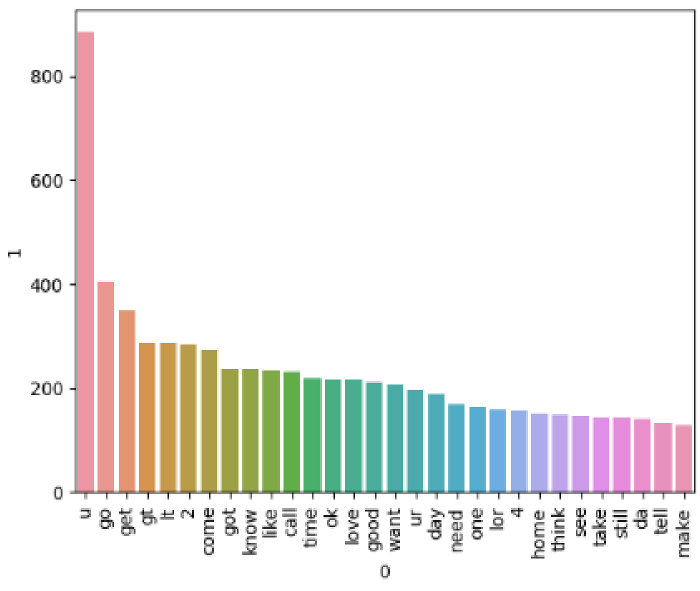


Not Spam messages

Based on the previous word cloud, we observed the most frequent words present in both types of messages. Now, we are interested in examining the 30 most common words and their respective frequencies.



Spam messages



not spam messages

FEATURE ENGINEERING: In simple terms, when dealing with textual data, it falls under the domain of natural language processing (NLP). The process of feature engineering in NLP involves converting the text into numerical representations so that machine learning algorithms, such as Naïve Bayes, logistic regression, and SVM, can be applied.

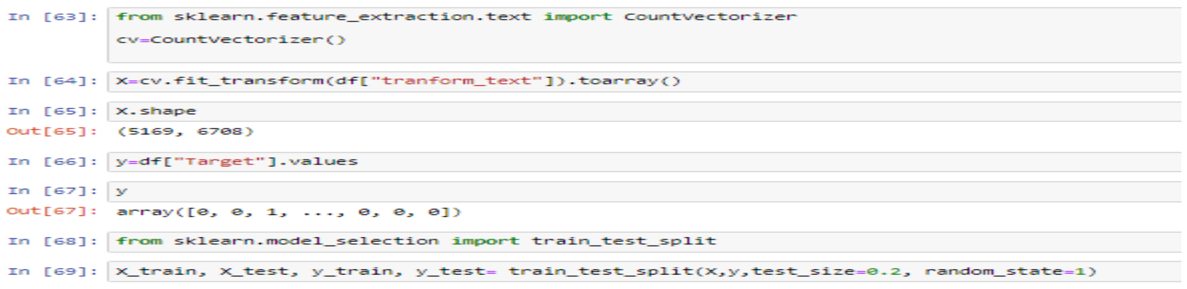
For feature engineering, we have three popular techniques: Bag of Words, TFIDF (Term Frequency – Inverse Document Frequency), and word2vec. For this comparison, we will focus on Bag of Words and TFIDF.

Before delving into feature engineering, let's recap the key characteristics of Bag of Words and TFIDF.

Bag of Words is a popular NLP technique that converts Text into numerical Representations. It creates a Vocabulary Of unique words from the corpus, Representing each document as a frequency Vector. It treats words independently, disregarding context, Making it useful for text classification tasks like Sentiment analysis and Spam detection.

TFIDF is a technique to convert text into numerical representations, Building on Bag of Words. It considers both word frequency in a Document (Term Frequency) and importance across the Corpus (Inverse Document Frequency). High TFIDF scores for rare Words emphasize their significance in distinguishing documents, While common words get lower scores as they are less informative. TFIDF finds extensive use in information retrieval, document Clustering, and text summarization.

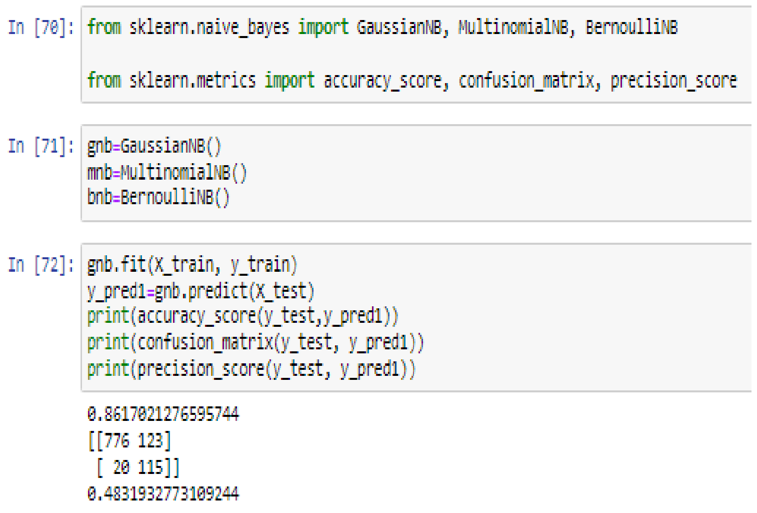
First, for feature engineering, we begin with the bag of words approach. To implement bag of words, we import the Count-Vectorizer, which will convert our text into numerical representations using the underlying logic of the bag of words technique.



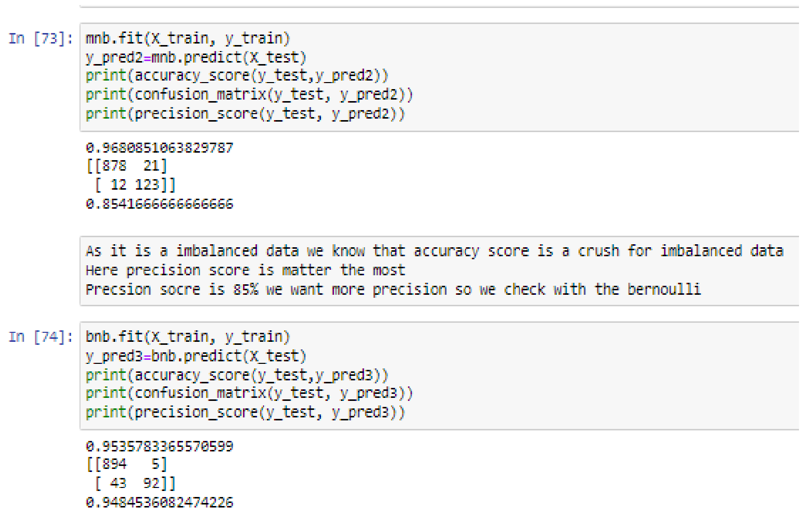
```
In [63]: from sklearn.feature_extraction.text import CountVectorizer
         cv=CountVectorizer()

In [64]: X=cv.fit_transform(df["tranform_text"]).toarray()

In [65]: X.shape
Out[65]: (5169, 6708)

In [66]: y=df["Target"].values

In [67]: y
Out[67]: array([0, 0, 1, ..., 0, 0, 0])

In [68]: from sklearn.model_selection import train_test_split

In [69]: X_train, X_test, y_train, y_test= train_test_split(X,y,test_size=0.2, random_state=1)
```

MODEL BUILDING: To build the model, we used the machine learning algorithm, namely Naïve Bayes, which is particularly effective for textual data. Before delving into model building, let's briefly review Naïve Bayes.

Naïve Bayes is characterized by its simplicity and efficiency in text classification. It assumes feature independence, leading to computational efficiency. Despite this assumption, it performs well with limited training data and high-dimensional text data. It exhibits robustness against irrelevant features, making it suitable for tasks like spam detection and sentiment analysis. While its assumption may not always hold, Naïve Bayes remains a powerful choice for many text classification problems.



```
In [70]: from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB

         from sklearn.metrics import accuracy_score, confusion_matrix, precision_score

In [71]: gnb=GaussianNB()
         mnb=MultinomialNB()
         bnb=BernoulliNB()

In [72]: gnb.fit(X_train, y_train)
         y_pred1=gnb.predict(X_test)
         print(accuracy_score(y_test,y_pred1))
         print(confusion_matrix(y_test, y_pred1))
         print(precision_score(y_test, y_pred1))

         0.8617021276595744
         [[776 123]
          [ 20 115]]
         0.4831932773109244
```



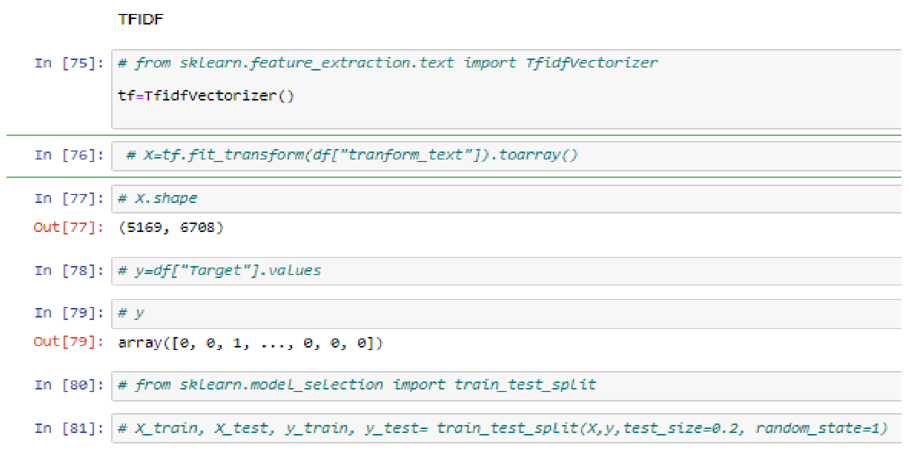
```
In [73]: mnb.fit(X_train, y_train)
         y_pred2=mnb.predict(X_test)
         print(accuracy_score(y_test,y_pred2))
         print(confusion_matrix(y_test, y_pred2))
         print(precision_score(y_test, y_pred2))

         0.9680851063829787
         [[878  21]
          [ 12 123]]
         0.8541666666666666

         As it is a imbalanced data we know that accuracy score is a crush for imbalanced data
         Here precision score is matter the most
         Precsion socre is 85% we want more precision so we check with the bernoulli

In [74]: bnb.fit(X_train, y_train)
         y_pred3=bnb.predict(X_test)
         print(accuracy_score(y_test,y_pred3))
         print(confusion_matrix(y_test, y_pred3))
         print(precision_score(y_test, y_pred3))

         0.9535783365570599
         [[894   5]
          [ 43  92]]
         0.9484536082474226
```

Interpretation: Gaussian naïve Bayes→ as the data is imbalanced, we are aware that accuracy score can be Misleading for imbalanced datasets. In this case, precision score holds the utmost importance. However, the Precision score is very low and is not performing up to the desired standards.

Multinomial Naïve Bayes→ As it is an imbalanced dataset, we are aware that the accuracy score may not be a reliable metric in such cases. In this context, the precision score becomes the most crucial evaluation metric. Currently, the precision score is at 85%. However, we aim for higher precision, so we explore the performance of the Bernoulli model to achieve this goal.

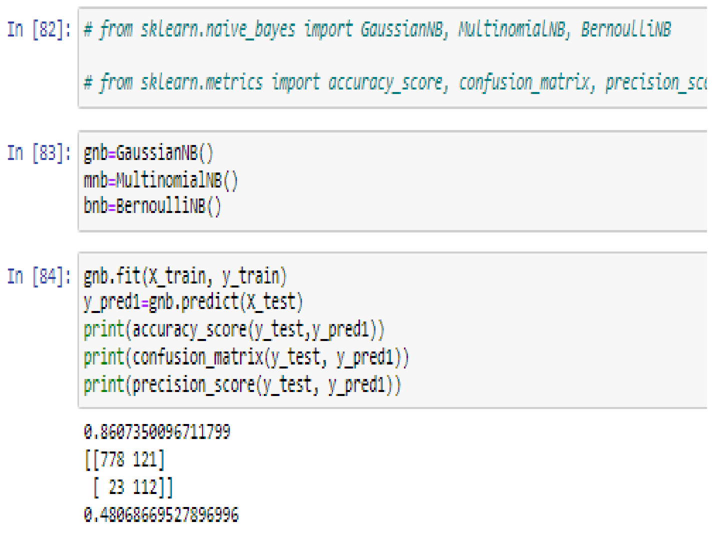
Bernoulli Naïve Bayes→ Precision score takes precedence in our evaluation. Currently, the precision score is 94%, surpassing the performance of the Gaussian and Multinomial models.

This precision score was obtained using Count Vectorizer, which applies the bag of words approach. Now, let's explore the performance of the TFIDF (Term Frequency-Inverse Document Frequency) technique.

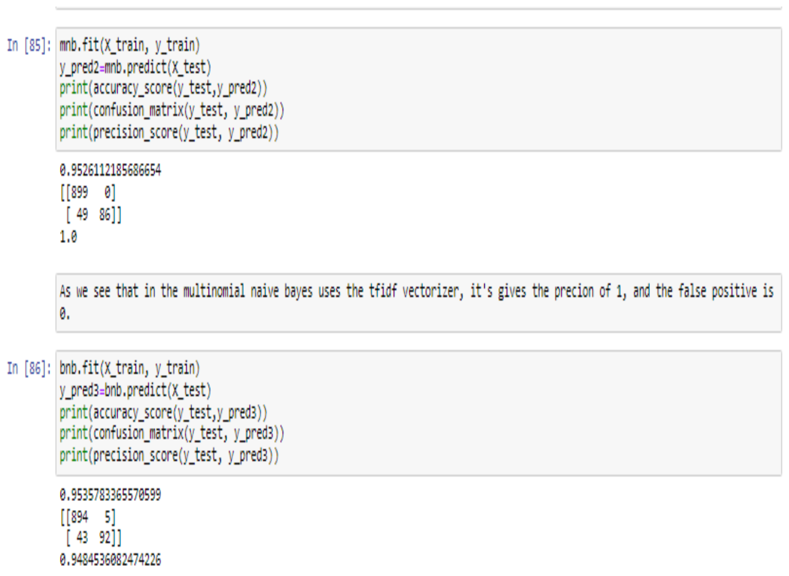


```
         TFIDF

In [75]: # from sklearn.feature_extraction.text import TfidfVectorizer
         tf=TfidfVectorizer()

In [76]: # X=tf.fit_transform(df["tranform_text"]).toarray()

In [77]: # X.shape
Out[77]: (5169, 6708)

In [78]: # y=df["Target"].values

In [79]: # y
Out[79]: array([0, 0, 1, ..., 0, 0, 0])

In [80]: # from sklearn.model_selection import train_test_split

In [81]: # X_train, X_test, y_train, y_test= train_test_split(X,y,test_size=0.2, random_state=1)
```

Feature engineering by TFIDF.



```
In [82]: # from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB

         # from sklearn.metrics import accuracy_score, confusion_matrix, precision_sc

In [83]: gnb=GaussianNB()
         mnb=MultinomialNB()
         bnb=BernoulliNB()

In [84]: gnb.fit(X_train, y_train)
         y_pred1=gnb.predict(X_test)
         print(accuracy_score(y_test,y_pred1))
         print(confusion_matrix(y_test, y_pred1))
         print(precision_score(y_test, y_pred1))

         0.8607350096711799
         [[778 121]
          [ 23 112]]
         0.48068669527896996
```



```
In [85]: mnb.fit(X_train, y_train)
         y_pred2=mnb.predict(X_test)
         print(accuracy_score(y_test,y_pred2))
         print(confusion_matrix(y_test, y_pred2))
         print(precision_score(y_test, y_pred2))

         0.9526112185686654
         [[899   0]
          [ 49  86]]
         1.0
```

As we see that in the multinomial naive bayes uses the tfidf vectorizer, it's gives the precion of 1, and the false positive is 0.

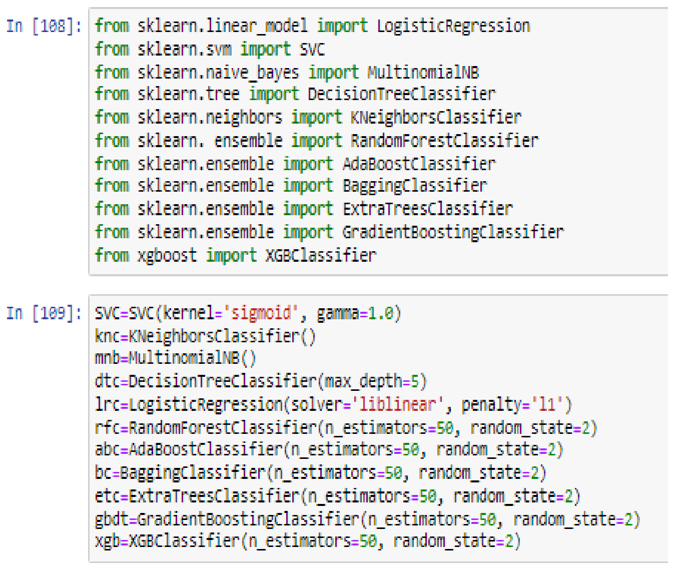
```
In [86]: bnb.fit(X_train, y_train)
         y_pred3=bnb.predict(X_test)
         print(accuracy_score(y_test,y_pred3))
         print(confusion_matrix(y_test, y_pred3))
         print(precision_score(y_test, y_pred3))

         0.9535783365570599
         [[894   5]
          [ 43  92]]
         0.9484536082474226
```

Interpretation: Gaussian Naïve Bayes→ as the data is imbalanced, we are aware that the accuracy score can be misleading for imbalanced datasets. In this case, the precision score becomes the most critical metric. Unfortunately, the precision score is currently very low and is not meeting the desired standards. We need to address this issue to improve the performance of our model.

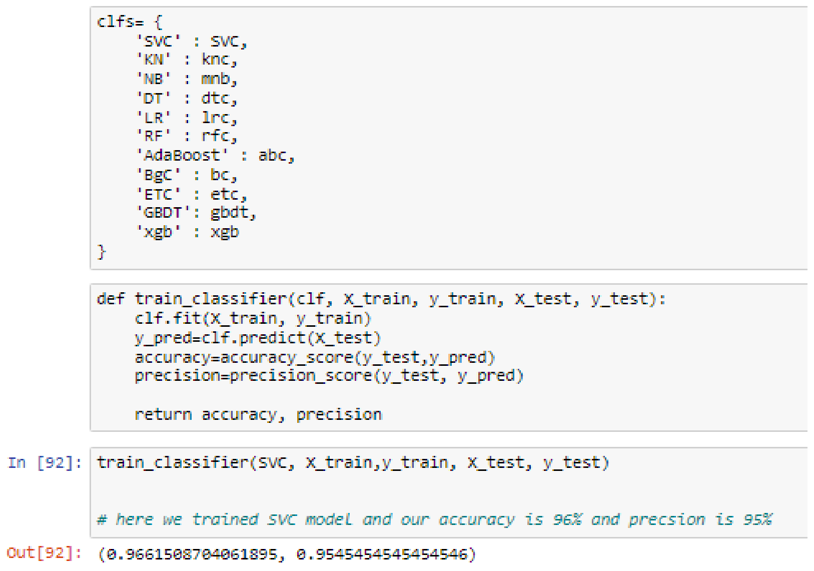
Multinomial Naïve Bayes→ As we observe, when using Multinomial Naive Bayes with the TFIDF vectorizer, it achieves a precision score of 1, indicating perfect precision. Additionally, the false positive rate is 0, meaning there are no instances incorrectly classified as positive when they are actually negative.

Bernoulli Naïve Bayes→ As we observe, the Bernoulli model provides a precision of 94%, which is lower than the precision achieved by the Multinomial Naive Bayes. For our analysis, higher precision is desirable, as it indicates a lower false positive rate. Thus, we consider the Multinomial Naive Bayes model to be more suitable for our needs.

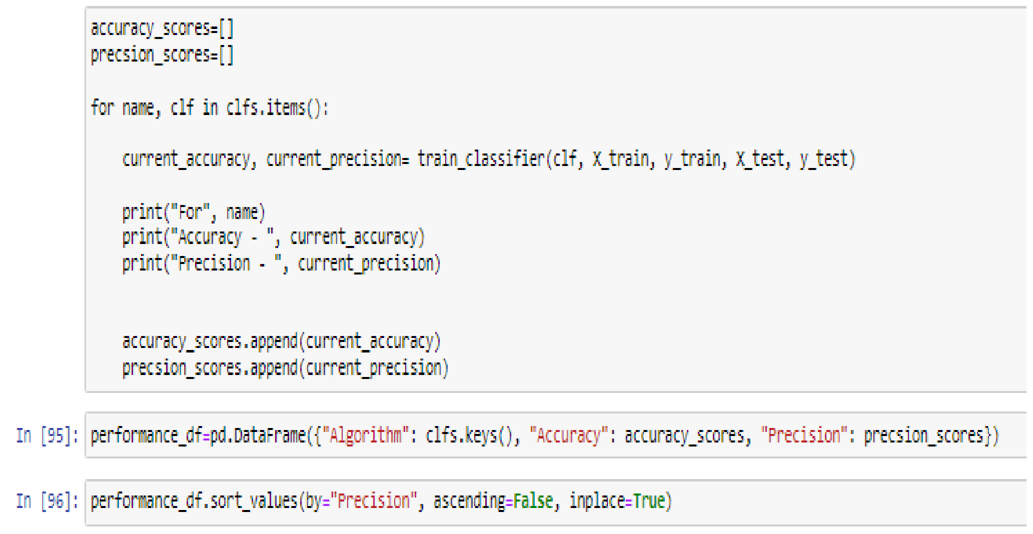
We achieved higher precision with the Multinomial Naive Bayes. However, a sudden query arises about the performance of other algorithms; they might perform better. To explore this, we decided to import all the algorithms and conduct the process to obtain better precision and accuracy, which are crucial for our analysis. After experimenting with a set of algorithms, we recorded the accuracy and precision of each algorithm and stored them in a data frame in descending order.



```
In [108]: from sklearn.linear_model import LogisticRegression
          from sklearn.svm import SVC
          from sklearn.naive_bayes import MultinomialNB
          from sklearn.tree import DecisionTreeClassifier
          from sklearn.neighbors import KNeighborsClassifier
          from sklearn. ensemble import RandomForestClassifier
          from sklearn.ensemble import AdaBoostClassifier
          from sklearn.ensemble import BaggingClassifier
          from sklearn.ensemble import ExtraTreesClassifier
          from sklearn.ensemble import GradientBoostingClassifier
          from xgboost import XGBClassifier

In [109]: SVC=SVC(kernel='sigmoid', gamma=1.0)
          knc=KNeighborsClassifier()
          mnb=MultinomialNB()
          dtc=DecisionTreeClassifier(max_depth=5)
          lrc=LogisticRegression(solver='liblinear', penalty='l1')
          rfc=RandomForestClassifier(n_estimators=50, random_state=2)
          abc=AdaBoostClassifier(n_estimators=50, random_state=2)
          bc=BaggingClassifier(n_estimators=50, random_state=2)
          etc=ExtraTreesClassifier(n_estimators=50, random_state=2)
          gbdt=GradientBoostingClassifier(n_estimators=50, random_state=2)
          xgb=XGBClassifier(n_estimators=50, random_state=2)
```

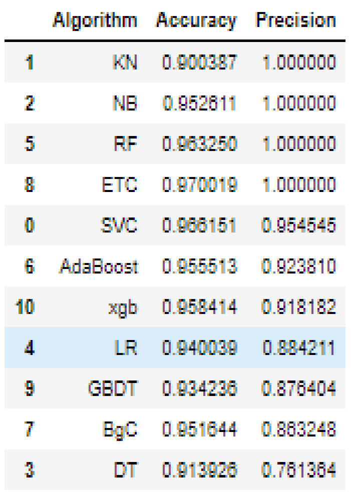


```
clfs= {
    'SVC' : SVC,
    'KN' : knc,
    'NB' : mnb,
    'DT' : dtc,
    'LR' : lrc,
    'RF' : rfc,
    'AdaBoost' : abc,
    'BgC' : bc,
    'ETC' : etc,
    'GBDT': gbdt,
    'xgb' : xgb
}

def train_classifier(clf, X_train, y_train, X_test, y_test):
    clf.fit(X_train, y_train)
    y_pred=clf.predict(X_test)
    accuracy=accuracy_score(y_test,y_pred)
    precision=precision_score(y_test, y_pred)

    return accuracy, precision

In [92]: train_classifier(SVC, X_train,y_train, X_test, y_test)

         # here we trained SVC model and our accuracy is 96% and precsion is 95%

Out[92]: (0.9661508704061895, 0.9545454545454546)
```



```
accuracy_scores=[]
precsion_scores=[]

for name, clf in clfs.items():

    current_accuracy, current_precision= train_classifier(clf, X_train, y_train, X_test, y_test)

    print("For", name)
    print("Accuracy - ", current_accuracy)
    print("Precision - ", current_precision)


    accuracy_scores.append(current_accuracy)
    precsion_scores.append(current_precision)
```

```
In [95]: performance_df=pd.DataFrame({"Algorithm": clfs.keys(), "Accuracy": accuracy_scores, "Precision": precsion_scores})

In [96]: performance_df.sort_values(by="Precision", ascending=False, inplace=True)
```



	Algorithm	Accuracy	Precision
1	KN	0.900387	1.000000
2	NB	0.952611	1.000000
5	RF	0.963250	1.000000
8	ETC	0.970019	1.000000
0	SVC	0.966151	0.954545
6	AdaBoost	0.955513	0.923810
10	xgb	0.958414	0.918182
4	LR	0.940039	0.884211
9	GBDT	0.934236	0.876404
7	BgC	0.951644	0.863248
3	DT	0.913926	0.761364

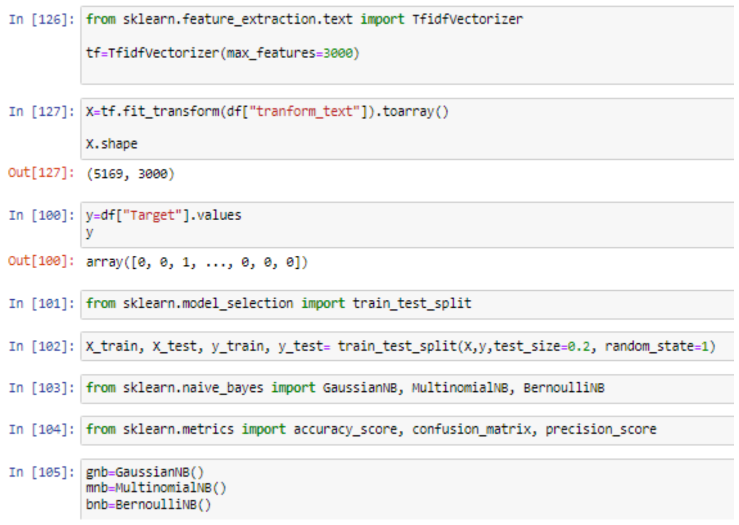
So, among the following algorithms, K-Nearest Neighbours (KN) and Naïve Bayes are performing well. However, we choose Naïve Bayes because it provides an accuracy of 96% and a precision of 1. Additionally, Extra Trees Classifier (ETC) also demonstrates good accuracy and precision.

Therefore, we make the decision to proceed with Naïve Bayes for our analysis.

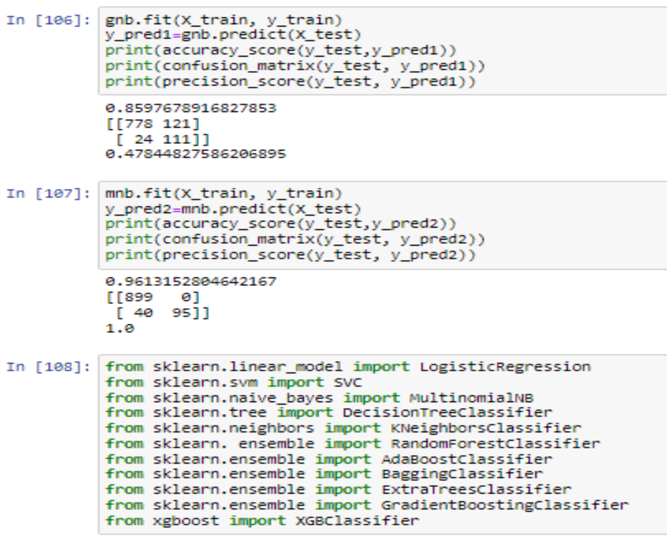
MODEL IMPROVEMENT:

Improvement 1: We have set the maximum number of features for TFIDF as 3000. After implementing this configuration, we conducted the entire process with the hope that it would enhance both the accuracy and precision of our model.

Before drawing conclusions about the model, it is essential to understand the implications of the TFIDF max features. The max-features parameter determines the number of unique words we want to include in the vocabulary.
In our investigation, we tried various values like 1000, 1500, 2000, and 2500, but they did not yield satisfactory results. Therefore, we decided to experiment with a max-features value of 3000, including only the top 3000 unique words in the vocabulary. This adjustment is an attempt to improve the model's performance and achieve better results.

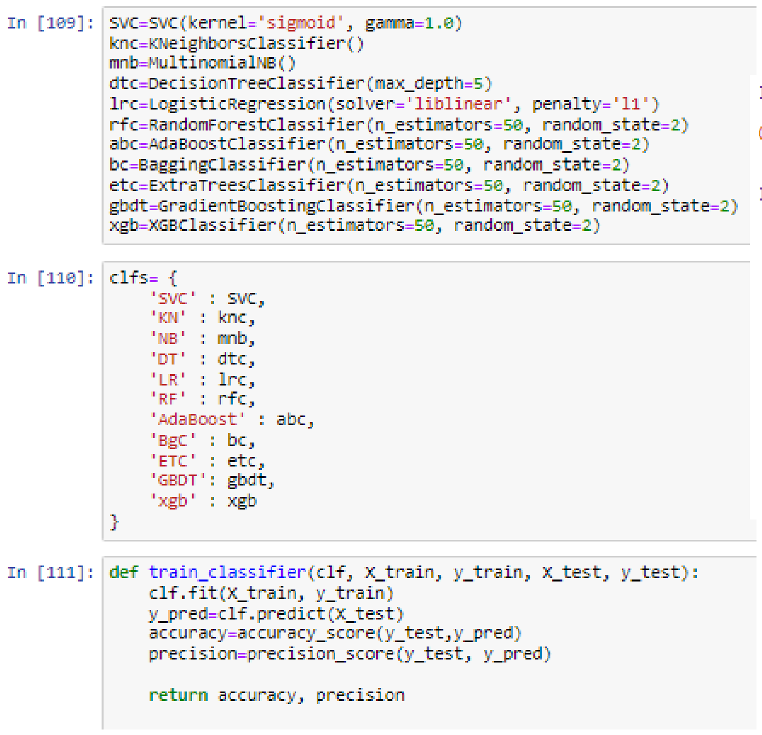


```
In [126]: from sklearn.feature_extraction.text import TfidfVectorizer

          tf=TfidfVectorizer(max_features=3000)

In [127]: X=tf.fit_transform(df["tranform_text"]).toarray()

          X.shape

Out[127]: (5169, 3000)

In [100]: y=df["Target"].values
          y

Out[100]: array([0, 0, 1, ..., 0, 0, 0])

In [101]: from sklearn.model_selection import train_test_split

In [102]: X_train, X_test, y_train, y_test= train_test_split(X,y,test_size=0.2, random_state=1)

In [103]: from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB

In [104]: from sklearn.metrics import accuracy_score, confusion_matrix, precision_score

In [105]: gnb=GaussianNB()
          mnb=MultinomialNB()
          bnb=BernoulliNB()
```

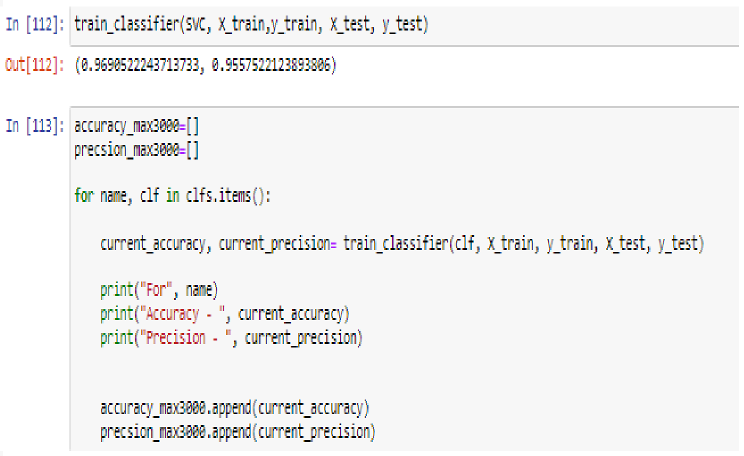


```
In [106]: gnb.fit(X_train, y_train)
          y_pred1=gnb.predict(X_test)
          print(accuracy_score(y_test,y_pred1))
          print(confusion_matrix(y_test, y_pred1))
          print(precision_score(y_test, y_pred1))

          0.8597678916827853
          [[778 121]
           [ 24 111]]
          0.47844827586206895

In [107]: mnb.fit(X_train, y_train)
          y_pred2=mnb.predict(X_test)
          print(accuracy_score(y_test,y_pred2))
          print(confusion_matrix(y_test, y_pred2))
          print(precision_score(y_test, y_pred2))

          0.9613152804642167
          [[899   0]
           [ 40  95]]
          1.0

In [108]: from sklearn.linear_model import LogisticRegression
          from sklearn.svm import SVC
          from sklearn.naive_bayes import MultinomialNB
          from sklearn.tree import DecisionTreeClassifier
          from sklearn.neighbors import KNeighborsClassifier
          from sklearn. ensemble import RandomForestClassifier
          from sklearn.ensemble import AdaBoostClassifier
          from sklearn.ensemble import BaggingClassifier
          from sklearn.ensemble import ExtraTreesClassifier
          from sklearn.ensemble import GradientBoostingClassifier
          from xgboost import XGBClassifier
```

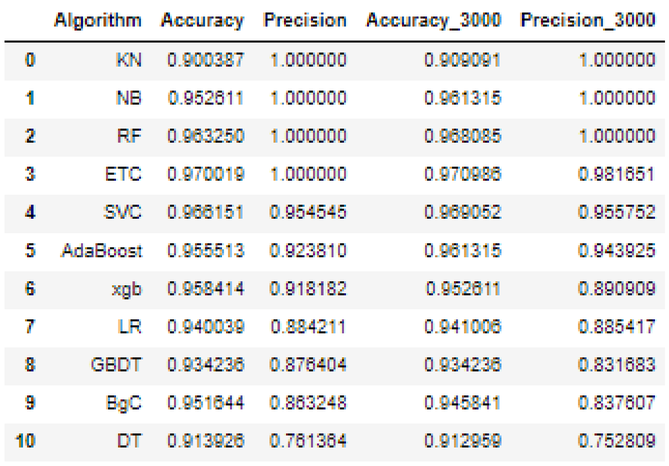
From our analysis, we observed that Multinomial Naïve Bayes outperforms Gaussian Naïve Bayes. The precision achieved by Multinomial Naïve Bayes is 1, a critical factor considering the imbalanced nature of the data. Now, our next step is to compare the results with other algorithms to determine if there are any models that perform better than Naïve Bayes.



```
In [109]: svc=SVC(kernel='sigmoid', gamma=1.0)
          knc=KNeighborsClassifier()
          mnb=MultinomialNB()
          dtc=DecisionTreeClassifier(max_depth=5)
          lrc=LogisticRegression(solver='liblinear', penalty='l1')
          rfc=RandomForestClassifier(n_estimators=50, random_state=2)
          abc=AdaBoostClassifier(n_estimators=50, random_state=2)
          bc=BaggingClassifier(n_estimators=50, random_state=2)
          etc=ExtraTreesClassifier(n_estimators=50, random_state=2)
          gbdt=GradientBoostingClassifier(n_estimators=50, random_state=2)
          xgb=XGBClassifier(n_estimators=50, random_state=2)

In [110]: clfs= {
              'SVC' : svc,
              'KN' : knc,
              'NB' : mnb,
              'DT' : dtc,
              'LR' : lrc,
              'RF' : rfc,
              'AdaBoost' : abc,
              'BgC' : bc,
              'ETC' : etc,
              'GBDT': gbdt,
              'xgb' : xgb
          }

In [111]: def train_classifier(clf, X_train, y_train, X_test, y_test):
              clf.fit(X_train, y_train)
              y_pred=clf.predict(X_test)
              accuracy=accuracy_score(y_test,y_pred)
              precision=precision_score(y_test, y_pred)

              return accuracy, precision
```



```
In [112]: train_classifier(SVC, X_train,y_train, X_test, y_test)

Out[112]: (0.9690522243713733, 0.9557522123893806)

In [113]: accuracy_max3000=[]
          precsion_max3000=[]

          for name, clf in clfs.items():

              current_accuracy, current_precision= train_classifier(clf, X_train, y_train, X_test, y_test)

              print("For", name)
              print("Accuracy - ", current_accuracy)
              print("Precision - ", current_precision)


              accuracy_max3000.append(current_accuracy)
              precsion_max3000.append(current_precision)
```



	Algorithm	Accuracy	Precision	Accuracy_3000	Precision_3000
0	KN	0.900387	1.000000	0.909091	1.000000
1	NB	0.952611	1.000000	0.961315	1.000000
2	RF	0.963250	1.000000	0.968085	1.000000
3	ETC	0.970019	1.000000	0.970986	0.981651
4	SVC	0.966151	0.954545	0.969052	0.955752
5	AdaBoost	0.955513	0.923810	0.961315	0.943925
6	xgb	0.958414	0.918182	0.952611	0.890909
7	LR	0.940039	0.884211	0.941006	0.885417
8	GBDT	0.934236	0.876404	0.934236	0.831683
9	BgC	0.951644	0.863248	0.945841	0.837607
10	DT	0.913926	0.761364	0.912959	0.752809

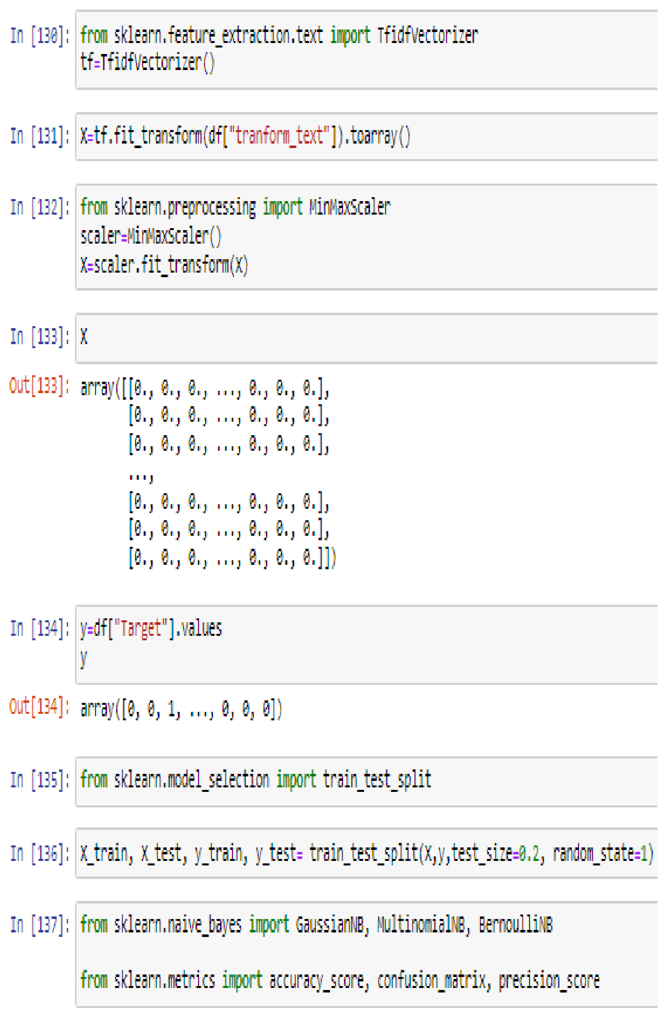
Interpretation: By examining the data frame containing the previous and current outputs, we observed that Naïve Bayes provides the highest accuracy and precision. The accuracy increased from 92% to 96%, indicating an improvement in the model's overall performance. Moreover, the precision remains the same, further confirming the reliability and consistency of the Naïve Bayes algorithm.

Improvement 2: to get a better performance I have done the scaling of X ranges from 0 to 1, before I directly show the results it better to get a little recap of the scaling, in simple words scaling means to scales the x range from 0 to 1 means transforming the values of the x-axis in a dataset to a new range that starts from 0 and ends at 1. This scaling process is known as **Min-Max scaling or normalization**.
In Min-Max scaling, each data point in the original range is transformed to a new value in the 0 to 1 range based on the formula:

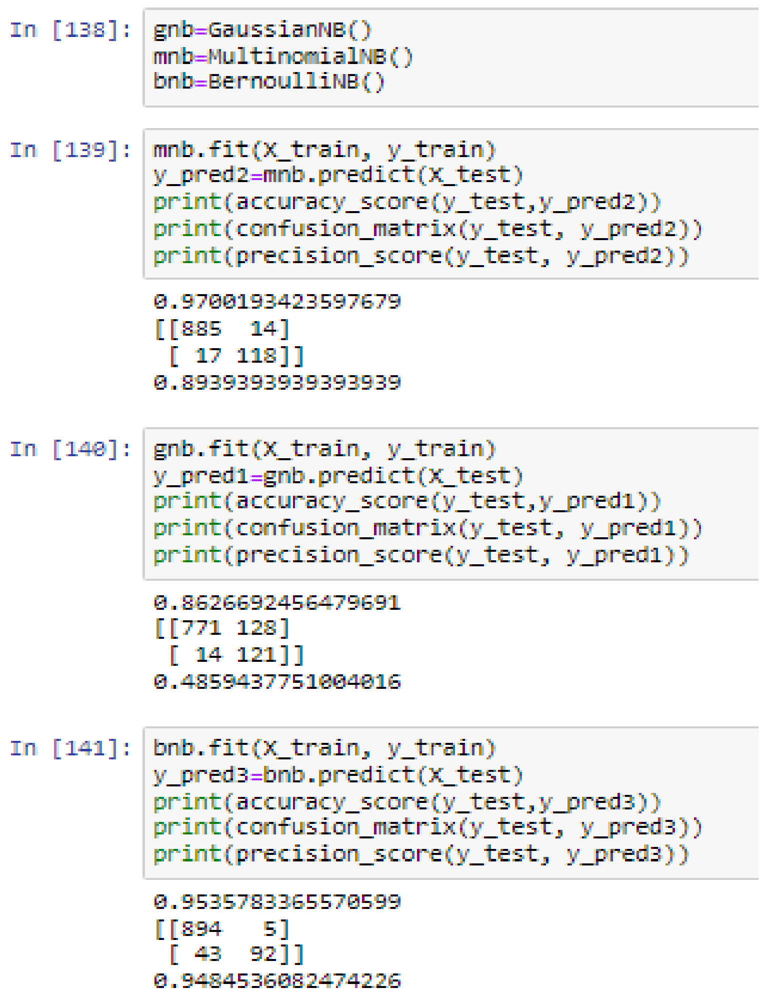
New value = (old value – min value) / (max value – min value)

Where:
Old value is the original value of the data point.   Min value is the minimum value in the original range.
Max value is the maximum value in the original range.

The result is that all the data points are transformed to a new range where the minimum value becomes 0, and the maximum value becomes 1. Scaling the x range from 0 to 1 is commonly used in machine learning and data analysis to ensure that all features or variables are on the same scale, which can lead to better performance and more accurate models, especially for algorithms sensitive to the magnitude of the input features.



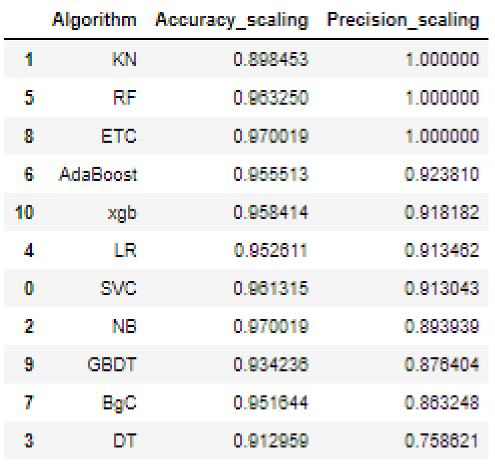
```
In [130]: from sklearn.feature_extraction.text import TfidfVectorizer
          tf=TfidfVectorizer()

In [131]: X=tf.fit_transform(df["tranform_text"]).toarray()

In [132]: from sklearn.preprocessing import MinMaxScaler
          scaler=MinMaxScaler()
          X=scaler.fit_transform(X)

In [133]: X

Out[133]: array([[0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 ...,
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.],
                 [0., 0., 0., ..., 0., 0., 0.]])

In [134]: y=df["Target"].values
          y

Out[134]: array([0, 0, 1, ..., 0, 0, 0])

In [135]: from sklearn.model_selection import train_test_split

In [136]: X_train, X_test, y_train, y_test= train_test_split(X,y,test_size=0.2, random_state=1)

In [137]: from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB

          from sklearn.metrics import accuracy_score, confusion_matrix, precision_score
```



```
In [138]: gnb=GaussianNB()
          mnb=MultinomialNB()
          bnb=BernoulliNB()

In [139]: mnb.fit(X_train, y_train)
          y_pred2=mnb.predict(X_test)
          print(accuracy_score(y_test,y_pred2))
          print(confusion_matrix(y_test, y_pred2))
          print(precision_score(y_test, y_pred2))

          0.9700193423597679
          [[885  14]
           [ 17 118]]
          0.8939393939393939

In [140]: gnb.fit(X_train, y_train)
          y_pred1=gnb.predict(X_test)
          print(accuracy_score(y_test,y_pred1))
          print(confusion_matrix(y_test, y_pred1))
          print(precision_score(y_test, y_pred1))

          0.8626692456479691
          [[771 128]
           [ 14 121]]
          0.4859437751004016

In [141]: bnb.fit(X_train, y_train)
          y_pred3=bnb.predict(X_test)
          print(accuracy_score(y_test,y_pred3))
          print(confusion_matrix(y_test, y_pred3))
          print(precision_score(y_test, y_pred3))

          0.9535783365570599
          [[894   5]
           [ 43  92]]
          0.9484536082474226
```

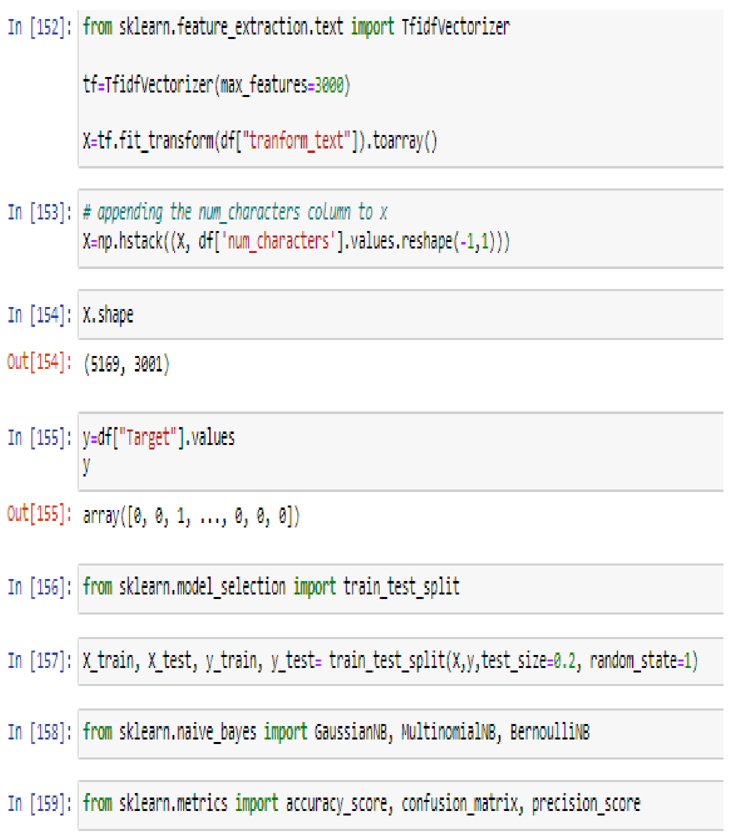
Interpretation: After scaling, we observed that the precision of Bernoulli Naïve Bayes was better than that of Gaussian Naïve Bayes and Multinomial Naïve Bayes. However, the precision was not as high as the value of 1 Obtained from the previous Multinomial Naïve Bayes model.

Interpretation: After scaling, we didn't observe a significant Improvement in our model's performance. In fact, it reduced the Precision of Naïve Bayes, which initially provided a precision of 1 With TFIDF. Another point to consider is that scaling increased the Accuracy, but it is not our primary concern for this analysis since Our data is imbalanced. Thus, we need to focus on precision, which Is more critical in dealing with imbalanced datasets.

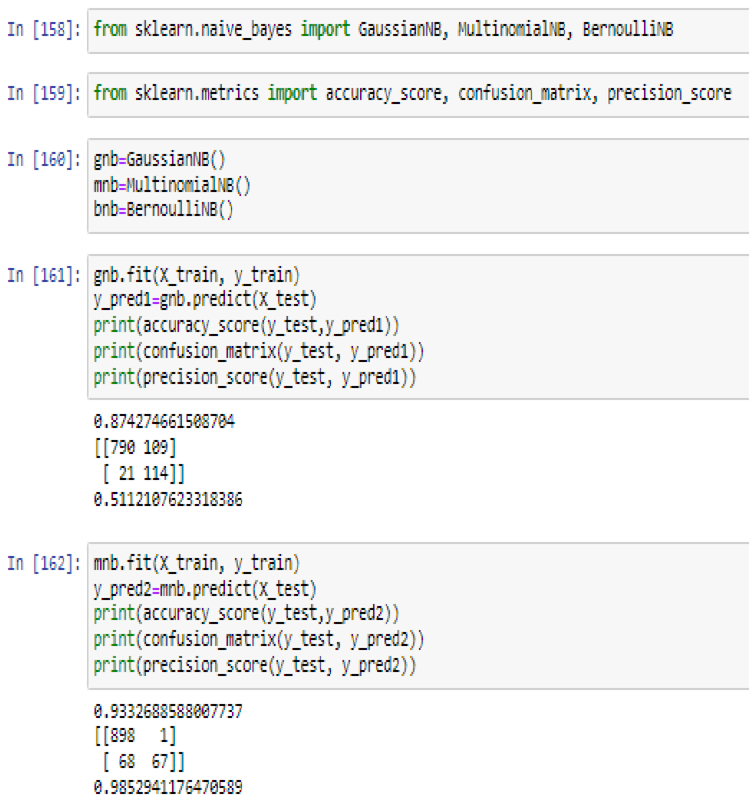


	Algorithm	Accuracy_scaling	Precision_scaling
1	KN	0.898453	1.000000
5	RF	0.963250	1.000000
8	ETC	0.970019	1.000000
6	AdaBoost	0.955513	0.923810
10	xgb	0.958414	0.918182
4	LR	0.952611	0.913462
0	SVC	0.961315	0.913043
2	NB	0.970019	0.893939
9	GBDT	0.934236	0.876404
7	BgC	0.951644	0.863248
3	DT	0.912959	0.758621

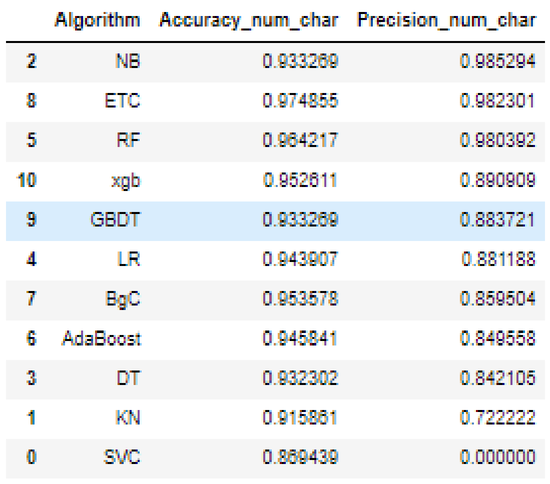
Improvement 3: To improve the results, I appended a new feature that I discovered during the exploratory data analysis (EDA). I included this feature in X, hoping to achieve better accuracy and precision in the model.



```
In [152]: from sklearn.feature_extraction.text import TfidfVectorizer

          tf=TfidfVectorizer(max_features=3000)

          X=tf.fit_transform(df["tranform_text"]).toarray()

In [153]: # appending the num_characters column to x
          X=np.hstack((X, df['num_characters'].values.reshape(-1,1)))

In [154]: X.shape

Out[154]: (5169, 3001)

In [155]: y=df["Target"].values
          y

Out[155]: array([0, 0, 1, ..., 0, 0, 0])

In [156]: from sklearn.model_selection import train_test_split

In [157]: X_train, X_test, y_train, y_test= train_test_split(X,y,test_size=0.2, random_state=1)

In [158]: from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB

In [159]: from sklearn.metrics import accuracy_score, confusion_matrix, precision_score
```



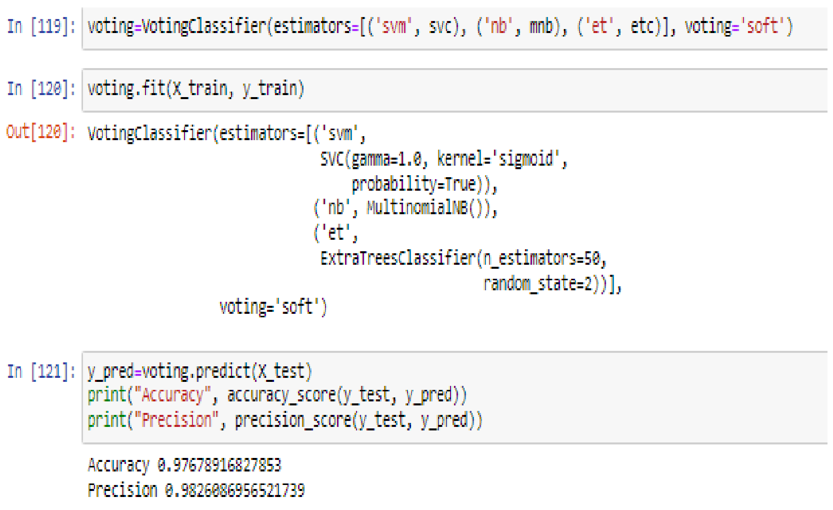
```
In [158]: from sklearn.naive_bayes import GaussianNB, MultinomialNB, BernoulliNB

In [159]: from sklearn.metrics import accuracy_score, confusion_matrix, precision_score

In [160]: gnb=GaussianNB()
          mnb=MultinomialNB()
          bnb=BernoulliNB()

In [161]: gnb.fit(X_train, y_train)
          y_pred1=gnb.predict(X_test)
          print(accuracy_score(y_test,y_pred1))
          print(confusion_matrix(y_test, y_pred1))
          print(precision_score(y_test, y_pred1))

          0.874274661508704
          [[790 109]
           [ 21 114]]
          0.5112107623318386

In [162]: mnb.fit(X_train, y_train)
          y_pred2=mnb.predict(X_test)
          print(accuracy_score(y_test,y_pred2))
          print(confusion_matrix(y_test, y_pred2))
          print(precision_score(y_test, y_pred2))

          0.9332688588007737
          [[898   1]
           [ 68  67]]
          0.9852941176470589
```



	Algorithm	Accuracy_num_char	Precision_num_char
2	NB	0.933269	0.985294
8	ETC	0.974855	0.982301
5	RF	0.964217	0.980392
10	xgb	0.952611	0.890909
9	GBDT	0.933269	0.883721
4	LR	0.943907	0.881188
7	BgC	0.953578	0.859504
6	AdaBoost	0.945841	0.849558
3	DT	0.932302	0.842105
1	KN	0.915861	0.722222
0	SVC	0.869439	0.000000

Interpretation: The highest precision we observed is from Naïve Bayes. However, compared to the previous TFIDF Results, it provided lower accuracy and precision. While we Could explore other model improvements.

Improvement 4: For better performance, I employed the voting classifier. Let's delve into what the voting classifier is: In simple terms, it's an ensemble machine learning technique that combines predictions from multiple individual classifiers to make a final decision. The voting classifier aims to enhance model performance and robustness through model averaging. It can utilize various algorithms and classifier types, and the ultimate prediction is determined by majority voting (for classification) or averaging (for regression). Voting classifiers prove advantageous when base classifiers possess complementary strengths, ultimately improving model accuracy and generalization across various machine learning tasks.

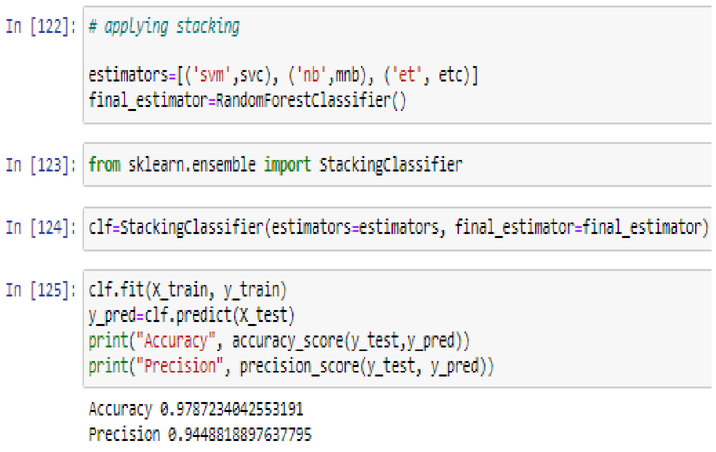

```
In [119]: voting=VotingClassifier(estimators=[('svm', svc), ('nb', mnb), ('et', etc)], voting='soft')

In [120]: voting.fit(X_train, y_train)

Out[120]: VotingClassifier(estimators=[('svm',
                                        SVC(gamma=1.0, kernel='sigmoid',
                                            probability=True)),
                                       ('nb', MultinomialNB()),
                                       ('et',
                                        ExtraTreesClassifier(n_estimators=50,
                                                             random_state=2))],
                           voting='soft')

In [121]: y_pred=voting.predict(X_test)
          print("Accuracy", accuracy_score(y_test, y_pred))
          print("Precision", precision_score(y_test, y_pred))

          Accuracy 0.97678916827853
          Precision 0.9826086956521739
```

Interpretation: The classifier achieved an Accuracy of approximately 97.87% on the Test dataset, correctly classifying the Majority of instances. Its precision was approximately 98.2%, indicating that Around 98.2% of the positive predictions Were true positives.

Improvement 5: Now, the fifth improvement we consider is stacking. Stacking, also known as Stacked Generalization, is an ensemble machine learning technique that combines multiple models (learners) to enhance predictive performance. In stacking, base-level models (first-level models) are trained on the same training data, and their predictions become input features for a higher-level model (meta-model) to make the final prediction.

The stacking technique leverages the diverse strengths of different base models and enables the meta-model to learn from the combined knowledge of these base models. This approach leads to improved predictive performance, better generalization, and increased robustness against over fitting.

Stacking is a powerful technique within machine learning ensemble methods and is widely used to address complex problems and enhance predictive accuracy across various ap


```
In [122]: # applying stacking

          estimators=[('svm',svc), ('nb',mnb), ('et', etc)]
          final_estimator=RandomForestClassifier()

In [123]: from sklearn.ensemble import StackingClassifier

In [124]: clf=StackingClassifier(estimators=estimators, final_estimator=final_estimator)

In [125]: clf.fit(X_train, y_train)
          y_pred=clf.predict(X_test)
          print("Accuracy", accuracy_score(y_test,y_pred))
          print("Precision", precision_score(y_test, y_pred))

          Accuracy 0.9787234042553191
          Precision 0.9448818897637795
```

Interpretation: The classifier achieved an Accuracy of approximately 97.87% on the Test dataset, correctly classifying the Majority of instances. Its precision was Approximately 94.49%, indicating that Around 94.49% of the positive predictions Were true positives.

CONCLUSION:

After thoroughly reviewing all 5 improvements, I have reached a decision. Considering the precision and accuracy achieved by each improvement, I have chosen Improvement 1 as the preferred approach. This improvement involves changing the TFIDF max-features to 3000.

The reason for selecting Improvement 1 is that it demonstrated notably good precision and accuracy compared to the other improvements. By setting the TFIDF max-features to 3000, we were able to enhance the model's performance, resulting in improved predictive power and accuracy. Therefore, Improvement 1 is the most suitable choice for our analysis.

MODEL DEPLOYEMENT:

Now that the model is built, my next step is to consider deployment. However, since I am not affiliated with any organization, I am unable to deploy the model due to the high cost associated with model deployment. However, there is an alternative solution we can explore. We could use stream lit to create a website where users can input text or SMS messages, and our model will provide the output indicating whether the message is spam or not. This way, we can still make the model accessible and useful to others through the web application.