

SMS Spam Detection Project

Author : Nahid Hasan
Department of CSE (20212041010),
North Western University.

Objective:

The goal of this project is to build a machine learning model that can classify SMS messages as spam (unwanted messages) or ham (legitimate messages). This helps automate the process of filtering unwanted messages, improving user experience and security.

Dataset

We will be using the SMS Spam Collection Dataset, available on [Kaggle](https://www.kaggle.com/datasets/uciml/sms-spam-collection-dataset). This dataset contains labeled SMS messages, making it suitable for binary classification. Each message is labeled as either:

- **Spam:** Messages identified as spam.
- **Ham:** Messages identified as not spam.

Dataset Link :

<https://www.kaggle.com/datasets/uciml/sms-spam-collection-dataset>

1. Loading the dataset

```
The dataset was gathered from https://www.kaggle.com/datasets/uciml/sms-spam-collection-dataset

[1] from google.colab import drive
    drive.mount('/content/drive')

Mounted at /content/drive

import pandas as pd
import numpy as np

file_path = '/content/drive/My Drive/Colab Notebooks/ML projects/sms spam classifier/spam.csv'
df = pd.read_csv(file_path, encoding='latin-1')
```

First we initialized the necessary environment for the project by mounting Google Drive and loading the dataset. The **pandas** and **numpy** libraries are imported for data manipulation and analysis.

The reason for specifying the **encoding='latin-1'** parameter while loading the dataset is to ensure the correct handling of special characters (such as accented letters or non-ASCII characters) that might be present in the data.

```
[4] df.sample(5)
```

	v1	v2	Unnamed: 2	Unnamed: 3	Unnamed: 4
5050	ham	With my sis lor... We juz watched italian job.	NaN	NaN	NaN
3694	ham	Hello, As per request from <#> Rs.5 ha...	NaN	NaN	NaN
4113	ham	Where are you ? What do you do ? How can you s...	NaN	NaN	NaN
1555	ham	Ok i found dis pierre cardin one which looks n...	NaN	NaN	NaN
1508	ham	Sounds like something that someone testing me ...	NaN	NaN	NaN

```
df.shape
```

```
(5572, 5)
```

Here we can get a quick preview of the dataset and its shape to understand its structure and dimensions.

2. Data Cleaning

This section is part of the data cleaning process. It checks the structure of the DataFrame and removes unnecessary columns and also checks if there are any null values or duplicate values.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 5572 entries, 0 to 5571  
Data columns (total 5 columns):  
#   Column          Non-Null Count  Dtype    
---  ---            
0    v1             5572 non-null   object   
1    v2             5572 non-null   object   
2    Unnamed: 2     50 non-null     object   
3    Unnamed: 3     12 non-null     object   
4    Unnamed: 4      6 non-null     object   
dtypes: object(5)  
memory usage: 217.8+ KB
```

```
[7] #drop last 3 columns  
df.drop(columns=['Unnamed: 2','Unnamed: 3','Unnamed: 4'],inplace = True)
```

Here we can see there are 3 unnamed columns with most of them being Null values. So we removed those columns using the drop() function.

```
#renaming the columns  
df.rename(columns={'v1':'target','v2':'text'},inplace=True)  
df.sample(5)
```

	target	text
2742	ham	No * am working on the ringing u thing but hav...
448	ham	LOL ... Have you made plans for new years?
1953	ham	Then just eat a shit and wait for ur monkey fa...
3754	ham	\lm on gloucesterroad what are uup to later?\""
2756	ham	Have a good trip. Watch out for . Remember whe...

Now we rename the columns in the dataset for better clarity and usability. Then we encode the target labels into numeric values using **LabelEncoder** from the **sklearn.preprocessing** module. Encoding categorical variables into numeric values is necessary for compatibility with machine learning models.

```
[ ] from sklearn.preprocessing import LabelEncoder
    encoder = LabelEncoder()

    # encoding the target feature
    df['target'] = encoder.fit_transform(df['target'])

[ ] # Display the mapping of categories
    category_mapping = dict(zip(encoder.classes_, encoder.transform(encoder.classes_)))
    print("Category Encoding Mapping:")
    for category, encoded_value in category_mapping.items():
        print(f"{category}: {encoded_value}")
```

⇒ Category Encoding Mapping:
ham: 0
spam: 1

We then checked for missing values and duplicate values and dropped the unnecessary values found.

```
# checking the missing values
df.isnull().sum()
```

⇒ 0

target	0
text	0

dtype: int64

As there are no null values so nothing needed to be done here.

```
# checking for duplicate values
df.duplicated().sum()
```

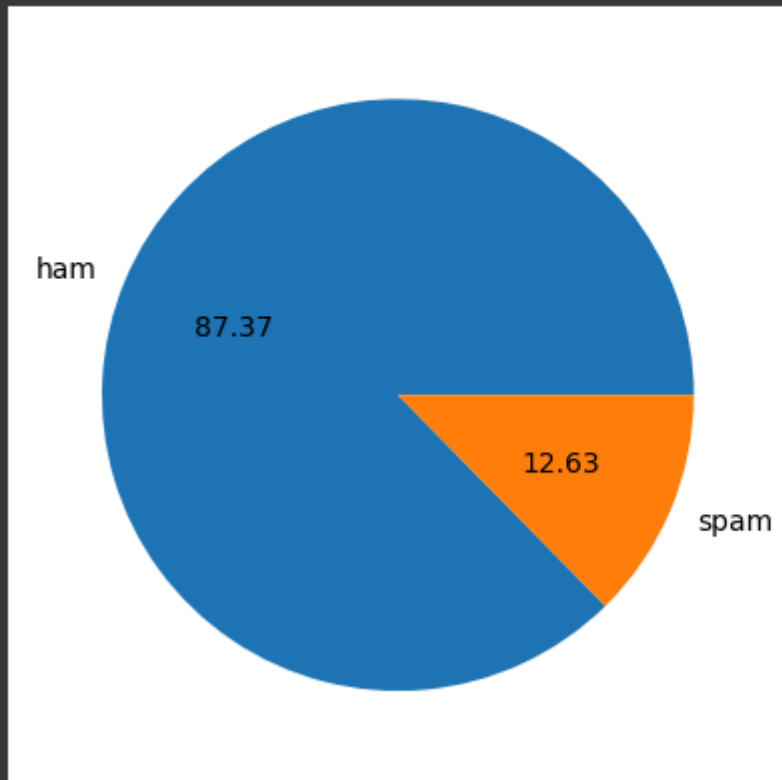
⇒ 403

```
[ ] # dropping the duplicate values (keeping at least one)
    df = df.drop_duplicates(keep='first')
```

3. EDA (Exploratory Data Analysis)

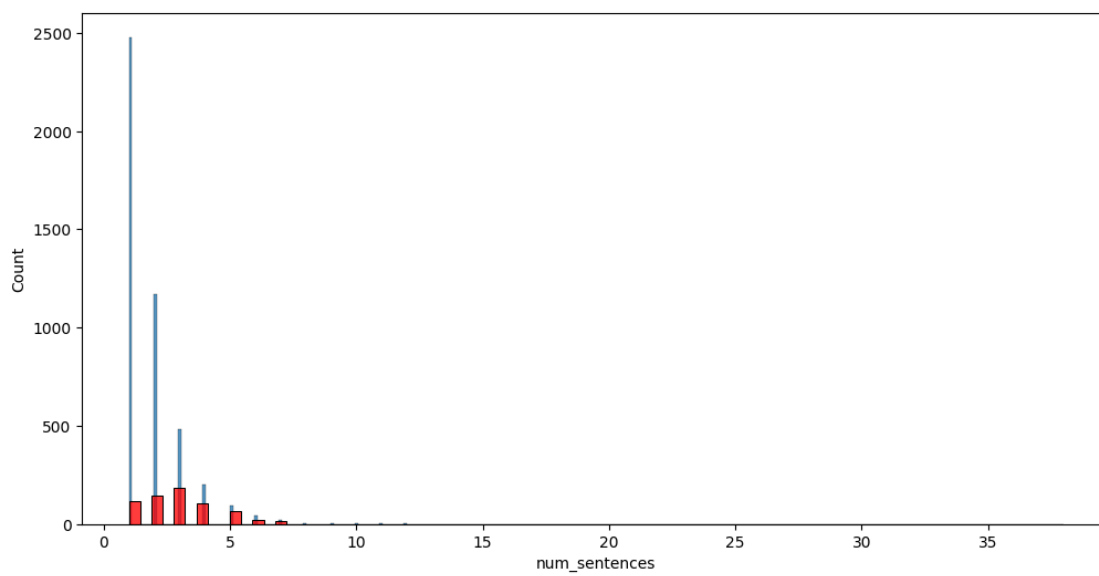
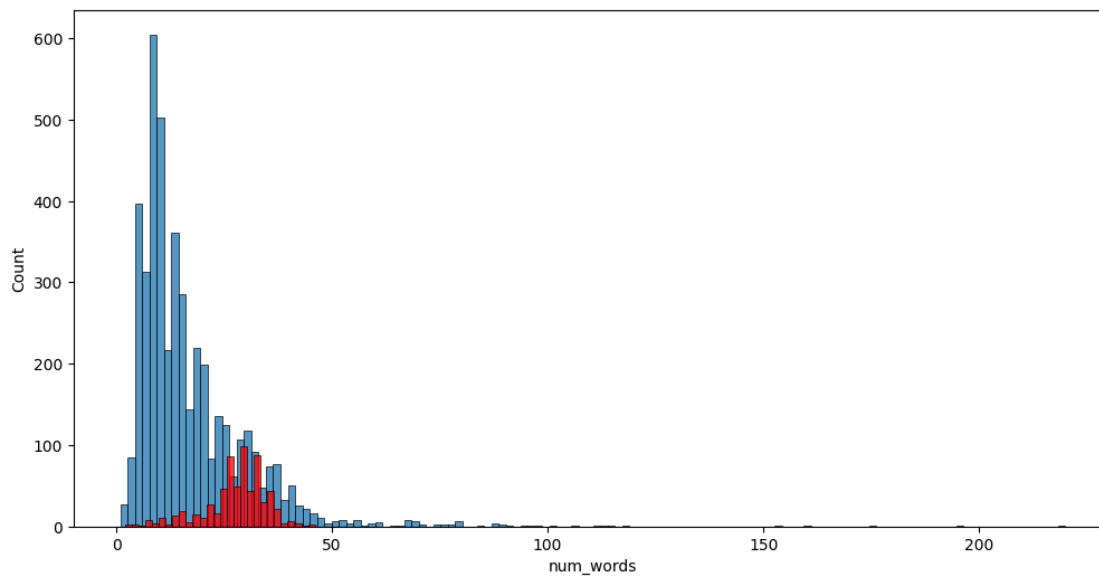
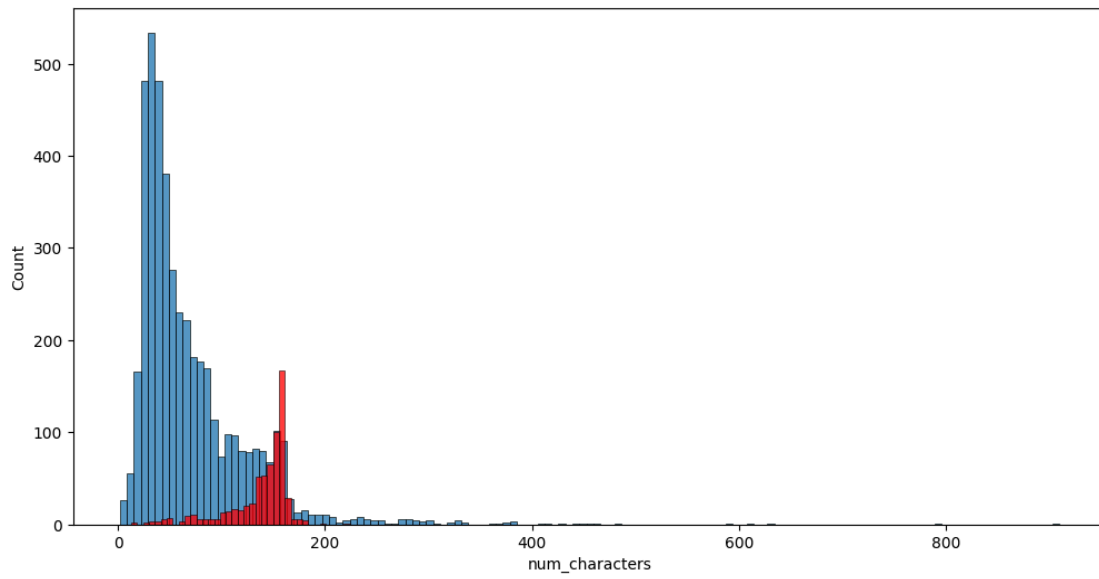
The EDA phase involves analyzing and visualizing the dataset to understand its structure, distribution, and relationships. This helps identify patterns, outliers, and potential issues (e.g., imbalance in target classes).

```
# Showing a basic pie chart for ham and spam ratio
import matplotlib.pyplot as plt
plt.pie(df['target'].value_counts(),labels=['ham','spam'],autopct="%0.2f")
plt.show()
```



From this pie chart we can see that the data is highly imbalanced. Only 12% of the data is Spam where almost 87% data is not spam.

Then we tried to visualize different distributions of the messages. Such as visualization that compares the distribution of the number of characters in ham (non-spam) and spam messages. It helps identify differences in message length patterns between the two classes. We also tried distribution of number of words and number of sentences. These highlight the key differences of the spam and not spam messages. Such as we can see from the following graphs that spam messages tend to be more wordy and have more characters than usual. The number of sentences also seem to be higher in the spam messages. These are some important findings that will help us decide on the future messages if they are spam or not.



4. Data Preprocessing

The preprocessing stage cleans and transforms raw SMS data into a suitable format for machine learning models. This involves text normalization and feature extraction to improve the model's performance.

For basic preprocessing for our code we did the following :

- **Lower casing the characters**
- **Tokenization**
- **Removing Special characters**
- **Removing Stop Words and punctuation**
- **Stemming**

```
# here we downloaded the stopwords dictionary from nltk
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
stopwords.words('english')
import string
string.punctuation

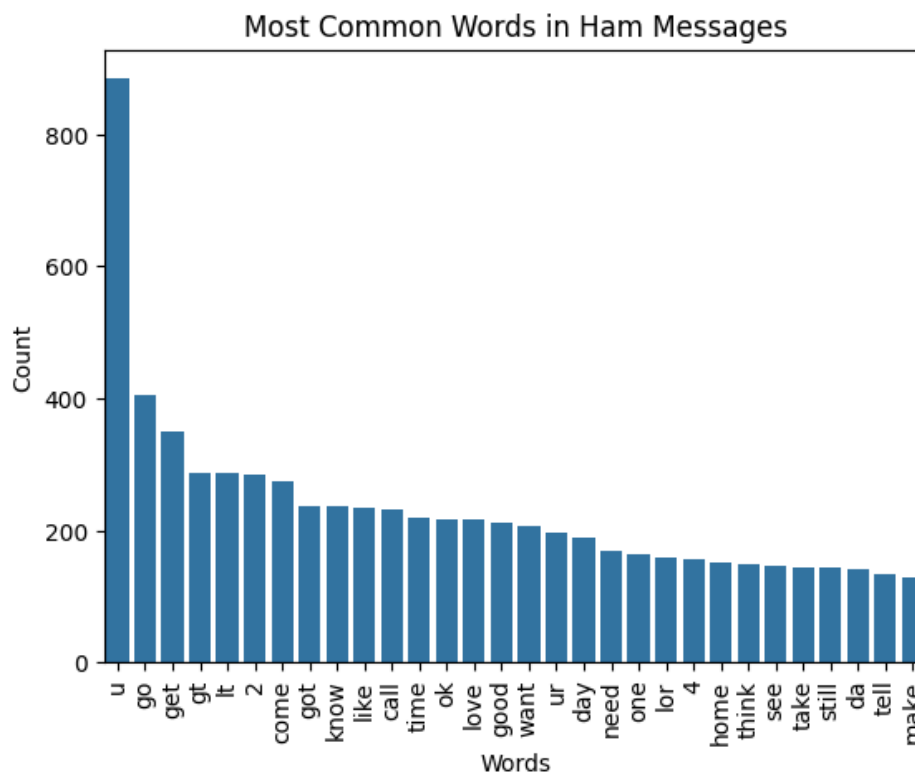
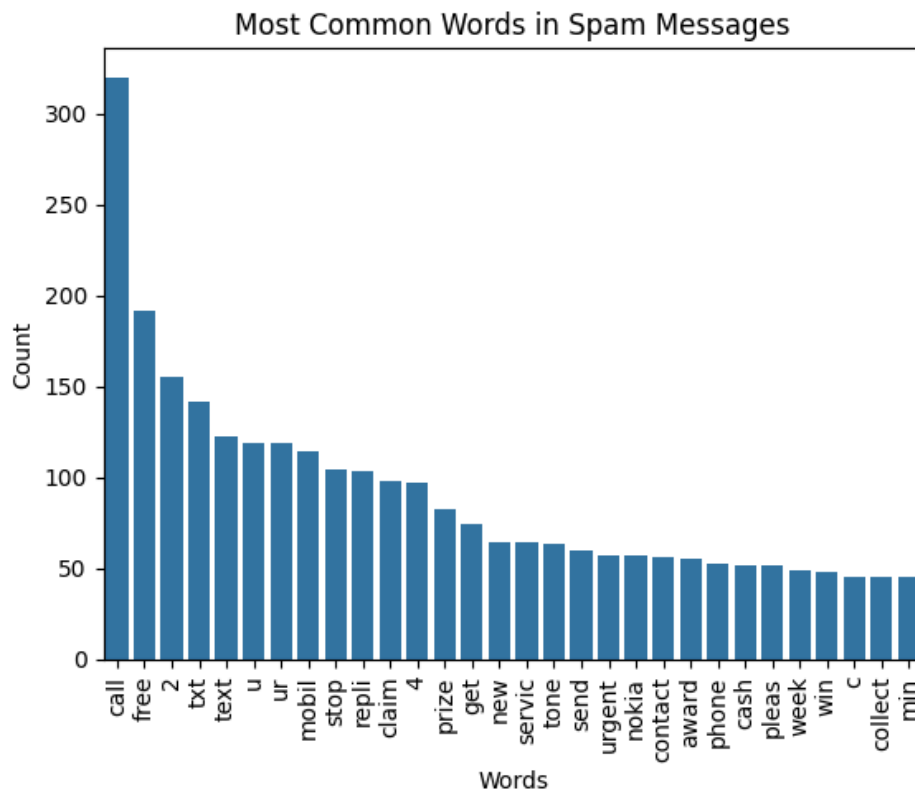
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'

[ ] # also imported a stemmer
from nltk.stem.porter import PorterStemmer
ps = PorterStemmer()
ps.stem('loving')
```

First we download corpus for the English **stopwords** and also the **punctuations**. We also imported a **Stemmer** that works very well on English words.

Then We created a function that **lowercases** the words and **tokenizes** it. Then we remove the **non-alphanumeric** tokens such as special characters. Then we also remove the **stopwords** and the **punctuations**. This reduces noise by focusing on meaningful tokens. We also applied stemming using the **Porter Stemmer (ps)**, reducing words to their root forms.

Then we created bar charts to display the 30 most common words in spam and not spam messages and their corresponding frequencies. It provides a more precise numeric visualization of dominant terms compared to a WordCloud.



5. Model Building

TFIDF

The **Term Frequency-Inverse Document Frequency (TF-IDF)** technique is used to convert text data into numerical vectors, capturing the importance of words relative to the entire dataset. This representation is essential for training machine learning models effectively.

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
cv = CountVectorizer()
tfidf = TfidfVectorizer(max_features=3000)

[67] X = tfidf.fit_transform(df['transformed_text']).toarray()

[68] X.shape
(5169, 3000)

[69] y = df['target'].values

[70] y
array([0, 0, 1, ..., 0, 0, 0])
```

Data splitting

This step splits the dataset into training and testing sets, ensuring that both sets represent the overall distribution of labels. Stratified splitting ensures that the proportion of spam and ham messages is preserved in both training and test datasets.

```
▼ Data Splitting (test/train)

[71] from sklearn.model_selection import train_test_split

[73] # Split the data with stratification
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
```

Model Definition

In this step, various machine learning algorithms and evaluation metrics are imported. These algorithms will be used to train and evaluate models for classifying SMS messages as ham or spam.

```
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score
from sklearn.naive_bayes import BernoulliNB, GaussianNB, MultinomialNB
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, GradientBoostingClassifier, BaggingClassifier
from xgboost import XGBClassifier
from sklearn.metrics import roc_curve, auc, precision_recall_curve, precision_recall_fscore_support, class_weight
```

This section defines multiple classification models to evaluate which performs best for SMS spam detection. The models are chosen based on their ability to handle different types of data and classification problems. Each classifier has its own strengths depending on the nature of the dataset.

```
# Define classifiers
bernoulli_nb = BernoulliNB() # Bernoulli Naive Bayes
gaussian_nb = GaussianNB() # Gaussian Naive Bayes
multinomial_nb = MultinomialNB() # Multinomial Naive Bayes
svm = SVC(kernel='sigmoid', gamma=1.0, class_weight='balanced') # Support Vector Machine (SVM)
logistic_reg = LogisticRegression(solver='liblinear', class_weight='balanced') # Logistic Regression
knn = KNeighborsClassifier() # K-Nearest Neighbors
id3 = DecisionTreeClassifier(criterion='entropy', class_weight='balanced') # ID3
c45 = DecisionTreeClassifier(criterion='entropy', class_weight='balanced') # C4.5
cart = DecisionTreeClassifier(criterion='gini', class_weight='balanced') # CART
rf = RandomForestClassifier(n_estimators=50, random_state=42, class_weight='balanced') # Random Forest
adaboost = AdaBoostClassifier(n_estimators=50, random_state=42) # AdaBoost
bagging = BaggingClassifier(n_estimators=50, random_state=42) # Bagging
extra_trees = ExtraTreesClassifier(n_estimators=50, random_state=42, class_weight='balanced') # Extra Tree
gb = GradientBoostingClassifier(n_estimators=50, random_state=42) # Gradient Boosting
xgb = XGBClassifier(n_estimators=50, random_state=42, scale_pos_weight=10) # XGBoost
```

Why Each Model?:

- **Naive Bayes:** Suitable for text classification with categorical data.
- **SVM:** Works well for high-dimensional feature spaces.
- **Logistic Regression:** Provides a simple but powerful classification model for binary classification tasks.
- **KNN:** Simple yet effective for many datasets.
- **Decision Trees:** Intuitive and interpretable models.
- **Ensemble Methods:** Combine multiple models to improve accuracy and robustness.

We also created a dictionary called `clfs` that contains the classifiers created in the previous step. The dictionary allows easy access to the classifier names.

Model Run Function

This function is responsible for training a classifier, making predictions on the test data, and evaluating the model's performance based on **accuracy** and **precision** metrics. It returns the accuracy and precision scores for further analysis or comparison.

```
[76] 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
```

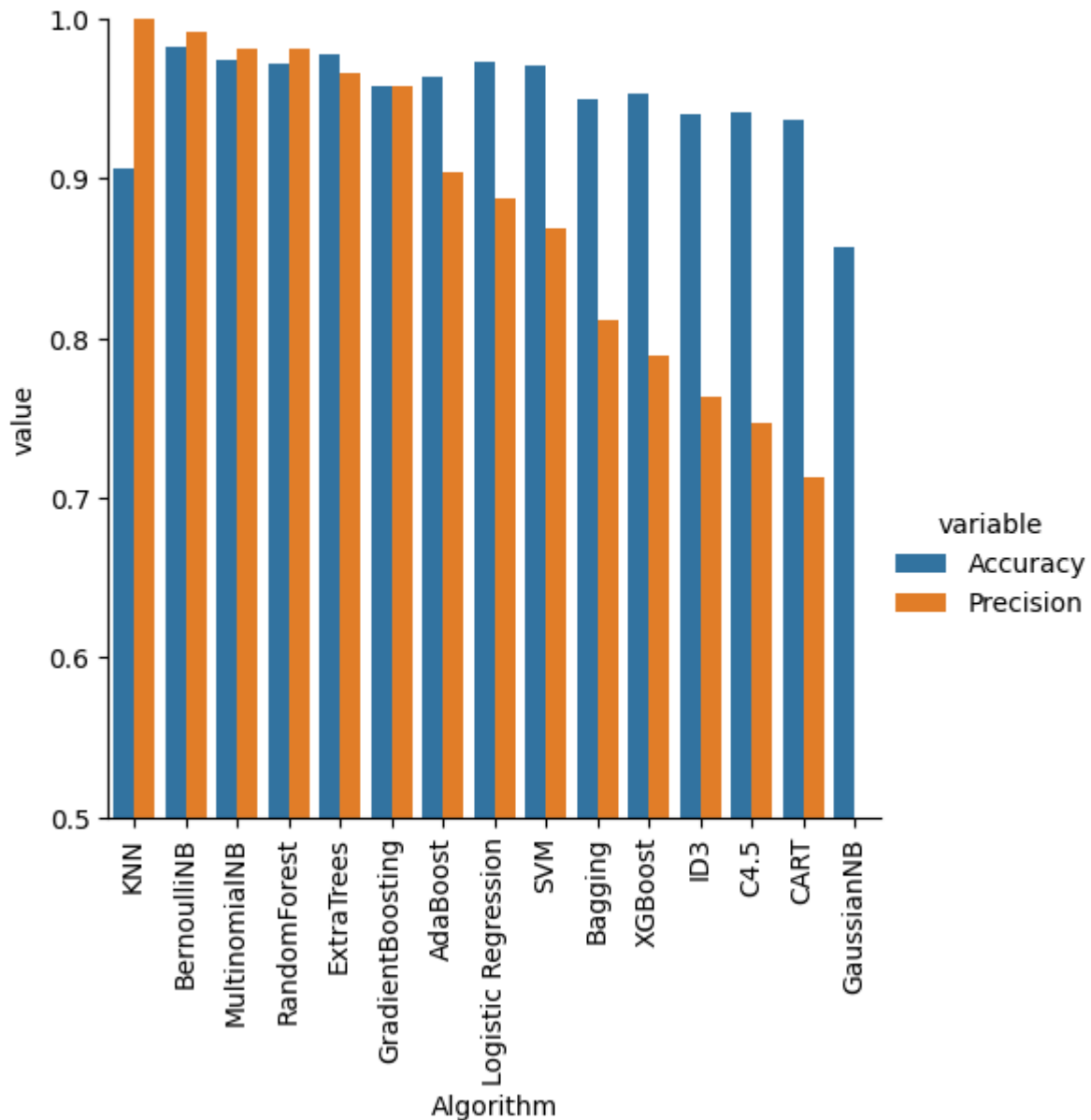
Running the models

This section iterates over the dictionary of classifiers (**clfs**), trains each classifier using the training data, evaluates its performance using the **train_classifier** function, and stores the results (**accuracy** and **precision**) for comparison.

```
[77] accuracy_scores = []  
      precision_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)  
          precision_scores.append(current_precision)
```

We also created a new DataFrame **performance_df** that stores the accuracy and precision scores for all classifiers. The DataFrame is sorted by precision in descending order to allow for easy comparison of the models based on their ability to correctly classify spam messages.

Also seaborn was used to create a bar plot that visualizes the performance (accuracy and precision) of each classifier, based on the **performance_df** DataFrame. The plot helps in comparing classifiers side by side and observing how each one performs across the two metrics.



```
# Bernoulli NB
bernoulli_nb.fit(X_train,y_train)
y_pred3 = bernoulli_nb.predict(X_test)
print(accuracy_score(y_test,y_pred3))
print(confusion_matrix(y_test,y_pred3))
print(precision_score(y_test,y_pred3))

0.9825918762088974
[[902  1]
 [ 17 114]]
0.991304347826087
```

Here we can see that though KNN (k nearest neighbor) has the highest precision it doesn't have a high accuracy rate. So we chose the next best model for the final model building. Here we can see that Bernoulli NB has both a good accuracy and precision. We can also see in the confusion matrix that it performed quite well overall.

6. Testing External Data

Now let's demonstrate how to test the trained SMS spam detection model on external data (new, unseen messages) to predict whether they are spam or not. The process involves preprocessing, vectorizing, and using the trained model for prediction.

```
msg1 = "Congratulations! You've won a $500 Amazon gift card. Claim it here"

msg2 = "Hey, got your message earlier. Sorry i was late"

msg3 = "Thank you for paying last month's bill. We're rewarding our very best customers with a gift for their loyalty. Click here! "
```

First we get some sample messages. We collected them from an online spam corpus. And also collected the not spam message from the random chat section.

```
# first preprocess the text
msg1 = transform_text(msg1)
msg2 = transform_text(msg2)
msg3 = transform_text(msg3)
```

Then we had to preprocess the data (lowercasing, tokenizing, stemming etc).

```
# use vectorizer
msg1_vectorized = tfidf.transform([msg1]).toarray()
# Predict using the trained model
prediction = bernoulli_nb.predict(msg1_vectorized)
# Print the prediction
if prediction[0] == 0:
    print("Not Spam")
else:
    print("Spam")
```


Then `tfidf.transform([msg1])`: This converts the preprocessed message (`msg1`) into a feature vector using the trained TF-IDF vectorizer. And Then based on the predicted label the program will print if it is spam or not spam. Here are all the results that we got :

test message 1

```
[95] # use vectorizer
msg1_vectorized = tfidf.transform([msg1]).toarray()

# Predict using the trained model
prediction = bernoulli_nb.predict(msg1_vectorized)

# Print the prediction
if prediction[0] == 0:
    print("Not Spam")
else:
    print("Spam")
```


 Spam

test message 2


```
[96] # use vectorizer
msg2_vectorized = tfidf.transform([msg2]).toarray()

# Predict using the trained model
prediction = bernoulli_nb.predict(msg2_vectorized)

# Print the prediction
if prediction[0] == 0:
    print("Not Spam")
else:
    print("Spam")
```


 Not Spam

test message 3

```
 # use vectorizer
msg3_vectorized = tfidf.transform([msg3]).toarray()

# Predict using the trained model
prediction = bernoulli_nb.predict(msg3_vectorized)

# Print the prediction
if prediction[0] == 0:
    print("Not Spam")
else:
    print("Spam")
```

 Spam

Conclusion

In this project, we have successfully built and tested a machine learning-based SMS spam detection system. The process involved multiple key steps, from data preprocessing to model training and evaluation:

1. **Data Collection & Cleaning:** We began by importing the SMS dataset and cleaning it to remove unnecessary columns and null values. We then renamed the columns and encoded the target labels to prepare the data for further analysis.
2. **Exploratory Data Analysis (EDA):** EDA helped us understand the distribution of spam and non-spam messages, as well as gain insights into the number of characters in each message. Visualizations like pie charts, histograms, and pair plots enabled a deeper understanding of the dataset and its characteristics.
3. **Text Preprocessing:** Preprocessing involved transforming the raw text messages by converting them to lowercase, tokenizing the text, removing special characters, stopwords, and applying stemming. These steps ensured that the text data was in a format suitable for vectorization and model training.
4. **Feature Extraction:** We utilized **TF-IDF (Term Frequency-Inverse Document Frequency)** vectorization to convert the preprocessed text data into numerical feature vectors that could be fed into the machine learning models.
5. **Model Training & Evaluation:** A variety of classifiers, including **Naive Bayes**, **Logistic Regression**, **Support Vector Machine (SVM)**, and **Random Forest**, were trained and evaluated on the dataset. The model's performance was measured using accuracy and precision scores, with a detailed comparison of each algorithm's effectiveness in detecting spam.
6. **Prediction on External Data:** Finally, we tested the model on new, unseen SMS messages (external data) to simulate real-world use. The model was able to predict whether a message was spam or not, providing practical value for identifying unwanted content.