**📄 Project Report: Email Spam Classifier | SMS Spam Classifier**

**1. Introduction**

Spam messages—whether through **emails** or **SMS**—pose a huge challenge in digital communication. Detecting spam automatically helps save time and protect users from malicious or irrelevant content.

In this project, we build a **Spam Classifier** using **Machine Learning (ML) and Natural Language Processing (NLP)** techniques. The classifier identifies whether a message is **spam (unwanted)** or **ham (legitimate)**.

We follow these stages:

* Collecting data from **Kaggle**
* Cleaning and preparing the dataset
* Exploratory Data Analysis (EDA)
* Text processing with **NLP techniques**
* Building and improving ML models
* Creating the project in **PyCharm**

**2. Data Collected from Kaggle**

We use the **SMS Spam Collection Dataset** from Kaggle.  
It contains two main columns:

* label → whether the message is spam or ham
* message → the actual SMS/email text

import pandas as pd

df = pd.read\_csv("spam.csv", encoding="latin1")

df.sample(5)

🔎 **Explanation**:

* import pandas as pd → loads **Pandas** for data handling.
* pd.read\_csv("spam.csv", encoding="latin-1") → reads the dataset file. Encoding is set to latin-1 because the dataset contains special characters.
* df.sample(5) → randomly shows 5 rows to quickly check the data.

**3. Data Cleaning**

The dataset contains extra **unnamed columns** that we don’t need.

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

df.rename(columns={'v1':'target','v2':'text'}, inplace=True)

df.sample(5)

🔎 **Explanation**:

* [['v1','v2']] → selects only the two useful columns.
* rename() → renames columns:
  + v1 → target (spam/ham labels)
  + v2 → text (message body)
* sample(5) → shows the sample 5 rows for confirmation.

Next, we convert the target labels (ham, spam) into numeric values.

from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()

df['target'] = encoder.fit\_transform(df['target'])

🔎 **Explanation**:

* LabelEncoder() → converts categorical labels into numbers.
* Here:
  + ham → 0
  + spam → 1

Checked for missing values in the dataset using df.isnull().sum(). The results indicated that there were no missing values in either of the columns (target and text), which allowed us to proceed without any imputation.

Next, we checked for duplicate rows with df.duplicated().sum(). The dataset contained 403 duplicate entries. To ensure data quality and prevent bias in the model, we removed these duplicates using df.drop\_duplicates(keep='first'). After this step, the dataset was free of duplicate rows (df.duplicated().sum() = 0).

**4. Exploratory Data Analysis (EDA)**

Check the class balance:

df['target'].value\_counts()

🔎 **Explanation**:

* This shows how many ham vs spam messages exist.
* Important because imbalanced data can bias the model.

Visualizing:

import matplotlib.pyplot as plt

plt.pie(df['target'].value\_counts(),labels=['ham','spam'],autopct="%0.2f%%")

plt.show()

🔎 **Explanation**:

A pie chart of the target variable shows that the dataset is highly imbalanced: **87.37% of messages are ham (non-spam), while only 12.63% are spam**. This imbalance suggests that the model could be biased toward predicting ham, and special care, such as resampling techniques or class-weight adjustments, may be needed to ensure accurate detection of spam messages.

**5. Text Processing (NLP)**

Step 1: Importing Libraries and Downloading NLTK Resources

import nltk

!pip install nltk

nltk.download('punkt')

nltk.download('stopwords')

**Explanation:**

* nltk is a powerful library for natural language processing.
* punkt is used for tokenizing text into words and sentences.
* stopwords contains common words like "is", "the", which we often remove in NLP preprocessing.
* These resources are necessary for tokenization and text cleaning.

Step 2: Extracting Basic Text Features

df['num\_characters'] = df['text'].apply(len)

df['num\_words'] = df['text'].apply(lambda x: len(nltk.word\_tokenize(x)))

df['num\_sentances'] = df['text'].apply(lambda x: len(nltk.sent\_tokenize(x)))

**Explanation:**

* **num\_characters**: Counts total characters in a message.
* **num\_words**: Counts total words using NLTK tokenization.
* **num\_sentances**: Counts total sentences using NLTK sentence tokenizer.

**Purpose:**

* These features help understand message length and structure.
* Spam messages are often longer or have more words/sentences than ham messages.

Step 3: Analyzing Feature Statistics

df[['num\_characters','num\_words','num\_sentances']].describe()

df[df['target']==0][['num\_characters','num\_words','num\_sentances']].describe()

df[df['target']==1][['num\_characters','num\_words','num\_sentances']].describe()

**Explanation:**

* Summarizes the statistics (mean, min, max, percentiles) for all messages and by class (ham=0, spam=1).
* **Observation:** Spam messages generally have higher character and word counts compared to ham.

Step 4: Visualizing Text Features

import seaborn as sns

import matplotlib.pyplot as plt

# Histogram of characters

plt.figure(figsize=(12,6))

sns.histplot(df[df['target']==0]['num\_characters'])

sns.histplot(df[df['target']==1]['num\_characters'], color='red')

# Histogram of words

plt.figure(figsize=(12,6))

sns.histplot(df[df['target']==0]['num\_words'])

sns.histplot(df[df['target']==1]['num\_words'], color='red')

# Pairplot for numerical features

sns.pairplot(df, hue='target')

# Correlation heatmap

numeric\_df = df.select\_dtypes(include=['number'])

plt.figure(figsize=(6,4))

sns.heatmap(numeric\_df.corr(), annot=True, cmap="coolwarm")

plt.show()

**Explanation:**

* **Histograms:** Compare distributions of message length for spam and ham.
* **Pairplot:** Visualizes relationships among numeric features, colored by target.
* **Heatmap:** Shows correlations between numeric features, which can help in feature selection.

Step 5: Text Cleaning & Transformation

from nltk.corpus import stopwords

import string

from nltk.stem.porter import PorterStemmer

ps = PorterStemmer()

def transform\_text(text):

# Lowercase

text = text.lower()

# Tokenize

text = nltk.word\_tokenize(text)

# Remove non-alphanumeric characters

y = [i for i in text if i.isalnum()]

text = y[:]

y.clear()

# Remove stopwords and punctuation

y = [i for i in text if i not in stopwords.words('english') and i not in string.punctuation]

text = y[:]

y.clear()

# Apply stemming

y = [ps.stem(i) for i in text]

return " ".join(y)

df['transformed\_text'] = df['text'].apply(transform\_text)

**Explanation:**

1. Converts text to **lowercase** to maintain consistency.
2. **Tokenizes** text into words for processing.
3. Removes **non-alphanumeric characters** to eliminate numbers and symbols.
4. Removes **stopwords** and **punctuation**, keeping only meaningful words.
5. Applies **stemming** to reduce words to their root form (dancing → danc).
6. Stores cleaned text in a new column transformed\_text.

Step 6: WordCloud Visualization

!pip install wordcloud

from wordcloud import WordCloud

wc = WordCloud(width=500, height=500, min\_font\_size=10, background\_color='white')

# Spam WordCloud

spam\_wc = wc.generate(df[df['target']==1]['transformed\_text'].str.cat(sep=" "))

plt.figure(figsize=(15,6))

plt.imshow(spam\_wc)

# Ham WordCloud

ham\_wc = wc.generate(df[df['target']==0]['transformed\_text'].str.cat(sep=" "))

plt.figure(figsize=(15,6))

plt.imshow(ham\_wc)

**Explanation:**

* Combines all transformed words for spam and ham messages.
* Creates visual **WordClouds** highlighting the most frequent words in each category.
* Provides intuitive insights into the language used in spam vs ham.

Step 7: Most Frequent Words Analysis

from collections import Counter

import pandas as pd

import seaborn as sns

# Spam corpus

spam\_corpus = [word for msg in df[df['target']==1]['transformed\_text'].tolist() for word in msg.split()]

pd.DataFrame(Counter(spam\_corpus).most\_common(30))

sns.barplot(x=pd.DataFrame(Counter(spam\_corpus).most\_common(30))[0],

y=pd.DataFrame(Counter(spam\_corpus).most\_common(30))[1])

plt.xticks(rotation='vertical')

plt.show()

# Ham corpus

ham\_corpus = [word for msg in df[df['target']==0]['transformed\_text'].tolist() for word in msg.split()]

pd.DataFrame(Counter(ham\_corpus).most\_common(30))

sns.barplot(x=pd.DataFrame(Counter(ham\_corpus).most\_common(30))[0],

y=pd.DataFrame(Counter(ham\_corpus).most\_common(30))[1])

plt.xticks(rotation='vertical')

plt.show()

**Explanation:**

* Creates separate lists of all words in spam and ham messages (corpora).
* Uses Counter to count word frequency.
* Visualizes the **top 30 most common words** in each category using bar plots.
* This highlights distinguishing words that can help the ML model classify spam vs ham.

✅ **Summary:**

* Extracted text-based numeric features.
* Cleaned and preprocessed text (lowercase, tokenization, stopword removal, stemming).
* Visualized word frequency and patterns for spam and ham.
* Created features and insights that will be fed into the machine learning model.

**6. Model Building**

Bag-of-Words (CountVectorizer)

from sklearn.feature\_extraction.text import CountVectorizer

cv = CountVectorizer()

X = cv.fit\_transform(df['transformed\_text']).toarray()

X.shape # -> (5169, 6708)

* CountVectorizer() converts the corpus into a document-term matrix where each column is a word in the vocabulary and each cell is the **count** of that word in the document.
* fit\_transform(...) learns the vocabulary from transformed\_text and returns the counts for every document.
* .toarray() converts the sparse matrix to a dense NumPy array (note: for large datasets this can use a lot of memory — prefer keeping it sparse unless you need dense arrays for a specific classifier).
* X.shape = (5169, 6708) means 5,169 messages and 6,708 distinct tokens/features.

y = df['target'].values

y.shape # -> (5169,)

* y is the target vector (0: ham, 1: spam). Shape 5169 matches number of samples.

Train / test split

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=2)

* Splits data into training and test sets. test\_size=0.2 → ~20% test (≈1,034 samples), ~80% train (≈4,135).
* random\_state=2 fixes the random seed so the split is reproducible.

Instantiate Naive Bayes variants and evaluation metrics

from sklearn.naive\_bayes import GaussianNB, MultinomialNB, BernoulliNB

from sklearn.metrics import accuracy\_score, confusion\_matrix, precision\_score

gnb = GaussianNB()

mnb = MultinomialNB()

bnb = BernoulliNB()

* GaussianNB assumes features follow a normal distribution (works best with continuous, real-valued features).
* MultinomialNB is designed for **count** data (word counts) — usually a good fit for CountVectorizer.
* BernoulliNB models binary features (word present/absent) — sometimes better if presence matters more than count.

Fit GaussianNB, evaluate

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))

* fit trains the classifier on training data. predict returns class labels for test set.
* accuracy\_score: (TP + TN) / total.
* confusion\_matrix: by default for binary labels [0,1] returns [[TN, FP], [FN, TP]].
  + Example output you had: [[792 104], [20 118]] → TN=792, FP=104, FN=20, TP=118.
* precision\_score = TP / (TP + FP); with that matrix precision = 118/(118+104) ≈ 0.5315 — many false positives (ham predicted as spam).
* **Interpretation:** GaussianNB performs poorly here because BoW integer counts are not well modeled by a Gaussian.

Fit MultinomialNB, evaluate

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))

* Example confusion matrix: [[871, 25], [12, 126]] → FP=25, FN=12, TP=126.
* Precision = 126/(126+25) ≈ 0.8344. Much better than GaussianNB — expected because MultinomialNB matches count data.

Fit BernoulliNB, evaluate

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))

* Example confusion matrix: [[893, 3], [28, 110]] → FP=3, FN=28, TP=110.
* Precision = 110/(110+3) ≈ 0.9734 — *very* high precision because very few false positives.
* **Trade-off noticed:** BernoulliNB reduced false positives (good precision) but increased false negatives (FN=28) compared to MultinomialNB (FN=12). That lowers recall (ability to find all spam).

**Key takeaway from the three NB runs:**

* MultinomialNB and BernoulliNB outperform GaussianNB for raw counts.
* BernoulliNB prioritized avoiding false positives (high precision) while MultinomialNB had a better balance (higher TP, fewer FN).

TF-IDF vectorization

from sklearn.feature\_extraction.text import TfidfVectorizer

tfidf = TfidfVectorizer(max\_features=3000)

X\_tf = tfidf.fit\_transform(df['transformed\_text']).toarray()

X\_tf.shape # -> (5169, 3000)

* TfidfVectorizer converts text to TF-IDF weighted features: it downweights words that are frequent across documents and upweights words that are more discriminative.
* max\_features=3000 restricts vocabulary to top 3,000 features (dimensionality reduction).
* .toarray() converts from sparse to dense. Again, consider keeping sparse for memory efficiency.

X\_tf\_train, X\_tf\_test, y\_train, y\_test = train\_test\_split(X\_tf, y, test\_size=0.2, random\_state=2)

* Same split parameters but now on TF-IDF features.

Re-run Naive Bayes on TF-IDF

You ran GaussianNB, MultinomialNB and BernoulliNB again on the TF-IDF features and printed metrics.

* **GaussianNB**: still worse on text features (precision ~0.507).
* **MultinomialNB**: example confusion [[896, 0], [30, 108]] → FP=0, TP=108, FN=30, precision 108/(108+0) = 1.0. Perfect precision here means the classifier did not produce any false positives on the test set — but it missed 30 spam messages (FN), so recall is lower.
* **BernoulliNB**: [[895, 1], [16, 122]] → precision ≈ 0.9919; a high-precision & higher recall than MNB in this run.

**Interpretation:** TF-IDF changed the signal; either MultinomialNB or BernoulliNB can perform extremely well depending on the vectorization and feature selection. Exact numbers can change with random splits or different max\_features.

Try many classifiers (scikit-learn + XGBoost)

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, AdaBoostClassifier, BaggingClassifier, ExtraTreesClassifier, GradientBoostingClassifier

from xgboost import XGBClassifier

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)

* Instantiated many commonly used classifiers with chosen hyperparameters. Brief notes:
  + SVC(kernel='sigmoid', gamma=1.0): sigmoid kernel behaves like a 2-layer neural activation; uncommon default — RBF is more typical.
  + KNeighborsClassifier() default uses k=5 neighbors.
  + DecisionTree(max\_depth=5) constrains tree depth to avoid overfitting.
  + Ensemble methods (RF, ETC, AdaBoost, Bagging, GBDT, XGBoost) are robust and often top performers on text after vectorization.

Put classifiers in a dictionary

clfs = {

'SVC' : svc,

'KN' : knc,

'NB': mnb,

'DT': dtc,

'LR': lrc,

'RF': rfc,

'AdaBoost': abc,

'BgC': bc,

'ETC': etc,

'GBDT': gbdt,

'xgb': xgb

}

* clfs maps short names to classifier objects for iteration.

Helper function to train & evaluate

def train\_classifier(clf, X\_tf\_train, y\_train, X\_tf\_test, y\_test):

clf.fit(X\_tf\_train, y\_train)

y\_pred = clf.predict(X\_tf\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

precision = precision\_score(y\_test, y\_pred)

return accuracy, precision

* This function standardizes the training + evaluation procedure: fit, predict, compute accuracy and precision, and return them.

Train every classifier and collect results

accuracy\_scores = []

precision\_scores = []

for name, clf in clfs.items():

current\_accuracy, current\_precision = train\_classifier(clf, X\_tf\_train, y\_train, X\_tf\_test, y\_test)

print("For ", name)

print("Accuracy - ", current\_accuracy)

print("Precision - ", current\_precision)

accuracy\_scores.append(current\_accuracy)

precision\_scores.append(current\_precision)

* Iterates over all classifiers, trains them on the TF-IDF train set and evaluates on test set.
* Prints accuracy and precision for each classifier and appends scores to lists for later comparison.

**Interpreting example printed results:**

* SVC and RF show very high accuracy (~0.9758) and high precision (~0.9748–0.9829).
* KN and NB can show precision=1.0 — meaning no false positives on this test split

Create a DataFrame of results and sort

performance\_df = pd.DataFrame({'Algorithm': clfs.keys(), 'Accuracy': accuracy\_scores, 'Precision': precision\_scores}).sort\_values('Precision', ascending=False)

* performance\_df holds the summary of algorithms and metrics, sorted by Precision descending so the highest precision models appear first.

Reshape for plotting and plot

performance\_df1 = pd.melt(performance\_df, id\_vars="Algorithm")

sns.catplot(x='Algorithm', y='value', hue='variable', data=performance\_df1, kind='bar', height=5)

plt.ylim(0.5,1.0)

plt.xticks(rotation='vertical')

plt.show()

* pd.melt(...) converts the wide performance\_df to long format (Algorithm, variable [Accuracy/Precision], value) needed for seaborn grouped barplot.
* sns.catplot(..., kind='bar') draws bars comparing Accuracy and Precision for each algorithm.
* plt.ylim(0.5,1.0) sets the y-axis range so plot focuses on the high end (50%–100%).
* Rotating x-ticks ensures algorithm names are readable.

**7. Improving Model Performance**

TF-IDF (all features)

tfidf\_all = TfidfVectorizer() # no max\_features limit

* Creates a TF-IDF vectorizer that **uses the full vocabulary** (no max\_features cap). Use this when you want to compare "all features" vs a limited set.

X\_tf\_all = tfidf\_all.fit\_transform(df['transformed\_text']).toarray()

* Learns vocabulary from transformed\_text and computes TF-IDF features for every document.
* .toarray() converts the sparse matrix to a dense NumPy array — **warning**: dense arrays can use a lot of memory for large vocabularies. Prefer sparse matrices if memory is a concern.

X\_train\_all, X\_test\_all, y\_train, y\_test = train\_test\_split(

X\_tf\_all, y, test\_size=0.2, random\_state=2

)

* Splits the TF-IDF feature matrix into training and test sets (80% train, 20% test). random\_state=2 ensures reproducibility.

accuracy\_scores\_all = []

precision\_scores\_all = []

for name, clf in clfs.items():

acc, prec = train\_classifier(clf, X\_train\_all, y\_train, X\_test\_all, y\_test)

accuracy\_scores\_all.append(acc)

precision\_scores\_all.append(prec)

train\_classifier is your helper that fits a classifier and returns (accuracy, precision).

This loop trains and evaluates every classifier in clfs on the TF-IDF (all features) split, collecting accuracy and precision scores.

temp\_df = pd.DataFrame({

'Algorithm': clfs.keys(),

'Accuracy\_all\_features': accuracy\_scores\_all,

'Precision\_all\_features': precision\_scores\_all

}).sort\_values('Precision\_all\_features', ascending=False)

Creates a DataFrame summarizing results for the "all features" run and sorts by precision descending.

new\_df = performance\_df.merge(temp\_df,on='Algorithm')

Merges performance\_df (your baseline, here TF-IDF with max\_features=3000) with temp\_df (TF-IDF all features) on the Algorithm column, so you can directly compare Accuracy/Precision before vs after removing the max\_features limit.

**Interpretation of your printed new\_df:**

* Columns: Algorithm | Accuracy | Precision | Accuracy\_all\_features | Precision\_all\_features
* Example observations (from your numbers):
  + NB (MultinomialNB): accuracy decreased from 0.970986 → 0.959381 when using all features (so restricting to top-3000 helped NB in that run).
  + RF: precision increased to 1.000000 on all features (could be due to test split variation).
  + xgb: accuracy increased on all features (0.967118 → 0.974855), indicating XGBoost may benefit from the larger vocabulary.
* **Important:** Small differences can be due to randomness or overfitting to the single test split.

Voting classifier (soft voting)

svc = SVC(kernel='sigmoid', gamma=1.0, probability=True)

mnb = MultinomialNB()

etc = ExtraTreesClassifier(n\_estimators=50, random\_state=2)

* Instantiates three base estimators. probability=True is required for SVC so it can provide probabilities for **soft voting**.

from sklearn.ensemble import VotingClassifier

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

* Creates a VotingClassifier that averages class **probabilities** across base learners and predicts the class with the highest average probability.

voting.fit(X\_tf\_train, y\_train)

y\_pred = voting.predict(X\_tf\_test)

print("Accuracy", accuracy\_score(y\_test, y\_pred))

print("Precision", precision\_score(y\_test, y\_pred))

# => Accuracy 0.9816247582205029

# Precision 0.9917355371900827

* Fits the voting ensemble on your training TF-IDF split and evaluates on the test set.
* Reported results show high accuracy and **very high precision** (few false positives).

**Why soft voting helps:** combining different model families (SVM, NB, trees) often captures complementary behavior and stabilizes predictions.

Stacking classifier

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

final\_estimator=RandomForestClassifier()

from sklearn.ensemble import StackingClassifier

clf = StackingClassifier(estimators=estimators, final\_estimator=final\_estimator)

* Defines a stacking ensemble: base learners generate meta-features (their predictions or probabilities), and the final\_estimator (a Random Forest) learns to combine them.

clf.fit(X\_tf\_train, y\_train)

y\_pred = clf.predict(X\_tf\_test)

print("Accuracy", accuracy\_score(y\_test, y\_pred))

print("Precision", precision\_score(y\_test, y\_pred))

# => Accuracy 0.9806576402321083

# Precision 0.946969696969697

* Fits stacking and evaluates. In your run stacking delivered slightly lower precision than voting; stacking can sometimes increase overall accuracy but reduce precision depending on how the meta-learner combines base outputs.

**Tip:** Stacking uses out-of-fold predictions for meta-training by default — this reduces leakage but still benefits from careful cross-validation and meta-model tuning.

Model persistence (save vectorizer + model)

import joblib

joblib.dump(vectorizer, "vectorizer.pkl")

joblib.dump(model, "model.pkl")

**8. Creating the Project in PyCharm**

1. To requirement.txt install – First open anaconda promp, write “where python”.

To change path to specific folder - cd /d "D:\Documents\Topmentor\Projects\3. SMS Spam Classifier"

Using **conda:**

conda create -n sms\_env python=3.11

conda activate sms\_env

Then write ‘pip freeze > requirements.txt’

Created requirement.txt file as per below details-

fastapi==0.116.1

numpy==2.3.3

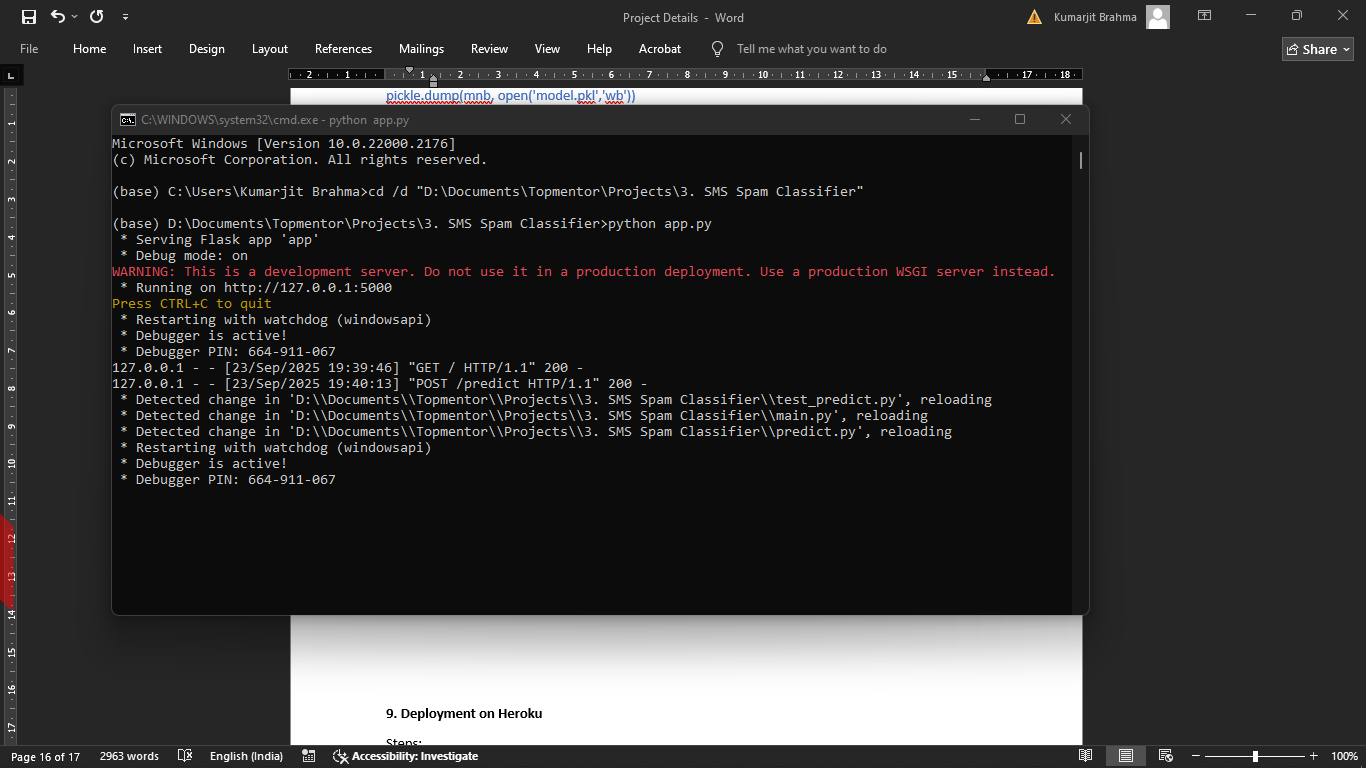
pandas==2.3.2

pydantic==2.11.9

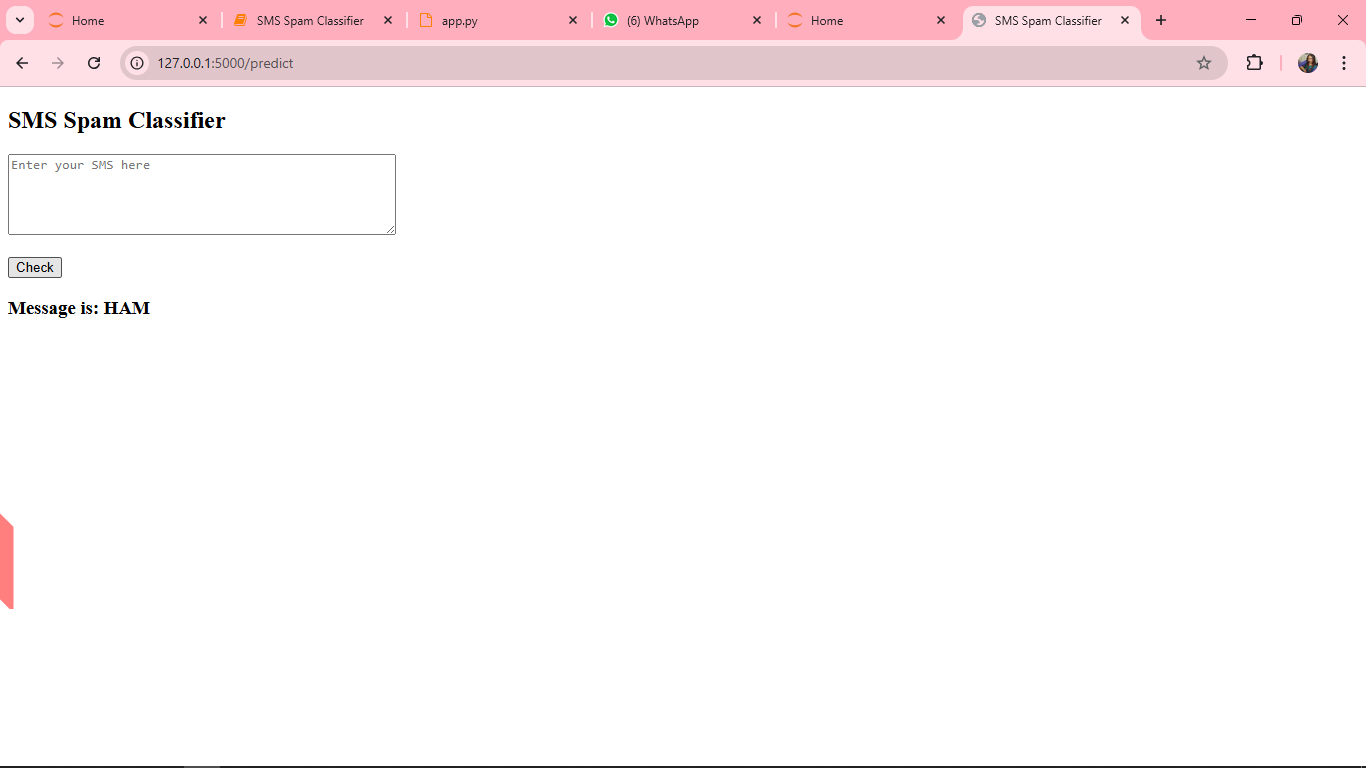
1. create app.py – we can create it on Jupiter notebook or in pycharm, codes are written
2. HTML file create – we can create it from Pycharm
3. Now Run the app.py – we can run in pycharm or anaconda prompt

In conda prompt we have changed the path to project folder then write ‘python app.py’

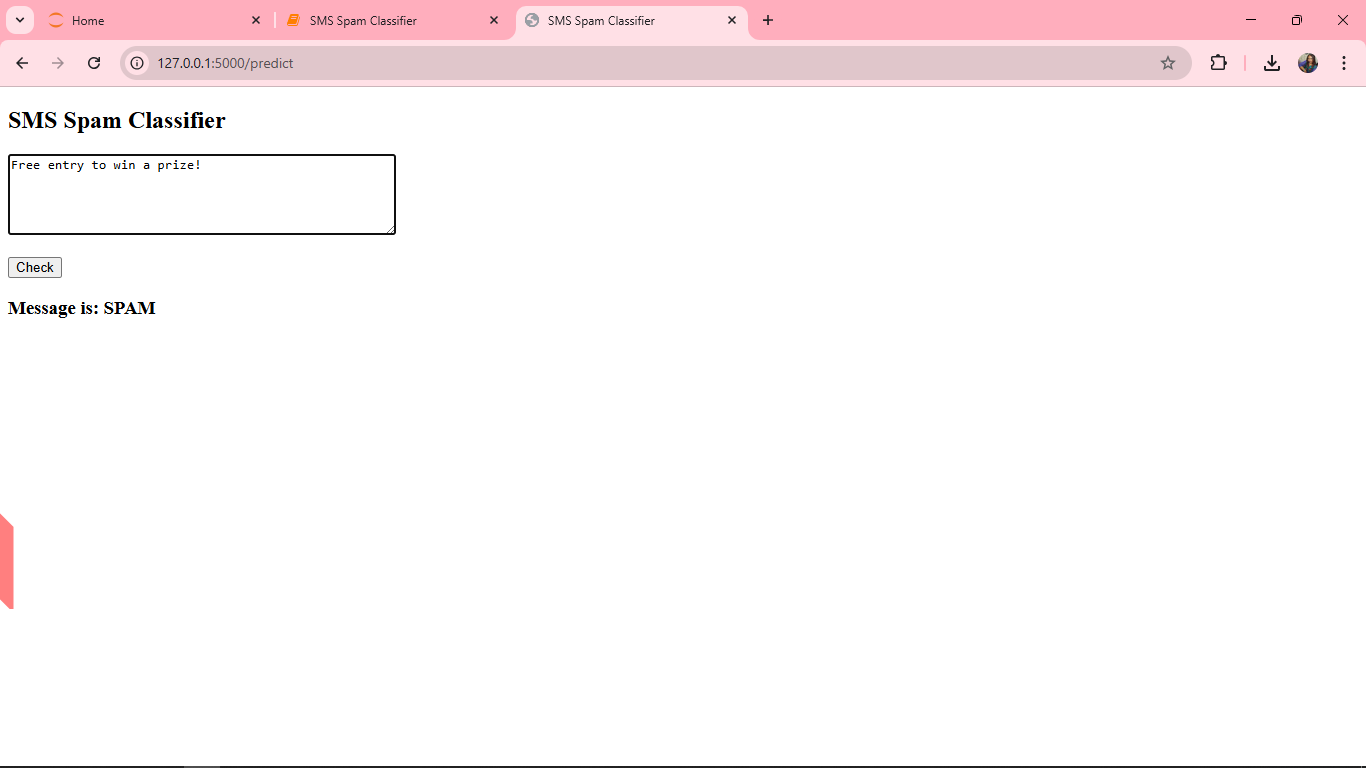
1. Return will be like-



1. Open your browser and go to:  
   👉 <http://127.0.0.1:5000>
2. Below screen will appear-



1. Enter the text ‘Free entry to win a prize!’, it will return as spam.



**9. Outro**

We successfully built and deployed an **SMS/Email Spam Classifier**.  
Key takeaways:

* NLP techniques (tokenization, stopwords removal, stemming) are crucial.
* TF-IDF vectorization helps convert text into machine-readable format.
* Naive Bayes is a strong baseline model for spam detection.
* Deployment on **Heroku** makes the model accessible to users worldwide.

This project demonstrates **end-to-end ML workflow**:  
📊 Data → 🔍 Cleaning & EDA → 🧠 Model → 🚀 Deployment