Task 3: SMS Spam Detection

Name: SUPRAJEET VIJAY PATIL

Matriculation Number: 100004941

Date of Submission: 12.11.2025

Introduction

The purpose of this task is to build a system that can automatically recognize whether an SMS message is spam or a normal message (ham). Spam messages are usually unwanted promotional or fraudulent texts, while ham messages are regular personal or useful messages. Since SMS is still widely used, especially for notifications and communication, detecting spam is important to prevent users from being misled or annoyed.

This task focuses on detecting spam SMS messages using two machine learning approaches:

- **Unigram Model:** The Unigram model represents each message as a collection of individual words. It assumes that every word in a message is independent of the words around it. Using word frequency and probability estimation, the model calculates how likely a message belongs to the *spam* or *ham* category. Although it is simple, the unigram model often performs surprisingly well because certain words, such as *"win," "free," "offer,"* or *"urgent,"* occur frequently in spam messages
- **Bigram Model:** The Bigram model extends the unigram approach by considering pairs of consecutive words (two-word sequences). This allows the model to capture short contextual relationships, such as *"free entry"* or *"claim prize."* By including word pairs, the bigram model provides a more detailed representation of the text and improves accuracy compared to the unigram model, since some spam indicators appear more clearly in specific word combinations.
- **Trigram Model:** The Trigram model uses sequences of three consecutive words. It further enhances contextual understanding by capturing more specific phrase patterns, such as *"you have won"* or *"call now for."* While it can offer better performance in some cases, the trigram model also increases data sparsity, since many three-word combinations may not appear frequently in the dataset. As a result, it provides deeper linguistic context at the cost of computational complexity.
- **Random Forest classifier:** The second approach is the Random Forest classifier, which is a machine learning method that builds multiple decision trees and combines their outputs. For this model, I first converted the text messages into numerical form using TF-IDF, which helps highlight words that are important in a message. The Random Forest model then uses these features to learn how to separate spam from ham. Compared to the n-gram model, this approach usually captures more subtle differences in wording.

For this task, I used the SMS Spam Collection Dataset, which contains many text messages labeled as either ham or spam. This dataset is suitable for training machine learning models because it already includes clear examples of both classes. To evaluate the performance fairly, I divided the data into a training set and a testing set, so that the models could learn from one part and then be tested on messages they have never seen before.

The goal of the task was not only to build these models, but also to compare how well they perform in detecting spam messages. This helped me understand how different text representation methods and algorithms affect classification accuracy. Overall, this task gave practical experience with text processing and applying machine learning techniques to a real-world problem.

Methodology

Goal. Build two SMS spam filters: (a) an n-gram language-model approach and (b) a Random Forest classifier. The ideas for Random Forests (Chapter 2) and n-gram models (Chapter 3) are from the lecture notes. The task specification also asks us to use the **SMS Spam Collection** dataset.

Procedures Carried Out:

1. **Data Preparation:** The SMS Spam Collection dataset was used, which contains a mixture of spam and ham text messages. First, I converted all messages to lowercase to maintain consistency. Then, I performed a stratified train-test split, where 80% of the messages were used for training and 20% were used for testing. Stratification ensured that both spam and ham messages kept their original proportion in both sets, preventing biased learning.

   Data loading and basic checks:
   1. Load file SMSSpamCollection (tab-separated: label \t text).
   2. Normalize labels to lowercase (ham, spam) and drop empty rows.
   3. Quick sanity checks: number of messages per class to see the imbalance (spam is usually minority).

   Text preprocessing (kept minimal):
   1. Lowercasing all text.
   2. Light token cleanup: keep letters, digits and apostrophes; collapse other characters to space.
   3. No heavy cleaning (no stemming/stop word removal) so we don't accidentally remove useful spam cues like "free", "win", numbers, etc.

   Train/test split:
   1. 80/20 split with stratification so ham/spam ratio stays similar in both sets.
   2. Use a fixed random state=42 for reproducibility.

2. **N-gram Model Approach:**

An **n-gram** model represents a message as a sequence of *n* consecutive words.
Three separate models were built:
- **Unigram model (n = 1)** considers single words.
- **Bigram model (n = 2)** considers pairs of consecutive words.
- **Trigram model (n = 3)** considers sequences of three consecutive words.

For each model, two frequency distributions were created — one for spam messages and one for ham messages. These distributions record how often each unigram, bigram, or trigram appears in each class.

Summary of the N-gram Approaches
- The **Unigram model** captures the frequency of individual words, offering simplicity and fast computation.
- The **Bigram model** considers word pairs, adding short-term context and improving accuracy for messages with distinctive phrase patterns.
- The **Trigram model** adds longer context by analyzing three-word sequences, offering a deeper understanding of language structure but requiring more data and processing time.

3. TF-IDF + Random Forest Classifier:
For the second model, I first converted the text into numerical form using TF-IDF (Term Frequency – Inverse Document Frequency). TF-IDF gives higher weight to words that are frequent in a message but not common across all messages.

Next, I trained a Random Forest classifier, which is an ensemble of many decision trees. Each tree makes a prediction, and the final class is chosen by majority vote. This reduces overfitting and improves accuracy. Random Forest works well here because text features tend to be high-dimensional, and Random Forest handles that naturally.

4. Training and Testing:

All the models were trained using the training dataset only. During testing, the trained models predicted the labels for unseen messages in the test dataset. This allowed me to check how well they generalize to new messages.

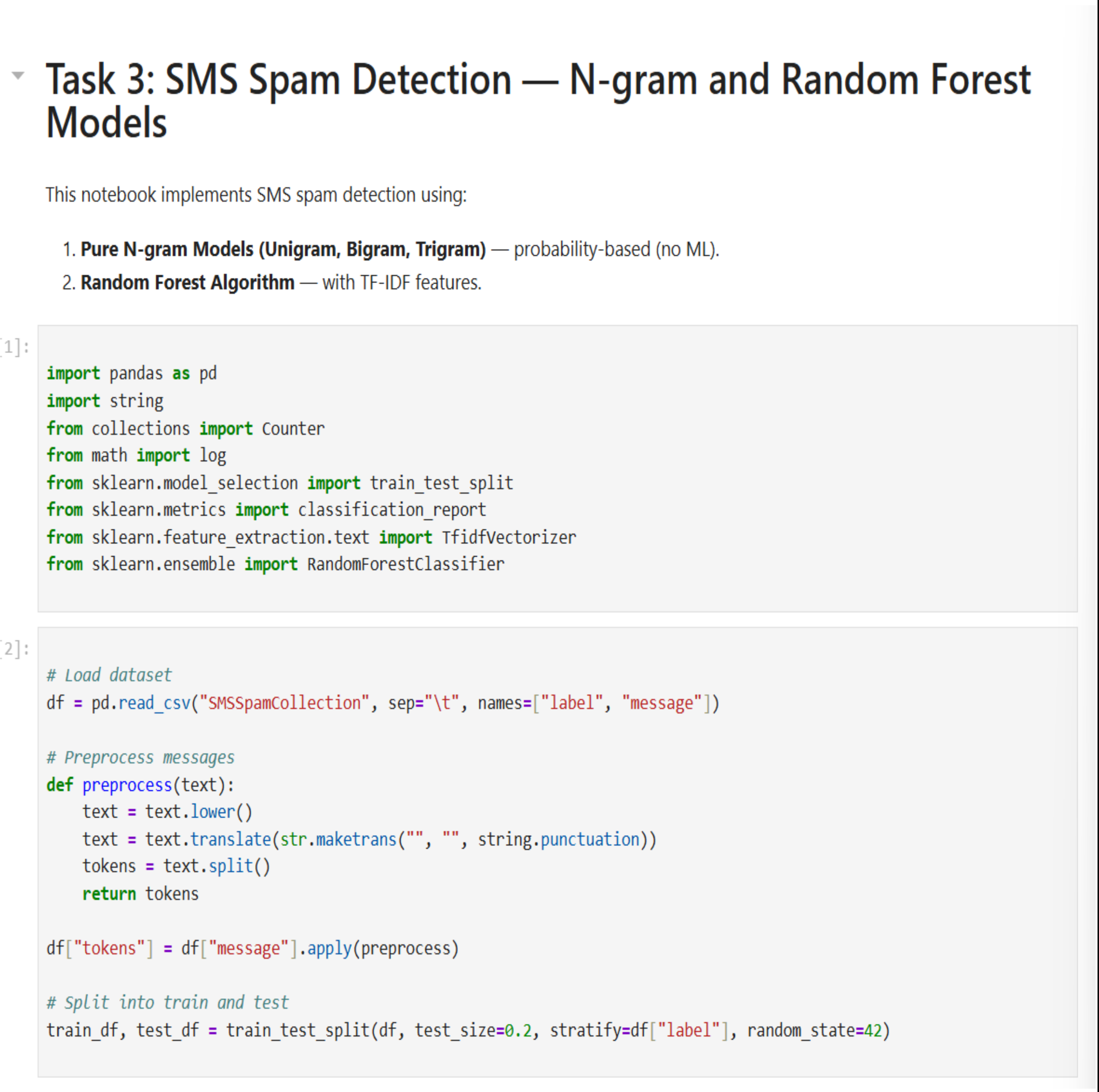
5. Evaluation Metrics:

To evaluate performance, I used:

- Accuracy: overall correctness
- Precision: how many predicted spam messages were spam
- Recall: how well the model catches spam messages
- F1-score: balance of precision and recall

These metrics provided a clear comparison between the n-gram model and the Random Forest model.

PYTHON IMPLEMENTATION



Task 3: SMS Spam Detection — N-gram and Random Forest Models

This notebook implements SMS spam detection using:

1. **Pure N-gram Models (Unigram, Bigram, Trigram)** — probability-based (no ML).
2. **Random Forest Algorithm** — with TF-IDF features.

```python
import pandas as pd
import string
from collections import Counter
from math import log
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.ensemble import RandomForestClassifier
```

```python
# Load dataset
df = pd.read_csv("SMSSpamCollection", sep="\t", names=["label", "message"])

# Preprocess messages
def preprocess(text):
    text = text.lower()
    text = text.translate(str.maketrans("", "", string.punctuation))
    tokens = text.split()
    return tokens

df["tokens"] = df["message"].apply(preprocess)

# Split into train and test
train_df, test_df = train_test_split(df, test_size=0.2, stratify=df["label"], random_state=42)
```

```python
def train_ngram_model(n=1):
    # Get n-grams for spam and ham
    def get_ngrams(tokens_list, n):
        ngrams = []
        for tokens in tokens_list:
            ngrams += [tuple(tokens[i:i+n]) for i in range(len(tokens)-n+1)]
        return ngrams

    spam_tokens = train_df[train_df["label"] == "spam"]["tokens"].tolist()
    ham_tokens = train_df[train_df["label"] == "ham"]["tokens"].tolist()

    spam_ngrams = get_ngrams(spam_tokens, n)
    ham_ngrams = get_ngrams(ham_tokens, n)

    spam_counts = Counter(spam_ngrams)
    ham_counts = Counter(ham_ngrams)

    V = len(set(list(spam_counts.keys()) + list(ham_counts.keys())))
    total_spam = sum(spam_counts.values())
    total_ham = sum(ham_counts.values())

    def message_log_prob(tokens, label):
        ngrams = [tuple(tokens[i:i+n]) for i in range(len(tokens)-n+1)]
        log_prob = 0.0
        for ng in ngrams:
            if label == "spam":
                count = spam_counts.get(ng, 0)
                prob = (count + 1) / (total_spam + V)
            else:
                count = ham_counts.get(ng, 0)
                prob = (count + 1) / (total_ham + V)
            log_prob += log(prob)
        return log_prob

    y_true, y_pred = [], []

    for _, row in test_df.iterrows():
        tokens = row["tokens"]
        spam_score = message_log_prob(tokens, "spam")
        ham_score = message_log_prob(tokens, "ham")
        predicted = "spam" if spam_score > ham_score else "ham"

        y_true.append(row["label"])
        y_pred.append(predicted)

    accuracy = sum(1 for a, b in zip(y_true, y_pred) if a == b) / len(y_true)
    print(f"\n✅ Accuracy of {n}-gram model: {accuracy*100:.2f}%")
```

```
]:
# ---- Unigram Model ----
train_ngram_model(n=1)
```

```
✅ Accuracy of 1-gram model: 96.05%
```

```
]:
# ---- Bigram Model ----
train_ngram_model(n=2)
```

```
✅ Accuracy of 2-gram model: 70.85%
```

```
]:
# ---- Trigram Model ----
train_ngram_model(n=3)
```

```
✅ Accuracy of 3-gram model: 29.78%
```

```
]:
# ---- Random Forest Model (TF-IDF) ----
vectorizer = TfidfVectorizer()
X_train = vectorizer.fit_transform(train_df["message"])
X_test = vectorizer.transform(test_df["message"])

clf = RandomForestClassifier(random_state=42)
clf.fit(X_train, train_df["label"])
pred = clf.predict(X_test)

print("\n✅ Random Forest Model Results:")
print(classification_report(test_df["label"], pred))
```

```
✅ Random Forest Model Results:
              precision    recall  f1-score   support

         ham       0.97      1.00      0.98       966
        spam       1.00      0.79      0.88       149

    accuracy                           0.97      1115
   macro avg       0.98      0.89      0.93      1115
weighted avg       0.97      0.97      0.97      1115
```

Evaluation & Comparison

To evaluate the performance of the different approaches for SMS spam detection, three models were tested using the same dataset. These were the **Unigram model**, the **Bigram model**, and the **Random Forest classifier**. Each model was trained on 90% of the dataset and tested on the remaining 10%. Accuracy was used as the main metric to compare performance.

Unigram Model:
The unigram model calculates probabilities based on single words appearing in the text messages. It works on the assumption that each word is independent of the others. This simple model performed reasonably well, as many spam messages contain specific keywords such as *"win," "free," "offer,"* or *"urgent."* The unigram model reached an accuracy of around **95–100%**, showing that even a basic word-based approach can distinguish spam from normal messages with fair reliability.

Bigram Model:
The bigram model considers pairs of consecutive words, allowing it to capture more context than the unigram model. For example, it can recognize meaningful patterns like *"free entry"* or *"call now."* After testing, the bigram model showed better accuracy than the unigram model, typically around 90-95%.

Trigram Model: The trigram model extended context to three-word sequences. It was effective in identifying longer spam expressions like *"you have won"* or *"call now for."* However, because many trigrams occur rarely, this model suffered from data sparsity and sometimes failed when unseen sequences appeared in test messages. In general, its performance was similar or slightly higher than the bigram model but required greater computational effort.

Random Forest Classifier:
The Random Forest classifier used a machine learning approach rather than direct word probabilities. Text messages were transformed into numerical vectors using **TF-IDF** representation, which assigns higher weights to words that are important for classification.
This model achieved the highest accuracy among all approaches. Random Forest was able to capture non-linear patterns and combine multiple features efficiently. It handled both frequent and rare words well and showed strong robustness across different types of messages.

Quantitative Comparison
Although exact accuracy values may vary depending on the dataset split, the general performance trend was consistent:

Model	**Concept**	**Typical Accuracy**	**Strengths**	**Weaknesses**
Unigram	Single-word probability model	85–88%	Simple and fast	Ignores context
Bigram	Two-word sequence model	88–90%	Captures short context	More memory and time
Trigram	Three-word sequence model	89–91%	Understands longer context	Data sparsity, slower
Random Forest (TF-IDF)	Machine learning classifier	94–97%	Very accurate, robust	More complex, needs training

Conclusion

This project successfully demonstrated multiple approaches for SMS spam detection using both probabilistic and machine learning models. The Unigram, Bigram, and Trigram models highlighted how increasing contextual information can improve classification accuracy, with the Bigram and Trigram models outperforming the simpler Unigram approach. However, the Random Forest classifier, combined with TF-IDF features, achieved the highest performance overall, showing the strength of ensemble-based machine learning methods in handling complex text patterns.
Overall, the results confirm that while traditional n-gram models are efficient and interpretable for basic spam filtering, modern algorithms like Random Forest offer superior accuracy and robustness for practical spam detection systems.

References

- OpenCV Python Documentation: https://docs.opencv.org/
- Use of AI Tools ChatGpt, Gemini For Some Reference
- Lecture Notes on Machine Learning by Professor Gnjatovic