Solo Machine Learning Project
Applied ML on Real Dataset
SMS Spam Classification

Jacques Nahri

Contents

1. Objective

The main objective of this project is to dive deeper into the machine learning topic, as after covering the fundamentals of ML with the Titanic and Iris dataset, it is important to work with more interesting dataset with the real impact, as classifying spam messages can be also applied on classifying any text based on sentiments, tone, or subject, which might be valuable for 3VO to assess its user engagement and satisfaction on their platform, adding to this the ability to detect spam content on their platforms also. The below sections will depict the work flow done on two datasets one containing SMS and the second Emails, classified as spam or not. Further, the dataset are preprocessed, cleaned and well organized before training, validating and testing the required models.

2. Implementation

2.1. Datasets

Two Kaggle datasets were used in this workflow, as the first dataset mainly contains SMS English messages classified as spam or not, and the second dataset contains Emails also classified as spam or not, here below first few rows of the two datasets are shown.

Data head of SMS data:

	Category	Message
0	ham	Go until jurong point, crazy.. Available only ...
1	ham	Ok lar... Joking wif u oni...
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...
3	ham	U dun say so early hor... U c already then say...
4	ham	Nah I don't think he goes to usf, he lives aro...

Figure 1 SMS Dataset

Data head of Email data:

	text	spam
0	Subject: naturally irresistible your corporate...	1
1	Subject: the stock trading gunslinger fanny i...	1
2	Subject: unbelievable new homes made easy im ...	1
3	Subject: 4 color printing special request add...	1
4	Subject: do not have money , get software cds ...	1

Figure 2 Emails Dataset

2.2. Data Understanding

To better understand the datasets, we applied some basic functions from the Pandas library, as to understand the attributes data types, the distribution of the data, and whether or not the datasets contains NaN values for certain cells.

2.2.1. Data Profiling

An Explanatory Data Analysis report was generated automatically as an HTML file using the Profile Report package available in Python, so the report will depict all the details of the datasets and help us have a detailed understanding of the dataset structure. Here below we explained the various info provided by the HTML file for both datasets.

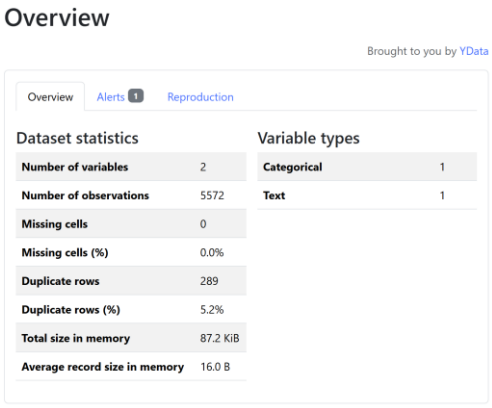


Figure 3 SMS Dataset Overview

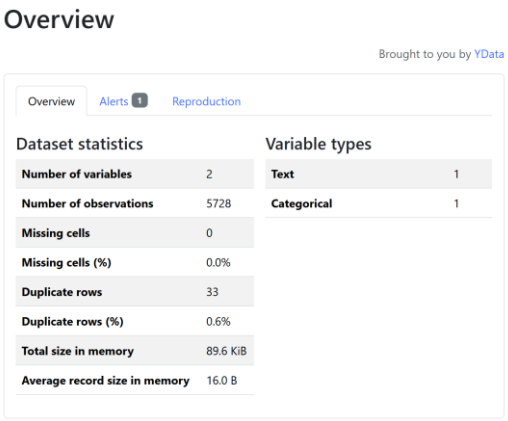


Figure 4 Emails Dataset Overview

As we can see the main section of the reports, showed us statistics of the datasets, that previously we used several functions to obtain it. And the fields that actually care about are the number of missing cells and duplicates rows, as we can see that both datasets have zero NaN values, and they actually suffer from duplicate rows which was expected in such kind of data where various SMS and Emails can have same content. So based on this we can understand that one essential step in preprocessing will be dropping the duplicate rows in the datasets.

2.3. Data Preprocessing

2.3.1. Drop Duplicate Rows

As first step before delving deeper into cleaning the data, we must first drop the duplicate rows from the datasets for better understanding of the classes distribution between spam and not spam.

```
# remove duplicates from data_frame_1
data_frame_1 = data_frame_1.drop_duplicates()
print('Duplicate rows in SMS data are:', data_frame_1.duplicated().sum())

# remove duplicated from data_frame_2
data_frame_2 = data_frame_2.drop_duplicates()
print('Duplicate rows in Emails data are:', data_frame_2.duplicated().sum())
```

```
Duplicate rows in SMS data are: 0
Duplicate rows in Emails data are: 0
```

Figure 5 Dropping Duplicates

As we can see the count of duplicate rows in both datasets is 0 after applying the above code snippet.

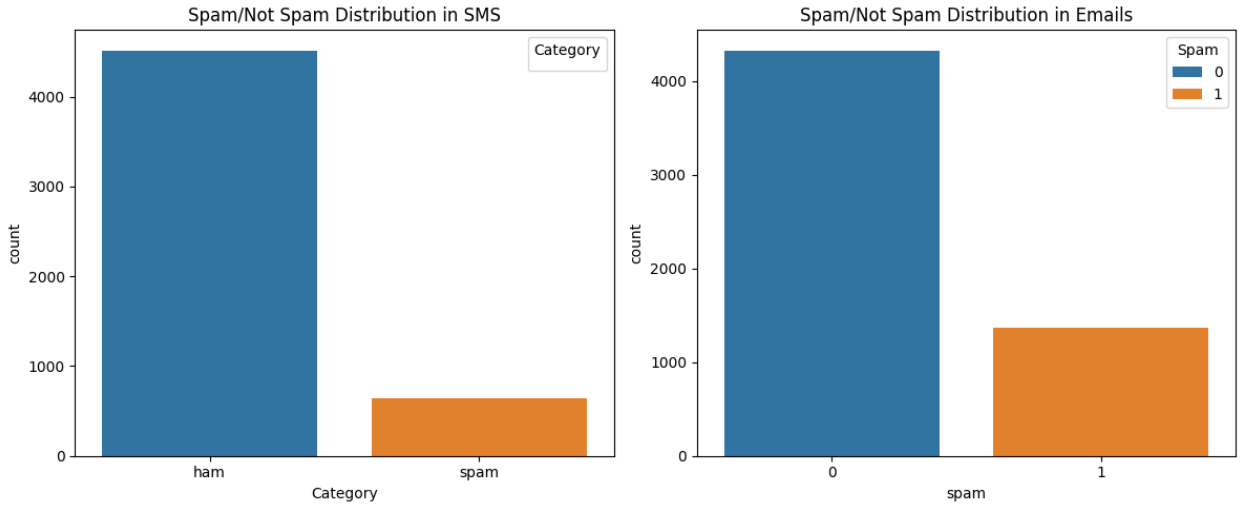
2.3.2. Data Distribution

An important aspect to check is how the data is distributed among classes in both datasets, because balanced data is essential in training and finetuning models and imbalanced data could lead to overfitting and wrong predictions during testing for the minority class.

```
print('SMS data distribution:')
print(data_frame_1['Category'].value_counts())
data_frame_1_imbalance_ratio = data_frame_1['Category'].value_counts()[0] / data_frame_1['Category'].value_counts()[1]
print('Imbalanace ratio:', data_frame_1_imbalance_ratio)

print('\nEmails data distribution:')
data_frame_2_imbalance_ratio = data_frame_2['spam'].value_counts()[0] / data_frame_2['spam'].value_counts()[1]
print(data_frame_2['spam'].value_counts())
print('Imbalance ratio:', data_frame_2_imbalance_ratio)
```

Figure 6 Data Distribution



It was noticed that in both datasets there is an unhealthy imbalance between classes distribution, hence it is essential to deal with this imbalance before any further processing.

2.3.3. Data Resampling and Feature Extraction

For data resampling there are many approaches, the basic approach is either to down sample the majority class in the dataset by removing entries or up sampling by minority class by adding duplicates rows, both approaches are not well suitable for training the model because down sampling will lead to data leakage and loosing valuable data and up sampling may lead to overfitting if the duplicates count is enormous. Hence another better approach must be followed which is similar to data augmentation. But before, we have to apply some modifications on the data to clean unnecessary symbols and text punctuation from the text attributes. For this purpose we defined several utility functions applied on the text field in both datasets. And note that we also renamed the columns in both datasets to 'text' and 'label' for consistency, mapped 0 as not spam and 1 as spam, and added a 'length' column to the datasets for better visualization of the effect of these utility functions.

The following pipeline depicts the modifications applied on the data before passing through the resampling part.

2.3.3.1. Data Cleaning Pipeline

The first step in this process involves importing the necessary natural language processing tools from the NLTK library. These include a tokenizer for splitting text into individual words, a lemmatizer to reduce words to their base or dictionary form, and predefined lists of stopwords. Required NLTK resources—such as the word tokenizer models, WordNet for lemmatization, and English stopword lists—are downloaded to ensure full functionality.

A sequence of cleaning functions is then defined to process the text. The first function handles emails specifically by removing the "Subject:" prefix, which appears in many email messages but provides no useful information for classification. Next, a normalization step converts all characters in the text to lowercase to ensure consistency, as machine learning models treat "Free" and "free" as distinct words if left unchanged.

Subsequent steps remove numerical characters and punctuation from the text, as these are generally not helpful in determining whether a message is spam. After this, stopwords are removed to reduce noise in the data, allowing the model to focus on more meaningful content. Additionally, any extra white spaces and isolated single characters are stripped from the text using regular expressions, ensuring a cleaner and more compact representation.

Finally, the lemmatization step reduces each word to its base form (e.g., "running" becomes "run"), further standardizing the vocabulary and reducing redundancy. These cleaning operations are applied systematically to both the SMS and email datasets.

Together, these steps ensure that the input data is clean, normalized, and well-prepared for feature extraction and model training. This preprocessing pipeline is essential for boosting classification accuracy, minimizing noise, and improving generalization across both SMS and email spam detection tasks

```python
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
nltk.download('punkt_tab')
nltk.download('wordnet')
nltk.download('stopwords')
import re
import string

# remove the 'Subject' from text in Emails data
def remove_subject_from_text_emails(text: str):
    if text.lower().startswith("subject:"):
        return text[len("subject:"):].strip()
    return text

# to lower
def to_lower(text: str):
  return text.lower()

# remove numbers from text
def remove_numbers(text: str):
  for c in text:
    if c.isnumeric():
      text = text.replace(c,' ')
  return text

# lemmatizing
def lemmatizing(text: str):
  lemmatizer = WordNetLemmatizer()
  tokens = word_tokenize(text)
  for i in range(len(tokens)):
    lemma_world = lemmatizer.lemmatize(tokens[i])
    tokens[i] = lemma_world
  return " ".join(tokens)

# remove punctuation
def remove_punctuation(text: str):
  for c in text:
    if c in string.punctuation:
      text = text.replace(c,'')
  return text

# remove stop words
def remove_stopwords(text: str):
  removed = []
  stop_words = list(stopwords.words("english"))
  tokens = word_tokenize(text)
  for i in range(len(tokens)):
    if tokens[i] not in stop_words:
      removed.append(tokens[i])
  return " ".join(removed)

# remove extra white spaces
def remove_extra_white_spaces(text: str):
  single_char_pattern = r'\s+[a-zA-Z]\s+'
  without_sc = re.sub(pattern=single_char_pattern, repl=" ", string= text)
  return without_sc
```

Figure 7 Data Cleaning Pipeline

2.3.3.2. Augment SMS and Emails training data and Extract Features

To avoid data leakage and that the data being understood and likely being all discovered before testing, a good approach is to split the data into training and testing parts before applying augmentation as to avoid this issue.

```python
from sklearn.model_selection import train_test_split
# split SMS data
X_train_SMS, X_test_SMS, y_train_SMS, y_test_SMS = train_test_split(data_frame_1['text'], data_frame_1['label'], test_size = 0.25,
                                                    stratify = data_frame_1['label'], random_state=42)

print('Distribution in SMS train data: ')
display(y_train_SMS.value_counts())

# split Emails data
X_train_Emails, X_test_Emails, y_train_Emails, y_test_Emails = train_test_split(data_frame_2['text'], data_frame_2['label'], test_size = 0.25,
                                                    stratify = data_frame_2['label'], random_state=42)

print('\nDistribution in Emails train data: ')
display(y_train_Emails.value_counts())
```

Figure 8 Datasets Split

After splitting the data we decided to use two different approaches for augmenting the data, so for SMS data we used the NLP AUG package in Python and for Emails data we used the SMOTE package. Each approach requires different workflow as discussed below.

2.3.3.2.1. Augment SMS data using NLPAUG

Using NLPAUG requires raw data as text, so we directly fed the augmenter the raw text SMS data to be augmented.

The code performs **data augmentation** on the training portion of the SMS dataset, specifically targeting the spam class (label = 1) to address class imbalance. It begins by creating a DataFrame (data_frame_1_training) that combines the training text and labels. Using the ContextualWordEmbsAug augmenter from the nlpaug library, it leverages a pretrained distilbert-base-uncased language model to insert contextually appropriate words into existing spam messages, generating new synthetic samples. Each spam message is augmented three times, and the resulting texts are collected and labeled as spam. These new samples are then concatenated with the original training data to produce an augmented dataset (data_frame_1_training_augmented) with a more balanced class distribution. The use of tqdm provides a progress bar during augmentation for better tracking. Finally, the updated class distribution is printed, demonstrating the effectiveness of augmentation in increasing spam message representation.

```
import nlpaug.augmenter.word.context_word_embs as aug
from tqdm import tqdm

# make a data frame from the training data of SMS data
data_frame_1_training = pd.DataFrame({'text': X_train_SMS, 'label': y_train_SMS})

print('Distribution before augmentation:')
display(data_frame_1_training['label'].value_counts())

augmenter = aug.ContextualWordEmbsAug(
    model_path='distilbert-base-uncased',
    action="insert",
    top_k=20)

augmented_rows = []

# augment the entries having label as 1 in the sample_data_frame_1 and append it
spam_df = data_frame_1_training[data_frame_1_training['label'] == 1].reset_index(drop=True)

for i in tqdm(range(len(spam_df)), desc="Augmenting spam messages"):
    for _ in range(3):
        augmented = augmenter.augment(spam_df.iloc[i]['text'])
        if isinstance(augmented, list):
            augmented_text = " ".join(augmented)
        else:
            augmented_text = augmented
        augmented_rows.append({'text': augmented_text, 'label': 1})

aug_df = pd.DataFrame(augmented_rows)
data_frame_1_training_augmented = pd.concat([data_frame_1_training, aug_df], ignore_index=True)

print('\nDistribution after augmentation:')
display(data_frame_1_training_augmented['label'].value_counts())
```

Figure 9 Augmenting SMS train data

2.3.3.2.2. Feature Extraction from SMS Data

After augmentation we extracted the features from the augment data using the TFIDF Vectorizer package in Python. This code snippet performs **text vectorization** on the SMS datasets using the TF-IDF (Term Frequency-Inverse Document Frequency) method. First, it initializes a TfidfVectorizer object, which converts raw text data into numerical feature vectors that reflect the importance of each word relative to the document and the entire corpus. The vectorizer is then fitted and applied to the augmented and balanced training SMS data (X_train_SMS_balanced), transforming the text into a sparse matrix of TF-IDF features. Next, the same vectorizer is used to transform the test SMS data (X_test_SMS) without refitting, ensuring consistency between training and testing feature representations.

TF-IDF is a statistical measure that evaluates how relevant a word is to a document in a collection of documents. It combines two metrics: **Term Frequency (TF)**, which counts how frequently a word appears in a document, and **Inverse Document Frequency (IDF)**, which reduces the weight of common words appearing in many documents, highlighting words that are more unique and informative. This weighting helps models focus on meaningful words rather than common stopwords.

Using TF-IDF is important in text classification because it transforms text into a numerical format that machine learning algorithms can process while emphasizing discriminative features. This improves the model's ability to distinguish between spam and non-spam messages effectively.

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer_SMS = TfidfVectorizer()
X_train_SMS_balanced = vectorizer_SMS.fit_transform(X_train_SMS_balanced)
X_test_SMS = vectorizer_SMS.transform(X_test_SMS)
```

Figure 10 Features Extraction SMS

2.3.3.2.3. Feature Extraction and Augment Emails Data using SMOTE

SMOTE requires the data to be vectorized before applying augmentation, so we applied feature extraction on the Emails training data before resampling, also we applied feature extraction on the Emails test data but without augmentation.

This code performs **data augmentation** on the email training dataset to address class imbalance using **SMOTE (Synthetic Minority Over-sampling Technique)**.

First, it vectorizes the raw email text data using TF-IDF, converting the text into numerical feature vectors (X_train_Emails_vectorized) suitable for machine learning algorithms. Vectorization is done before applying SMOTE because SMOTE requires numerical input to generate synthetic samples.

SMOTE is an oversampling technique used to balance imbalanced datasets by creating synthetic examples of the minority class rather than simply duplicating existing samples. It works by selecting a minority class sample and identifying its k nearest neighbors (usually in feature space). Then, it generates new synthetic samples along the line segments connecting the sample to its neighbors. This interpolation introduces new, plausible samples that help the model learn a better decision boundary.

After applying SMOTE, the code prints the class distribution before and after augmentation, showing that the minority class (spam emails) is better represented. Balancing the dataset in this way helps improve model training and ultimately leads to better classification performance.

```
from imblearn.over_sampling import SMOTE

# vectorize the text before applying SMOTE
vectorizer_Emails = TfidfVectorizer()
X_train_Emails_vectorized = vectorizer_Emails.fit_transform(X_train_Emails)

# apply SMOTE
smote = SMOTE(random_state=42)
X_train_Emails_balanced, y_train_Emails_balanced = smote.fit_resample(X_train_Emails_vectorized, y_train_Emails)

print('Distribution before augmentation:')
display(y_train_Emails.value_counts())

print('\nDistribution after augmentation:')
display(y_train_Emails_balanced.value_counts())
```

Figure 11 Features Extraction and Augmentation

2.4. Models

Once the data was augmented we rechecked the distribution of the classes in the training split of the datasets and found that the distribution became better than before and can now be used for models training.

For this tutorial we applied three main models on the two datasets, as we trained and testes the models and got interesting accuracies among all models for both of the datasets the thing that confirms the importance of cleaning and augmenting the data to avoid any sort of overfitting in the models.

The first model used was Logistic Regression model with fine tuned hyper parameters, the second model used is the Mulinomial model also used with alpha parameter fine tuned and the third model is the Random Forest Classifier model. And note that the parameters were fine tuned using the GridSerachCV in Python. Here below the three models are discussed, with each model trained and tested on both datasets respectively.

2.4.1. Logistic Regression Model

Logistic Regression is a widely used and interpretable classification algorithm that is particularly effective for binary classification tasks such as spam detection. It models the probability that a given input belongs to a particular class using the logistic (sigmoid) function, making it ideal for distinguishing between "spam" and "not spam" categories. This model is robust, efficient for high-dimensional data such as TF-IDF text features, and performs well even with limited computational resources.

In the code below, we define a hyperparameter grid to optimize the performance of the logistic regression model using GridSearchCV. The grid includes different values for regularization penalties (L1 and L2), inverse regularization strength C, solver types (liblinear and lbfgs), and the number of maximum iterations. This setup allows the model to be tuned and cross-validated over multiple configurations to find the best-performing combination for our SMS and Email spam classification tasks.

```python
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
import sklearn.metrics
import numpy as np

# define the parameter grid
param_grid = [
    # Most commonly used: L2 penalty with reliable solvers
    {
        'penalty': ['l2'],
        'C': [0.01, 0.1, 1, 10],
        'solver': ['liblinear', 'lbfgs'],
        'max_iter': [100, 1000]
    },
    # L1 penalty: only use liblinear for speed and compatibility
    {
        'penalty': ['l1'],
        'C': [0.01, 0.1, 1],
        'solver': ['liblinear'],
        'max_iter': [1000]
    }
]
```

Figure 12 Logistic Regression Model

To evaluate the performance of Logistic Regression on the SMS and Emails dataset, we used a grid search approach (GridSearchCV) to identify the best combination of hyperparameters. The model was trained on the balanced and augmented datasets, and its performance was assessed using 4-fold cross-validation with F1-score as the evaluation metric. After training, we extracted the best estimator, its corresponding hyperparameters, and the best cross-validated score. The model was then tested on unseen data to evaluate its generalization capabilities. Performance metrics including accuracy, confusion matrix, and a detailed classification report were generated to analyze the effectiveness of the model in distinguishing spam from non-spam messages. A heatmap of the confusion matrix was also plotted for visual interpretation of the classification results

2.4.1.1. Logistic regression model results on SMS data

Based on the results of the Logistic Regression model on the SMS dataset, several important conclusions can be drawn:

- The model achieved very high **training accuracy** (~99.9%), indicating it learned the training data patterns well without significant underfitting.
- The **best hyperparameters** selected by grid search favor a relatively strong regularization parameter (C=10) with L2 penalty and the lbfgs solver, showing this configuration best balances bias and variance for the data.
- The **best cross-validation F1 score** of approximately 0.98 suggests the model generalizes well during training.
- On the **test set**, the accuracy remains high at around 97.2%, indicating strong generalization to unseen data and low overfitting.
- The **classification report** shows excellent performance for the majority class (non-spam, label 0) with precision and recall both near 98-99%. For the minority spam class (label 1), the precision is 91% and recall is 86%, yielding a solid F1-score of 0.88. This indicates that while the model is very good at identifying spam messages, there is a slight tendency to miss some spam instances (false negatives).

- The macro-average and weighted-average F1-scores (~0.93 and 0.97 respectively) confirm the model's overall balanced performance across both classes, despite class imbalance.

In summary, this Logistic Regression model performs strongly in distinguishing spam from non-spam SMS messages, with particularly high precision and recall for the majority class, and good but slightly lower recall for spam detection. This is a good baseline result for spam classification tasks.

```
Logistic Regression Model on SMS Dataset:
Fitting 4 folds for each of 19 candidates, totalling 76 fits

Training accuracy:
0.9992467043314501

Best parameters:
{'C': 10, 'max_iter': 100, 'penalty': 'l2', 'solver': 'lbfgs'}

Best estimator:
LogisticRegression(C=10)

Best score:
0.984028263351836

Testing accuracy:
0.9720930232558139

Classification report:
              precision    recall  f1-score   support

           0       0.98      0.99      0.98      1130
           1       0.91      0.86      0.88       160

    accuracy                           0.97      1290
   macro avg       0.95      0.92      0.93      1290
weighted avg       0.97      0.97      0.97      1290
```

Figure 13 Classification Report SMS Log Reg Model

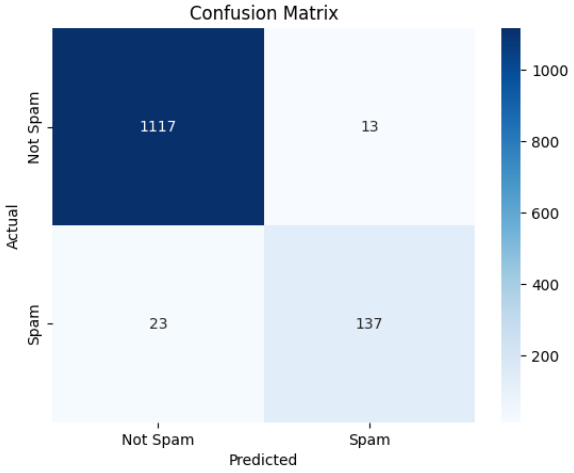


Figure 14 Confusion matrix

Furthermore we applied just this time an investigation on the misclassified SMS messages to see what data gave wrong predictions, but we kept it simple to just checking the wrong classification as already the accuracies during the testing phase are very successful. And in connection to where this project can be applied, in the context of 3VO—a crypto-focused social platform where user trust, safety, and smooth communication are critical—the performance of our spam classification model is highly satisfactory. With a **high test accuracy of 97.2%**, **strong F1-scores**, and only **36 misclassified messages out of 1,290 samples** (less than 3% error), the model demonstrates excellent reliability. Both **precision and recall are high**, especially for non-spam (the majority class), and remain reasonably strong for spam detection. Importantly, the model is also **simple, fast, and highly interpretable**—based on **Logistic Regression with**

TF-IDF vectorization—making it easy to deploy and maintain within a production pipeline. For a platform like 3VO, which is still evolving and prioritizes scalability and performance efficiency, this model strikes an ideal balance between accuracy and simplicity. While further enhancements could slightly reduce false positives or negatives, the current setup is already well-suited for **non-critical, high-volume tasks** such as basic spam filtering, allowing development teams to focus on more urgent or complex AI features without sacrificing performance.

2.4.1.2. Logistic regression model results on Emails data

Based on the outcome of the Logistic Regression model applied to the Emails dataset, we can conclude that the model performs exceptionally well in distinguishing between spam and non-spam emails. The model achieved a **perfect training accuracy of 1.0**, which indicates that it fit the training data completely. However, more importantly, it maintained a **very high testing accuracy of 99.37%**, showing strong generalization performance on unseen data.

The **best hyperparameters** selected by GridSearchCV were a regularization strength C = 10, L2 penalty, and the lbfgs solver with 100 maximum iterations. These settings contributed to the model achieving its best cross-validated F1 score of **0.9957**, which reflects a strong balance between precision and recall during training.

The **classification report** further confirms the robustness of the model:

- **Precision** for class 1 (spam) is 0.98, meaning 98% of the predicted spam messages were actually spam.
- **Recall** for class 1 is 0.99, meaning the model successfully detected 99% of all actual spam messages.
- **F1-score** for both classes is high (≥ 0.99), indicating a well-balanced and effective classifier.

These results demonstrate that the logistic regression model, when properly tuned and trained on balanced data using SMOTE, is highly effective for the task of spam email classification. The small gap between training and testing accuracy also suggests that the model is not overfitting.

```
Logistic Regression Model on Emails Dataset:
Fitting 4 folds for each of 19 candidates, totalling 76 fits

Training accuracy:
1.0

Best parameters:
{'C': 10, 'max_iter': 100, 'penalty': 'l2', 'solver': 'lbfgs'}

Best estimator:
LogisticRegression(C=10)

Best score:
0.9956998605215814

Testing accuracy:
0.9936797752808989

Classification report:
              precision    recall  f1-score   support

           0       1.00      0.99      1.00      1082
           1       0.98      0.99      0.99       342

    accuracy                           0.99      1424
   macro avg       0.99      0.99      0.99      1424
weighted avg       0.99      0.99      0.99      1424
```

Figure 15 Classification Report Emails Log Reg Model

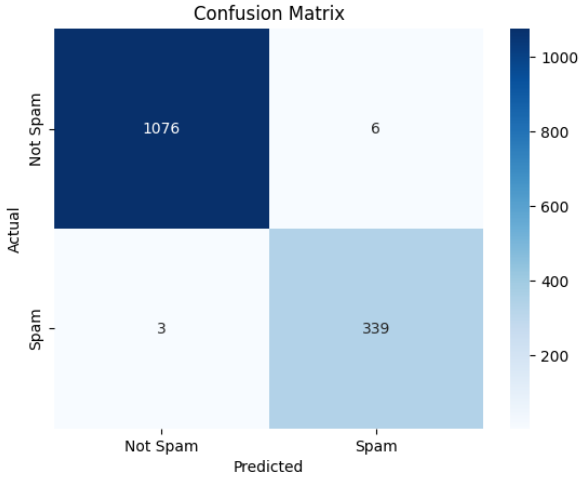


Figure 16 Confusion matrix

2.4.2. Multinomial Naïve Bayes Model

The **Multinomial Naive Bayes** model is a popular and efficient algorithm for text classification tasks, particularly well-suited for problems involving word count or frequency-based features such as those generated by TF-IDF. It is based on Bayes' Theorem and assumes that the features (words) are conditionally independent given the class label, and that their occurrences follow a multinomial distribution — a suitable assumption when dealing with discrete term frequencies. In the context of spam detection, this model is especially effective because it naturally captures patterns in word usage between spam and non-spam messages. Despite its simplicity, Multinomial Naive Bayes often performs competitively with more complex models in NLP applications. Its fast training time and low computational requirements make it a strong candidate for our binary classification task, especially when working with large text dataset.

This code defines a hyperparameter tuning setup for the **Multinomial Naive Bayes** model using GridSearchCV. The parameter being optimized is alpha, which is a **smoothing parameter** that helps prevent zero probabilities for words not seen in the training data. Lower values of alpha result in less smoothing, while higher values add more regularization. The code sets up a grid of candidate alpha values: 0.1, 0.5, 1.0, and 2.0.

```
param_grid_nb = {'alpha': [0.1, 0.5, 1.0, 2.0]}
grid_nb_SMS = GridSearchCV(MultinomialNB(), param_grid_nb, cv=4, scoring='f1', n_jobs=-1)
```

Figure 17 MultinomialNB model

2.4.2.1. MultinomialNB model results on SMS data

Based on the results of the Multinomial Naive Bayes model on the SMS dataset, we observe strong overall performance in classifying spam and non-spam messages. The model achieves a **training accuracy of approximately 98.8%** and a **testing accuracy of around 96.7%**, indicating good generalization with only a slight drop on unseen data.

Looking at the classification report, the model performs exceptionally well on the majority class (non-spam) with high precision and recall (~98%), showing it rarely misclassifies legitimate messages. For the minority class (spam), the precision is slightly lower at 86%, and recall is about 89%, indicating some false positives and false negatives but still relatively strong detection capability. The F1-score of 0.87 for spam reflects a balanced trade-off between precision and recall.

The macro-averaged metrics around 0.92–0.93 demonstrate that the model handles both classes fairly well, despite the class imbalance. These results confirm that the Multinomial Naive Bayes model is an effective and efficient choice for SMS spam classification, providing reliable predictions while being computationally lightweight.

```
Multinomial Model on SMS Dataset:

Training accuracy:
0.988135593220339

Testing accuracy:
0.9674418604651163

Classification report:
              precision    recall  f1-score   support

           0       0.98      0.98      0.98      1130
           1       0.86      0.89      0.87       160

    accuracy                           0.97      1290
   macro avg       0.92      0.93      0.93      1290
weighted avg       0.97      0.97      0.97      1290
```

Figure 18 Classification Report SMS MutinomialNB model

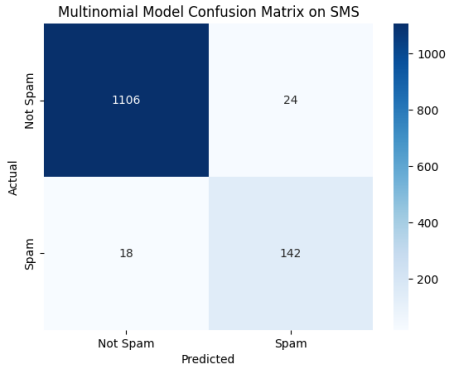


Figure 19 Confusion Matrix

2.4.2.2. MultinomialNB model results on Emails data

The results from the Multinomial Naive Bayes model on the Emails dataset demonstrate outstanding performance in spam detection. The model achieved an impressive **training accuracy of nearly 99.9%** and a **testing accuracy of approximately 99.6%**, indicating excellent generalization and minimal overfitting.

The classification report shows near-perfect precision, recall, and F1-scores for both classes. For the non-spam class, all metrics are effectively 1.00, meaning almost no legitimate emails were misclassified. For the spam class, the precision, recall, and F1-score are around 0.99, showing that the model accurately identifies nearly all spam emails with very few false positives or false negatives.

These results confirm that the Multinomial Naive Bayes model is highly effective for email spam classification, benefiting from its suitability for discrete word frequency data and its computational efficiency, making it a reliable choice for practical applications in email filtering.

```
Multinomial Model on Emails Dataset:

Training accuracy:
0.9990755007704161

Testing accuracy:
0.9957865168539326

Classification report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      1082
           1       0.99      0.99      0.99       342

    accuracy                           1.00      1424
   macro avg       0.99      0.99      0.99      1424
weighted avg       1.00      1.00      1.00      1424
```

Figure 20 Classification Report Emails MutinomialNB model

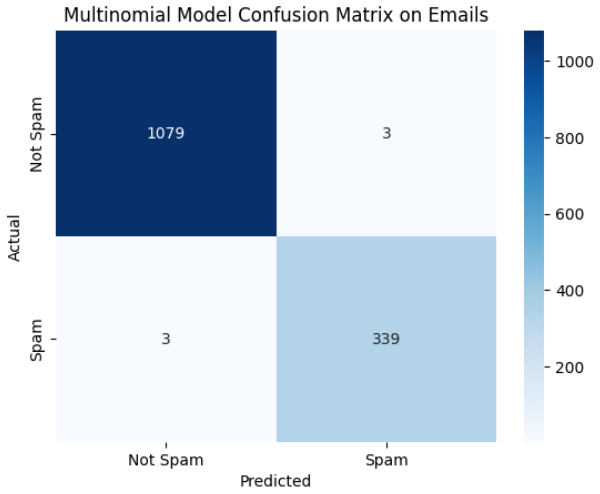


Figure 21 Confusion Matrix

2.4.3. Random Forest Classifier Model

The **Random Forest Classifier** is a powerful ensemble learning method that combines multiple decision trees to improve predictive accuracy and control overfitting. Each tree is trained on a random subset of the data and features, and their predictions are aggregated (usually by majority voting) to produce a final classification. This approach enhances robustness and generalization, especially useful in complex datasets with noisy or nonlinear relationships. In the context of spam classification, Random Forests can capture intricate patterns in text-derived features, making them well-suited to handle the variability in language and message structure found in both SMS and email datasets. Additionally, Random Forests provide feature importance insights, which can be valuable for understanding influential words or phrases in spam detection.

2.4.3.1. RFC model results on SMS data

The Random Forest Classifier (RFC) model on the SMS dataset shows excellent performance, with a perfect **training accuracy of 100%**, indicating that the model has fully learned the training data. On the testing set, it achieves a strong **accuracy of approximately 97.1%**, demonstrating good generalization to unseen messages.

The classification report reveals very high precision (97%) for both spam and non-spam classes, meaning the model's positive predictions are usually correct. Recall for the non-spam class is perfect at 100%, so nearly all legitimate messages are correctly identified. However, recall for the spam class is lower at 79%, suggesting some spam messages were missed (false negatives), though the F1-score of 0.87 still reflects a solid balance between precision and recall.

Overall, the RFC effectively captures complex patterns in the SMS data, resulting in robust spam detection, but there is a slight tendency to miss some spam instances compared to non-spam ones.

```
RFC Model on SMS Dataset:

Training accuracy:
1.0

Testing accuracy:
0.9713178294573643

Classification report:
              precision    recall  f1-score   support

           0       0.97      1.00      0.98      1130
           1       0.97      0.79      0.87       160

    accuracy                           0.97      1290
   macro avg       0.97      0.90      0.93      1290
weighted avg       0.97      0.97      0.97      1290
```

Figure 22 Classification Report SMS RFC model

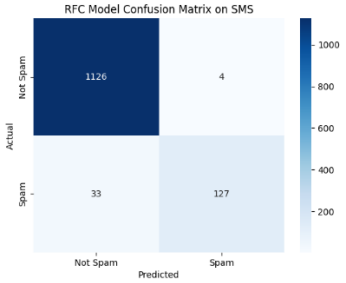


Figure 23 Confusion Matrix

2.4.3.2. RFC model results on Emails data

The Random Forest Classifier (RFC) applied to the Emails dataset achieved a **perfect training accuracy of 100%**, indicating the model learned the training data thoroughly. On the testing set, it maintained excellent performance with an overall accuracy of approximately **99%**.

The classification report shows very high precision and recall for both classes: non-spam emails have precision and recall around 99%, while spam emails have slightly lower but still strong precision of 98% and recall of 96%. The resulting F1-scores confirm the model's effectiveness at balancing precision and recall across both classes.

These results demonstrate that the Random Forest Classifier is highly capable of handling the email spam classification task, capturing complex patterns in the data while maintaining strong generalization and low error rates.

Ask ChatGPT

```
RFC Model on Emails Dataset:

Training accuracy:
1.0

Classification report:
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      1082
           1       0.98      0.96      0.97       342

    accuracy                           0.99      1424
   macro avg       0.98      0.98      0.98      1424
weighted avg       0.99      0.99      0.99      1424
```

Figure 24 Classification Report Emails RFC model

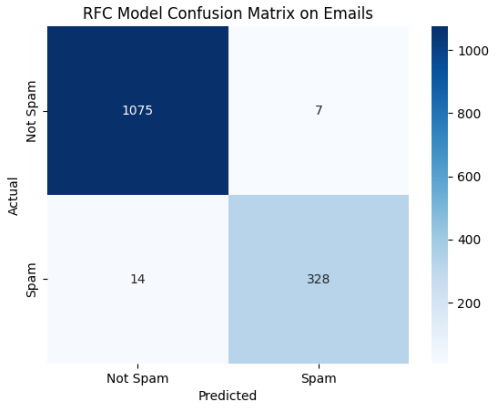


Figure 25 Confusion Matrix

2.4.4. Models Comparison

Here below we did a comparison between the models performance on both datasets and discussed our findings.

Table 1 Models Comparison on SMS dataset

Metric	Logistic Regression	Multinomial Naive Bayes	Random Forest Classifier
Accuracy	0.97	0.97	0.97
Precision (Spam)	0.90	0.86	0.97
Recall (Spam)	0.88	0.89	0.79
F1-Score (Spam)	0.89	0.87	0.87
Precision (Not Spam)	0.98	0.98	0.97
Recall (Not Spam)	0.99	0.98	1.00
F1-Score (Not Spam)	0.98	0.98	0.98

- **Accuracy:** All three models achieve very similar overall accuracy (~97%), indicating strong general performance.

- **Precision:** Random Forest shows the highest precision for spam detection (97%), meaning it makes fewer false positive errors in labeling legitimate messages as spam.
- **Recall:** Multinomial Naive Bayes has the highest recall for spam (89%), indicating it detects more spam messages correctly than the others. Random Forest's recall is notably lower (79%), meaning it misses more spam messages.
- **F1-Score:** Logistic Regression slightly edges out in balanced performance with an F1 of 0.89 on spam, indicating a good balance between precision and recall. Multinomial Naive Bayes and Random Forest have close F1-scores (~0.87).

Overall, Logistic Regression offers a balanced performance with strong precision and recall. Random Forest is more conservative in false positives but tends to miss more spam (lower recall). Multinomial Naive Bayes trades slightly lower precision for better recall on spam detection.

Table 2 Models Comparison on Emails data

Metric	Logistic Regression	Multinomial Naive Bayes	Random Forest Classifier
Accuracy	0.99	1.00	0.99
Precision (Spam)	0.98	0.99	0.98
Recall (Spam)	0.99	0.99	0.96
F1-Score (Spam)	0.99	0.99	0.97
Precision (Not Spam)	1.00	1.00	0.99
Recall (Not Spam)	0.99	1.00	0.99
F1-Score (Not Spam)	1.00	1.00	0.99

- **Accuracy:** All models perform exceptionally well, with Multinomial Naive Bayes achieving near-perfect accuracy (100%).
- **Precision & Recall:** Both Logistic Regression and Multinomial Naive Bayes maintain near-perfect precision and recall for both classes. Random Forest performs slightly lower on spam recall (96%) and F1-score (0.97), but still very strong.

Overall**,** The differences between the models here are minimal, and all are highly effective in email spam detection.

3. Conclusion

On the SMS dataset, Logistic Regression provides the best balance between precision and recall, making it the most reliable choice for practical spam filtering where both false positives and false negatives need to be minimized. Random Forest offers higher precision but at the cost of recall, risking missed spam. Multinomial Naive Bayes, while slightly lower in precision, excels at catching more spam messages.

On the Emails dataset, all models perform extremely well with very high precision, recall, and accuracy. Multinomial Naive Bayes achieves the best overall accuracy and balanced metrics, suggesting it is very well suited for email spam classification.

General recommendation: Logistic Regression is a robust, interpretable, and well-performing model for SMS spam classification, whereas Multinomial Naive Bayes slightly outperforms others on email data due to its affinity for text data represented as word frequencies. Random Forest can be considered when interpretability and feature importance insights are desired, but it might require tuning to avoid missing some spam messages.

Choosing the best model depends on the application's tolerance for false positives versus false negatives and the dataset characteristics. Combining these models in an ensemble or stacking approach could potentially improve performance further.

Self Assessment

No doubt that this project enhanced my knowledge in machine learning, as it helped me dealing with real data having a direct impact, as spam detection nowadays is a must especially on social platforms such as 3VO where credibility and correctness of posted data is valuable. Also such platform dealing with cryptocurrency wallets and web3 technologies should be always protected from any malicious activity that in the most of the times is initiated by a spam message or post that fool the legitimate users and steal their credentials.

Furthermore, this project was challenging especially the data augmentation part, where I learned new topics about NLP and how different tools behave to augment data and what is the importance of cleaning and augmenting data so any model even the simplest one can be used and perform well on the data.

Hence the workflow enriched my technical knowledge as well as critical thinking and ability to understand the impact of projecting this project into any business that might benefit from it.

Appendix

```
# -*- coding: utf-8 -*-
"""SpamClassification.ipynb

Automatically generated by Colab.

Original file is located at
    https://colab.research.google.com/drive/1zqln3G56gRXf3nLPRDRRpGMbZeiHhKY8

# SMS Spam Classification (Text Classification)

# Load and explore the datasets

## Load the datasets
"""

# Commented out IPython magic to ensure Python compatibility.
# %reset -f
import pandas as pd

data_frame_1 = pd.read_csv('/content/sample_data/spam.csv')
data_frame_2 = pd.read_csv('/content/sample_data/emails.csv')

print('SMS data shape:')
print(data_frame_1.shape)

print('\nEmails data shape:')
print(data_frame_2.shape)

"""## Explore columns and first few rows"""

print('Columns in SMS data:')
print(data_frame_1.columns)

print('\nData head of SMS data:')
print(data_frame_1.head())

print('\nColumns in Emails data:')
print(data_frame_2.columns)

print('\nData head of Email data:')
print(data_frame_2.head())

"""## Check null values count in datatsets"""

print('\nNull values count in SMS data:')
print(data_frame_1.isna().sum())

print('\nNull values count in Emails data:')
print(data_frame_2.isna().sum())

"""## Datatypes in datasets"""

print('\nData types in SMS data:')
```

```
print(data_frame_1.dtypes)

print('\nData types in Emails data:')
print(data_frame_1.dtypes)

"""## Generate an automatic Explanatory Data Analysis EDA report"""

!pip install ydata_profiling

from ydata_profiling import ProfileReport
profile_data_frame_1 = ProfileReport(data_frame_1, title="SMS Data Profiling Report")
profile_data_frame_2 = ProfileReport(data_frame_2, title="Emails Data Profiling Report")

profile_data_frame_1.to_file("sms_data_report.html")
print('EDA report created for SMS dataset\n\n\n')

profile_data_frame_2.to_file("emails_data_report.html")
print('EDA report created for Emails dataset')

"""# Datasets Preprocessing, Cleaning and Resampling

## Drop duplicate rows from datasets
"""

print('There are', data_frame_1.duplicated().sum(),'duplicate rows in SMS dataset')
print('There are', data_frame_2.duplicated().sum(),'duplicate rows in Emails dataset')

# remove duplicates from data_frame_1
data_frame_1 = data_frame_1.drop_duplicates().copy()
print('\nDuplicate rows in SMS data are:', data_frame_1.duplicated().sum())

# remove duplicated from data_frame_2
data_frame_2 = data_frame_2.drop_duplicates().copy()
print('Duplicate rows in Emails data are:', data_frame_2.duplicated().sum())

"""## Check data distribution in datasets"""

print('SMS data distribution:')
print(data_frame_1['Category'].value_counts())
data_frame_1_imbalance_ratio = data_frame_1['Category'].value_counts().iloc[0]/
data_frame_1['Category'].value_counts().iloc[1]
print('Imbalanace ratio:', data_frame_1_imbalance_ratio)

print('\nEmails data distribution:')
data_frame_2_imbalance_ratio = data_frame_2['spam'].value_counts().iloc[0] /
data_frame_2['spam'].value_counts().iloc[1]
print(data_frame_2['spam'].value_counts())
print('Imbalance ratio:', data_frame_2_imbalance_ratio)

"""## Plot data distribution"""

import matplotlib.pyplot as plt
import seaborn as sns

fig, axes = plt.subplots(1, 2, figsize=(12, 5))
```

```
# Plot 1: SMS Scatter plot
sns.countplot(data=data_frame_1, x='Category', hue='Category', ax=axes[0])
axes[0].set_title('Spam/Not Spam Distribution in SMS')
axes[0].legend(title='Spam')

# Plot 2: Email Data
sns.countplot(data=data_frame_2, x='spam', hue='spam', ax=axes[1])
axes[1].set_title('Spam/Not Spam Distribution in Emails')
axes[1].legend(title='Spam')

plt.tight_layout()
plt.show()

"""***Data distribution for both classes is imbalanced so we need to do resampling for the data***

## Rename columns as 'text' and 'label'
"""

data_frame_1.rename(columns={'Category': 'label', 'Message': 'text'}, inplace=True)
data_frame_2.rename(columns={'spam': 'label', 'text':'text'}, inplace=True)

"""## Map label values to 0 for not spam and 1 for spam in SMS data"""

data_frame_1['label'] = data_frame_1['label'].map({'ham': 0, 'spam': 1})

"""## Add length column for datasets"""

data_frame_1['length'] = data_frame_1['text'].apply(lambda x: len(x))
print(data_frame_1.head())

data_frame_2['length'] = data_frame_2['text'].apply(lambda x: len(x))
print(data_frame_2.head())

"""## Pipeline for cleaning text raw data"""

import nltk
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
nltk.download('punkt_tab')
nltk.download('wordnet')
nltk.download('stopwords')
import re
import string

# remove the 'Subject' from text in Emails data
def remove_subject_from_text_emails(text: str):
    if text.lower().startswith("subject:"):
        return text[len("subject:"):].strip()
    return text

# to lower
def to_lower(text: str):
  return text.lower()
```

```
# remove numbers from text
def remove_numbers(text: str):
  for c in text:
    if c.isnumeric():
      text = text.replace(c,' ')
  return text

# lemmatizing
def lemmatizing(text: str):
  lemmatizer = WordNetLemmatizer()
  tokens = word_tokenize(text)
  for i in range(len(tokens)):
    lemma_world = lemmatizer.lemmatize(tokens[i])
    tokens[i] = lemma_world
  return " ".join(tokens)

# remove punctuation
def remove_punctuation(text: str):
  for c in text:
    if c in string.punctuation:
      text = text.replace(c,'')
  return text

# remove stop words
def remove_stopwords(text: str):
  removed = []
  stop_words = list(stopwords.words("english"))
  tokens = word_tokenize(text)
  for i in range(len(tokens)):
    if tokens[i] not in stop_words:
      removed.append(tokens[i])
  return " ".join(removed)

# remove extra white spaces
def remove_extra_white_spaces(text: str):
  single_char_pattern = r'\s+[a-zA-Z]\s+'
  without_sc = re.sub(pattern=single_char_pattern, repl=" ", string= text)
  return without_sc

data_frame_2['text'] = data_frame_2['text'].apply(lambda x: remove_subject_from_text_emails(x))

data_frame_1['text'] = data_frame_1['text'].apply(lambda x: to_lower(x))
data_frame_2['text'] = data_frame_2['text'].apply(lambda x: to_lower(x))

data_frame_1['text'] = data_frame_1['text'].apply(lambda x: remove_numbers(x))
data_frame_2['text'] = data_frame_2['text'].apply(lambda x: remove_numbers(x))

data_frame_1['text'] = data_frame_1['text'].apply(lambda x: remove_punctuation(x))
data_frame_2['text'] = data_frame_2['text'].apply(lambda x: remove_punctuation(x))

data_frame_1['text'] = data_frame_1['text'].apply(lambda x: remove_stopwords(x))
data_frame_2['text'] = data_frame_2['text'].apply(lambda x: remove_stopwords(x))

data_frame_1['text'] = data_frame_1['text'].apply(lambda x: remove_extra_white_spaces(x))
data_frame_2['text'] = data_frame_2['text'].apply(lambda x: remove_extra_white_spaces(x))
```

```
data_frame_1['text'] = data_frame_1['text'].apply(lambda x: lemmatizing(x))
data_frame_2['text'] = data_frame_2['text'].apply(lambda x: lemmatizing(x))

"""## Update the length after cleaning"""

data_frame_1['length_after_cleaning'] = data_frame_1['text'].apply(lambda x: len(x)).copy()
data_frame_2['length_after_cleaning'] = data_frame_2['text'].apply(lambda x:len(x)).copy()

print('After cleaning SMS data:')
display(data_frame_1.head())

print('\nAfter cleaning Emails data:')
display(data_frame_2.head())

"""## Better to split now the data into training and testing to avoid data leakage"""

from sklearn.model_selection import train_test_split
# split SMS data
X_train_SMS, X_test_SMS, y_train_SMS, y_test_SMS = train_test_split(data_frame_1['text'], data_frame_1['label'],
test_size = 0.25,
                                                    stratify = data_frame_1['label'], random_state=42)
print('Distribution in SMS train data: ')
display(y_train_SMS.value_counts())

# split Emails data
X_train_Emails, X_test_Emails, y_train_Emails, y_test_Emails = train_test_split(data_frame_2['text'],
data_frame_2['label'], test_size = 0.25,
                                                    stratify = data_frame_2['label'], random_state=42)
print('\nDistribution in Emails train data: ')
display(y_train_Emails.value_counts())

"""## Augment SMS training data using NLPAUG"""

!pip install nlpaug

import nlpaug.augmenter.word.context_word_embs as aug
from tqdm import tqdm

# make a data frame from the training data of SMS data
data_frame_1_training = pd.DataFrame({'text': X_train_SMS, 'label': y_train_SMS})

print('Distribution before augmentation:')
display(data_frame_1_training['label'].value_counts())

augmenter = aug.ContextualWordEmbsAug(
    model_path='distilbert-base-uncased',
    action="insert",
    top_k=20)

augmented_rows = []

# augment the entries having label as 1 in the sample_data_frame_1 and append it
spam_df = data_frame_1_training[data_frame_1_training['label'] == 1].reset_index(drop=True)

for i in tqdm(range(len(spam_df)), desc="Augmenting spam messages"):
    for _ in range(3):
```

```
        augmented = augmenter.augment(spam_df.iloc[i]['text'])
        if isinstance(augmented, list):
            augmented_text = " ".join(augmented)
        else:
            augmented_text = augmented
        augmented_rows.append({'text': augmented_text, 'label': 1})

aug_df = pd.DataFrame(augmented_rows)
data_frame_1_training_augmented = pd.concat([data_frame_1_training, aug_df], ignore_index=True)

print('\nDistribution after augmentation:')
display(data_frame_1_training_augmented['label'].value_counts())

""" ## Set training SMS data"""

X_train_SMS_balanced = data_frame_1_training_augmented['text']
y_train_SMS_balanced = data_frame_1_training_augmented['label']

"""## Save a raw version of SMS augmented data"""

X_train_SMS_balanced_raw = X_train_SMS_balanced.copy()
y_train_SMS_balanced_raw = y_train_SMS_balanced.copy()

X_test_SMS_raw = X_test_SMS.copy()

"""## Vectorize text in SMS data"""

from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer_SMS = TfidfVectorizer()
X_train_SMS_balanced = vectorizer_SMS.fit_transform(X_train_SMS_balanced)
X_test_SMS = vectorizer_SMS.transform(X_test_SMS)

"""## Augment Emails data using SMOTE"""

from imblearn.over_sampling import SMOTE

# vectorize the text before applying SMOTE
vectorizer_Emails = TfidfVectorizer()
X_train_Emails_vectorized = vectorizer_Emails.fit_transform(X_train_Emails)

# apply SMOTE
smote = SMOTE(random_state=42)
X_train_Emails_balanced, y_train_Emails_balanced = smote.fit_resample(X_train_Emails_vectorized,
y_train_Emails)

print('Distribution before augmentation:')
display(y_train_Emails.value_counts())

print('\nDistribution after augmentation:')
display(y_train_Emails_balanced.value_counts())

"""## Vectorize text in test Emails data"""

X_test_Emails = vectorizer_Emails.transform(X_test_Emails)
```

```
"""# Models"""

from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, confusion_matrix

"""## Logistic Regression"""

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
import sklearn.metrics
import numpy as np

# define the parameter grid
param_grid = [
   # Most commonly used: L2 penalty with reliable solvers
   {
      'penalty': ['l2'],
      'C': [0.01, 0.1, 1, 10],
      'solver': ['liblinear', 'lbfgs'],
      'max_iter': [100, 1000]
   },
   # L1 penalty: only use liblinear for speed and compatibility
   {
      'penalty': ['l1'],
      'C': [0.01, 0.1, 1],
      'solver': ['liblinear'],
      'max_iter': [1000]
   }
]

"""### SMS Logistic Regression Model"""

logistic_regression_model_SMS = GridSearchCV(LogisticRegression(), param_grid, cv=4, scoring='f1', n_jobs=-1,
verbose=1)

print('Logistic Regression Model on SMS Dataset:')
logistic_regression_model_SMS.fit(X_train_SMS_balanced, y_train_SMS_balanced)

print('\nTraining accuracy:')
y_train_pred = logistic_regression_model_SMS.predict(X_train_SMS_balanced)
print(sklearn.metrics.accuracy_score(y_train_SMS_balanced, y_train_pred))

print('\nBest parameters:')
print(logistic_regression_model_SMS.best_params_)

print('\nBest estimator:')
print(logistic_regression_model_SMS.best_estimator_)

print('\nBest score:')
print(logistic_regression_model_SMS.best_score_)

# testing the model

predictSMS = logistic_regression_model_SMS.predict(X_test_SMS)

print('\nTesting accuracy:')
```

```
print(sklearn.metrics.accuracy_score(y_test_SMS, predictSMS))

print('\nConfusion matrix:')
print(confusion_matrix(y_test_SMS, predictSMS))

cm1 = confusion_matrix(y_test_SMS, predictSMS)
sns.heatmap(cm1, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()

print('\nClassification report:')
print(classification_report(y_test_SMS, predictSMS))

"""#### Investigate misclassified SMS messages"""

misclassified_indices = np.where(predictSMS != y_test_SMS)[0]
print(f"\nNumber of misclassified SMS: {len(misclassified_indices)}")

error_df = pd.DataFrame({
    'text': X_test_SMS_raw.iloc[misclassified_indices],  # this is the cleaned version of test data
    'true_label': y_test_SMS.iloc[misclassified_indices].values,
    'predicted': predictSMS[misclassified_indices]
})

print("\nSample misclassified SMS messages:")
display(error_df.head(5))

"""### Emails Logistic Regression Model"""

logistic_regression_model_Emails = GridSearchCV(LogisticRegression(), param_grid, cv=4, scoring='f1', n_jobs=-1, verbose=1)

print('Logistic Regression Model on Emails Dataset:')
logistic_regression_model_Emails.fit(X_train_Emails_balanced, y_train_Emails_balanced)

print('\nTraining accuracy:')
y_train_pred = logistic_regression_model_Emails.predict(X_train_Emails_balanced)
print(sklearn.metrics.accuracy_score(y_train_Emails_balanced, y_train_pred))

print('\nBest parameters:')
print(logistic_regression_model_Emails.best_params_)

print('\nBest estimator:')
print(logistic_regression_model_Emails.best_estimator_)

print('\nBest score:')
print(logistic_regression_model_Emails.best_score_)

# testing the model

predictEmails = logistic_regression_model_Emails.predict(X_test_Emails)

print('\nTesting accuracy:')
```

```
print(sklearn.metrics.accuracy_score(y_test_Emails, predictEmails))

print('\nConfusion matrix:')
print(confusion_matrix(y_test_Emails, predictEmails))

cm1 = confusion_matrix(y_test_Emails, predictEmails)
sns.heatmap(cm1, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()

print('\nClassification report:')
print(classification_report(y_test_Emails, predictEmails))

"""## Multinomial Regression"""

from sklearn.naive_bayes import MultinomialNB

"""### SMS Multinomial Model"""

param_grid_nb = {'alpha': [0.1, 0.5, 1.0, 2.0]}
grid_nb_SMS = GridSearchCV(MultinomialNB(), param_grid_nb, cv=4, scoring='f1', n_jobs=-1)
multinomial_model_SMS = MultinomialNB()

print('Multinomial Model on SMS Dataset:')
multinomial_model_SMS.fit(X_train_SMS_balanced, y_train_SMS_balanced)

print('\nTraining accuracy:')
y_train_pred = multinomial_model_SMS.predict(X_train_SMS_balanced)
print(sklearn.metrics.accuracy_score(y_train_SMS_balanced, y_train_pred))

# testing the model

predictSMS = multinomial_model_SMS.predict(X_test_SMS)

print('\nTesting accuracy:')
print(sklearn.metrics.accuracy_score(y_test_SMS, predictSMS))

print('\nConfusion matrix:')
print(confusion_matrix(y_test_SMS, predictSMS))

cm1 = confusion_matrix(y_test_SMS, predictSMS)
sns.heatmap(cm1, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Multinomial Model Confusion Matrix on SMS')
plt.show()

print('\nClassification report:')
print(classification_report(y_test_SMS, predictSMS))

"""### Emails Multinomial Model"""
```

```
multinomial_model_Emails = GridSearchCV(MultinomialNB(), param_grid_nb, cv=4, scoring='f1', n_jobs=-1)

print('Multinomial Model on Emails Dataset:')
multinomial_model_Emails.fit(X_train_Emails_balanced, y_train_Emails_balanced)

print('\nTraining accuracy:')
y_train_pred = multinomial_model_Emails.predict(X_train_Emails_balanced)
print(sklearn.metrics.accuracy_score(y_train_Emails_balanced, y_train_pred))

# testing the model

predictEmails = multinomial_model_Emails.predict(X_test_Emails)

print('\nTesting accuracy:')
print(sklearn.metrics.accuracy_score(y_test_Emails, predictEmails))

print('\nConfusion matrix:')
print(confusion_matrix(y_test_Emails, predictEmails))

cm1 = confusion_matrix(y_test_Emails, predictEmails)
sns.heatmap(cm1, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Spam', 'Spam'], yticklabels=['Not Spam',
'Spam'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Multinomial Model Confusion Matrix on Emails')
plt.show()

print('\nClassification report:')
print(classification_report(y_test_Emails, predictEmails))

"""## Random Forest Classifier"""

from sklearn.ensemble import RandomForestClassifier

"""### SMS Random Forest Classifier Model"""

RFC_model_SMS = RandomForestClassifier(n_estimators=50, random_state=42)

print('RFC Model on SMS Dataset:')
RFC_model_SMS.fit(X_train_SMS_balanced, y_train_SMS_balanced)

print('\nTraining accuracy:')
y_train_pred = RFC_model_SMS.predict(X_train_SMS_balanced)
print(sklearn.metrics.accuracy_score(y_train_SMS_balanced, y_train_pred))

# testing the model

predictSMS = RFC_model_SMS.predict(X_test_SMS)

print('\nTesting accuracy:')
print(sklearn.metrics.accuracy_score(y_test_SMS, predictSMS))

print('\nConfusion matrix:')
print(confusion_matrix(y_test_SMS, predictSMS))

cm1 = confusion_matrix(y_test_SMS, predictSMS)
```

```
sns.heatmap(cm1, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('RFC Model Confusion Matrix on SMS')
plt.show()

print('\nClassification report:')
print(classification_report(y_test_SMS, predictSMS))

"""### Emails Random Forest Classifier Model"""

RFC_model_Emails = RandomForestClassifier(n_estimators=50, random_state=42)

print('RFC Model on Emails Dataset:')
RFC_model_Emails.fit(X_train_Emails_balanced, y_train_Emails_balanced)

print('\nTraining accuracy:')
y_train_pred = RFC_model_Emails.predict(X_train_Emails_balanced)
print(sklearn.metrics.accuracy_score(y_train_Emails_balanced, y_train_pred))

# testing the model

predictEmails = RFC_model_Emails.predict(X_test_Emails)

print('\nTesting accuracy:')
print(sklearn.metrics.accuracy_score(y_test_Emails, predictEmails))

print('\nConfusion matrix:')
print(confusion_matrix(y_test_Emails, predictEmails))

cm1 = confusion_matrix(y_test_Emails, predictEmails)
sns.heatmap(cm1, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('RFC Model Confusion Matrix on Emails')
plt.show()

print('\nClassification report:')
print(classification_report(y_test_Emails, predictEmails))
```