Hyperlink Is All You Need:
Classifying Spam Comments of YouTube Contents

Alexander Lee, Domenico Oppedisano, Jaehyeon Park, Makoto Takahara

December 8, 2024

- **Quercus Group Number**: 100
- **Kaggle Competition**: Detect spam youtube comment
- **Kaggle Team Name**: Machine Teachers
- **Kaggle Private Score**: 0.93309
- **Kaggle Public Score**: 0.94719
- **Kaggle Private Ranking**: 35
- **Kaggle Public Ranking**: 22

1 Introduction

YouTube is one of the world's most popular social media platforms, where users can share and engage with video content. However, the platform's popularity and community-driven nature make it highly susceptible to spam. YouTube's parent company Google (n.d.) defines spam as "content or correspondences that create a negative experience by making it difficult to find more relevant and substantive material." Spam comments undermine user experiences, creating challenges with broad-reaching business, ethical, and social implications.

To address these challenges, this project focuses on two key objectives. The first objective is to develop a method for accurately predicting whether a YouTube comment is spam based on its contents and metadata. A well functioning spam detection system should be able to identify spam comments to ensure harmful content does not persist, but it should not be too strict such that legitimate comments get marked as spam as well. False positives (legitimate comments incorrectly flagged as spam) pose a great risk to community engagement and brand perception, potentially alienating users and undermining content creator autonomy. As Perez (2024) demonstrated, even isolated incidents of mislabeling legitimate content can harm YouTube's reputation and reduce revenues. In contrast, the consequences of a missed spam comment are less severe, as long as the volume of missed spam comments is low. Therefore, this project aims to achieve high overall accuracy while emphasizing minimizing false positives in its approach to spam detection.

In addition to accurate spam prediction, this project seeks to gain a deeper understanding of the factors that indicate whether a comment is spam. Specifically, hyperlink-related keywords—such as *www* or *https*—have often been associated with spam content, as reflected in Google's recent ban on links in comments on YouTube Shorts (Hale, 2023). By identifying words that are most indicative of spam, this project can provide valuable insights to enhance detection systems and help creators make informed decisions when moderating their own channels.

- **RQ1:** *How can we predict whether a YouTube comment is spam based on its contents and metadata, while minimizing the risk of misclassifying legitimate comments?*
- **RQ2:** *What are the most important words in YouTube comments that indicate they are likely to be spam, and do keywords related to hyperlinks play a significant role in identifying spam content?*

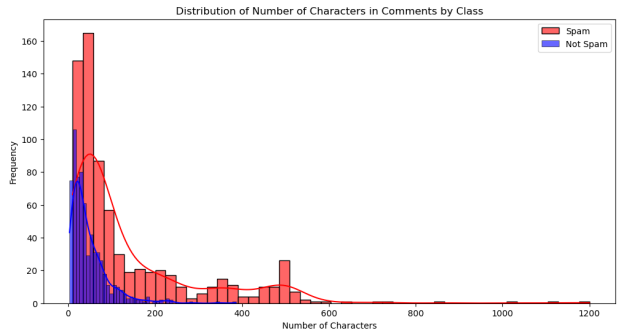
Accurately answering **RQ1** will improve the user experience by reducing harmful content while preserving the openness of the platform for legitimate users and creators. Furthermore, it aligns with Google's ethical responsibility to safeguard users against identity theft, phishing, and the spread of malicious software, while also protecting its brand image and revenue streams. Along with that, answering **RQ2** will not only support better detection but also allows content creators to create safer and more engaging communities. This project highlights the connection between hyperlinks and spam, offering insights into Google's content moderation policies.

This project seeks to address the challenges of spam detection on YouTube through two key objectives. First, it aims to develop a spam prediction method that minimizes false positives to maintain user trust and engagement. Second, it investigates the most important words indicative of spam to enhance both detection systems and community moderation efforts. Together, these goals aim to improve user experience, support content creators, and strengthen YouTube as a trusted and engaging platform.

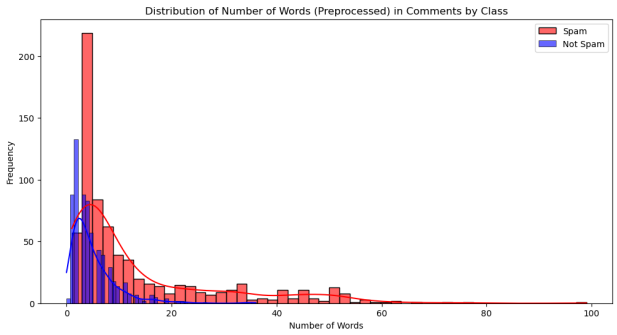
2 Data

2.1 Dataset

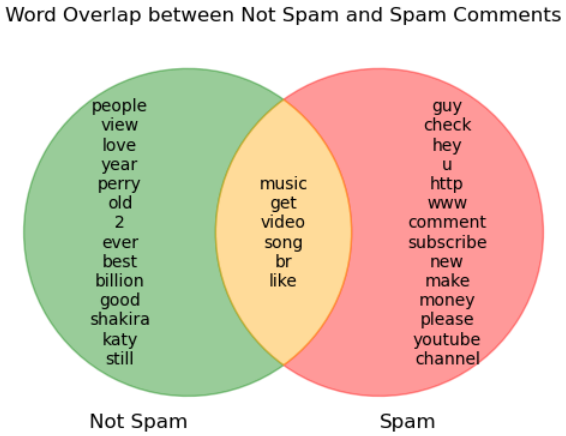
The data was sourced from the Kaggle competition "Detect Spam in YouTube Comments" designed for course STA314 at the University of Toronto.



(a) Figure 1



(b) Figure 2



(c) Figure 3

2.2 Exploratory Data Analysis

The training data has 1369 observations, with 51.86% of these comments being spam and the remaining 48.14% being legitimate comments, indicating a balanced dataset. In addition to the binary variable classifying a comment as spam, we have categorical variables for the account name of the comment, the date of the comment, the comment itself, and the name of the video it was commented on. 170 observations had a missing `DATE` feature, in which all observations with missing dates were classified as spam.

The five most common words in the dataset were: *this, the, and, to,* and *I*. These are examples of stopwords, providing no significant meaning to the content. Thus, in the exploratory data analysis, these words were omitted. We also find that *\ufeff* is a common word, but it is a metadata marker and not the actual content of the comment. Additionally, we observe that words with identical meanings are considered separately, either through different cases like *Check* and *check* or variations of the same stem word like *you* and *your*.

We first investigated whether the length of a comment, in terms of word count and character count, was associated with the content of the comment being spam. When we overlay histograms of word count for each class in Figures 1 and 2, we see that their distributions are very similar in modes and variance. We obtain similar results for character count. Visual inspection indicates little difference between the word counts and character counts of spam comments and legitimate comments, so we do not consider them meaningful predictors of whether a comment is spam.

Figure 3 shows the most common words in comments that were spam and legitimate, as well as words that were common in both. We see that legitimate comments appear to have words related to the content of the video. In contrast, spam comments tend to commonly have YouTube-related keywords like *subscribe, comment,* and *channel*. Words that call for action, like *please, check,* and *hey* are also common for spam comments. These words could be crucial to differentiating spam comments from legitimate comments, allowing us to answer our research questions of predicting whether a comment is spam based on its contents and investigating the words most indicative of spam.

br is common in both; we suspect this is an example of a typo. Typo frequency was observed to be 55.84% in legitimate comments and 84.65% in spam comments when a word is classified as a typo if not found in the English dictionary. Although Fisher's Exact Test indicates that the difference in these proportions is statistically significant, we chose not to incorporate typo presence in our further analysis because proper nouns and slang terms were incorrectly marked as typos, despite being contextually valid.

3 Methods

3.1 Data Preprocessing

3.1.1 Non-Comment Feature Engineering

From the exploratory data analysis, we recognized that the comment without the dates are always spam. We assumed that such information is significant and can improve our model. Thus, we transformed `DATE` column of the dataset into a dummy variable, where 1 = No date given 0 = date given.

3.1.2 Comment Cleaning

Natural Language Processing Toolkit (NLTK) is a python library that we used for during comment preprocessing. This library processes human language data and provides resources like WordNet and tools for natural language processing tasks such as tokenization and semantic analysis. Because our comment dataset involves natural language, we utilized this library throughout the data preprocessing steps.

First, we cleaned our sentence structured raw comment by removing insignificant words. In the function `remove_punctuations_stop_words()` we removed the string *\ufeff* since it is not interpretable when we tokenize it. The NLTK library was utilized to get rid of the stopwords. After that, we removed all punctuation marks besides the apostophe (') and stopwords because it doesn't add any value or meaning. Stopwords are words that appear often that bring no meaningful value to a sentence such as *a, the, in,* etc. We didn't filter out the apostrophe since the stopwords would miscategorize stopwords if the apostrophe isn't there. For example *couldn* is a stopword but *couldn't* is not a stopword. We dealt with apostrophes later when we ran TreebankTokenizer. From the EDA we standardize words by making all words lowercase and taking the stem words to reduce variation.

Then, because YouTube decided to ban links from YouTube shorts, we assumed hyperlink-related keywords would be an appropriate indicator of spam comments. Thus, we specifically identified hyperlink-related contents from the comments. In the function `identify_web_related_terms()` we isolated the occurrences of *https* and *www* from their original links because we are not concerned with the specific website but rather the fact that a hyperlink exists. By doing so, if a comment contains a link, there will be dedicated features that will indicate that the comment contains a link. Without this we can't properly define a feature that indicates whether a comment has a hyperlink or not. We also replaced double whitespace to a single whitespace to avoid inconsistent spacing.

In the function `tokenize()` we once again used the NLTK library to tokenize the comment. We used the `TreebankWordTokenizer` method which split standard contractions. For example the word *[don't]* will turn into *[do] [n't]* or *[they'll]* will turn into *[they] ['ll].* We then used the `WordNetLemmatizer` method which is also from the NLTK library. This simplifies words to their base form, such as turning *dogs* into *dog* or *abaci* into *abacus.*

3.1.3 Word Vectorization

From the exploratory data analysis, we know that there exists unique words used in each type of comment like *love* for non-spam and *money* for spam comments that carry a specific connotation that aligns with the intent of the comment. In contrast, common words like *song* are common in both, making them less informative for identifying the intent of the comment. From this, we assumed that

these unique words have greater weight in spam classification tasks than common words and wanted to recognize such phenomena when representing the comments from the dataset into numerical values.

TF-IDF (Term Frequency-Inverse Document Frequency) is a common method in natural language processing that turns words into numerical values. The method weights the importance of a word in a comment relative to all the comments. The following is an example of how TF-IDF vectorization functions.

- **Term Frequency (TF):** Measures how often a term appears in a document. Higher frequency means the word is more important within that specific document.

$$\text{TF} = \frac{N}{TT}$$

where N = Number of times a term appears in the document & TT = Total number of terms in the document

- **Inverse Document Frequency (IDF):** Reduces the weight of commonly occurring words across many documents, such as *the* or *and*, as they are less significant for understanding content.

$$\text{IDF} = \log\left(\frac{\text{TD}}{\text{NC}}\right)$$

where TD = Total number of documents & NC = Number of documents containing the term This example represents a small sample size that briefly explains the process of how TF-IDF vectorization works by computing the $\text{TF} - \text{IDF}$ value of *www*.

  - Comment 1: *I like those cats*
  - Comment 2: *www hoopshub*
  - Comment 3: *Make money at www moneymoves*

For the term *www* it appears in:
Comment 1: $\frac{0}{4} = 0$ **Comment 2:** $\frac{1}{2} = 0.2$ **Comment 3:** $\frac{1}{5} = 0.2$
The Document Frequency (DF) and Inverse Document Frequency (IDF) of `"www"`:

$$\text{DF} = 2 \quad \text{IDF} = \log\left(\frac{3}{2}\right) \approx 0.176$$

The TF-IDF is calculated as $\text{TF-IDF} = \text{TF} \times \text{IDF}$. The following is the result:

Document	TF	IDF	TF-IDF
Comment 1	0	0.176	$0 \times 0.176 = 0$
Comment 2	0.2	0.176	$0.2 \times 0.176 = 0.0352$
Comment 3	0.2	0.176	$0.2 \times 0.176 = 0.0352$

As previously mentioned, we isolated "www" so that it can be its own individual feature to help determine whether it is spam or not. This also relates back to RQ2 where it helps us identify the keywords that are related to spam.

3.2 Model Selection and Training

3.2.1 Linear Separability

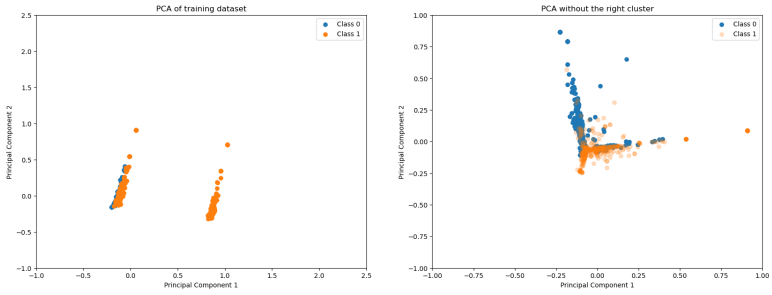
Assessing the linear separability of our dataset is crucial because some models perform better when data is linearly separable. To check if the data is separable, we used a hard-margin Support Vector Machine with a linear kernel for this procedure. A hard margin SVM determines whether the data is linearly separable and tries to find a hyperplane that perfectly separates the data with no misclassification. We ran to check the test accuracy and our result was 92.3%, which was inconclusive. If the

data was linearly separable we would have expected a test accuracy score of 100%. Thus, from this hard margin SVM model, we can assume that the dataset is not linearly-separable.

Because our test accuracy score was relatively high, we wanted to see if some data was separable. To do so, we ran a Principal Component Analysis (PCA) of this data. This procedure reduces the number of dimensions in large datasets to a smaller number of dimensions while trying to keep the same amount of variance as possible. The x-axis is the first component which is a linear combination of all the features that explains the most variance. The y-axis is the second component with which explains the second most variance. By doing so, we can approximate the data in two dimensions and visualize potential linear separability.

$$PC_1 = \sum_{i=1}^{n} w_i \cdot x_i \tag{1}$$

- PC_1: The value of the first principal component for a specific data point.
- n: The number of original features in the dataset.
- x_i: The value of the i-th feature for a specific data point.
- w_i: The weight assigned to the i-th feature for PC_1. These weights are derived from the eigenvector associated with the largest eigenvalue of the covariance matrix of the data.



(a) PCA graphs

Looking at the PCA graph we see that there seems to be some separability in the data since the right cluster are all classified as spam however the left cluster seems to have both not spam and spam. We applied PCA to just the left cluster, but our results didn't change significantly, indicating that the data is not linearly separable overall. Therefore, despite some indications of separability in certain clusters, the dataset as a whole is not linearly separable, which impacts our choice of models.

3.2.2 Model Selection

After evaluating various models for our dataset, we believe the random forest model is the most suitable choice because it accounts for both linear separability and non-linearity while effectively handling high-dimensional data. Random Forest can model complex, non-linear interactions between features, which is critical since our dataset isn't perfectly linearly separable as seen in our PCA results. By averaging multiple decision trees and randomly selecting subsets of predictors for splits, random forest reduces the variance associated with single decision trees. While the decision tree model is more interpretable, single decision trees are prone to overfitting, especially with a high number of features like our model. Random forest model averages predictions and uses metrics like the Gini Index to deliver strong performance and highlight key features that are used to identify spam and no spam. Random Forest offers high predictive accuracy, built-in feature importance ranking, and reduces the impact of outliers due to averaging.

Logistic Regression model is another easy model to interpret and performs well when the data is not linearly separable; with proper regularization (i.e., ridge), it can handle datasets with some

overlap. However, PCA suggests that while there exists some inseparability, the dataset overall is partially separable, which makes performance of logistic regression less effective.

A hard-margin SVM would excel if the dataset were linearly separable, but hard-margin SVM assumes perfect separability, which does not align with the observed PCA results. While soft-margin SVM could address this, the interpretability and scalability issues remain. Compared to Random Forest, SVM provides less insight into feature importance.

Random forest model combines the interpretability of decision trees with effective overfitting prevention, balances bias and variance effectively for small datasets, and can handle both linear and non-linear patterns that the other models cannot address. Thus, we can expect an interpretable and high-performing model for our dataset compared to other alternative models.

3.2.3 Model Training

We split the training data into a training set with 85% of the data and a testing set with 15% of the data because we used the Grid Search Cross Validation method to optimize the hyperparameters of the models. Grid Search performs a combinatorial search over all possible values we defined for the model, allowing us to identify the optimal combinations of parameters. Cross validation splits the training set into smaller subsets, training the model on some subsets and validating it on others, preventing overfitting the model and allowing more reliable estimates of how well the model will perform. As mentioned above, we tuned three models: logistic regression, SVM, and random forest model to determine which performs better on the dataset.

For logistic regression, we optimized the parameters for the `C` (regularization strength), `solver` (optimization algorithm), and `max_iter` (maximum solver iterations). The best combination identified was {`C: 1000, max_iter: 100, solver: lbfgs`}, indicating a model with less penalty that converges quickly.

For the SVM, we refined the model by tuning the `C` regularization strength, `kernel` type, and `gamma` parameter. The best combination identified was {`'C': 1, 'gamma': 'scale', 'kernel': 'linear'`}, resulting in a simpler model that balances margin maximization and provides a more generalized decision boundary since the gamma isn't too high or low.

For Random Forest, we optimized the tree depth, the minimum samples to split a node, and the minimum samples per leaf node. The best combination identified was {`'max_depth': 100, 'min_samples_leaf': 1, 'min_samples_split': 7`}, indicating a model that has less depth to reduce overfitting.

3.3 Model Evaluation

3.3.1 Evaluation Metrics

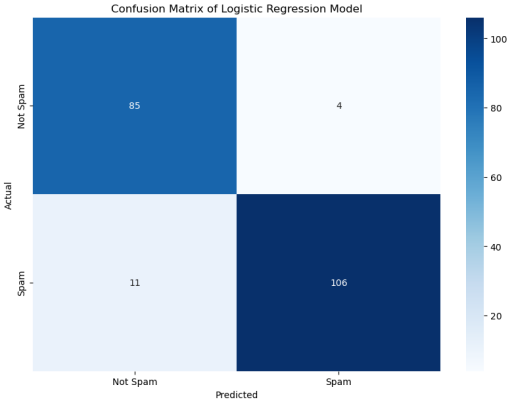
To evaluate model performance, we used accuracy and precision as the primary metrics. Accuracy was selected to provide an overall measure of model correctness and was prioritized over the F1 score due to the balanced nature of our classes since the F1 score is most useful for imbalanced datasets. After accuracy, our focus was on maximizing precision to ensure that flagged comments were highly likely to be spam. This emphasis reflects the business costs associated with reduced community engagement when legitimate comments are falsely identified as spam. By prioritizing precision, we aim to avoid the incorrect classification of legitimate comments, which allows us to maintain user trust and a positive community environment. To validate our selected model, we compared its metrics with other considered models: logistic regression, support vector machines.

To answer our secondary research question about identifying the words that are most indicative of spam, the model's feature importance was assessed using the Gini importance. This metric assesses the contribution of each feature to reducing impurity at decision nodes, providing a quantitative measure of how much each word impacts the classification process. Thus, analyzing Gini importance allows us to pinpoint the specific words that play a critical role in classifying spam comments, giving us a better understanding of the model's decision-making process and the linguistic patterns associated with spam and legitimate comments.

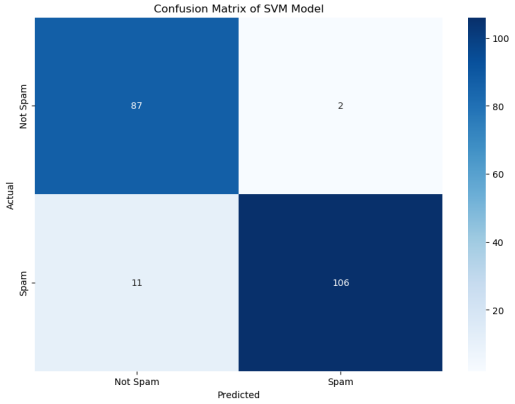
4 Results and Conclusions

4.1 Results

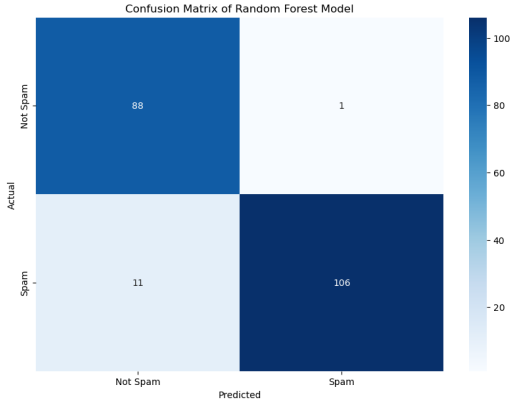
Our logistic regression model after hyperparameter tuning achieved an accuracy score of 0.9272 and precision score of 0.9636. For the SVM model, it achieved an accuracy score of 0.9369 and precision score of 0.9815. The random forest model achieved an accuracy of 0.9417 and a precision of 0.9907.



(a) Logistic Regression



(b) SVM



(c) Random Forest

In comparison with alternative models, the random forest model outperformed in both metrics. The most influential feature from the random forest had a gini score of 0.159, which was a feature for word `check` according to our TF-IDF vectorization. Other notable features were `subscribe`, `http`, `DATE_NA`, and `www` with their gini scores ranging from 0.032 to 0.0692.

4.2 Conclusion

Based on the results, we conclude that the random forest model is effective at classifying spam comments and legitimate comments, as indicated by the accuracy of 0.97. The high precision of 0.98 demonstrates the model's strong ability to minimize false positives, which was one of the primary objectives of this project. A secondary objective of this project was to identify the words that are most indicative of spam, with an emphasis on exploring hyperlink-related keywords. In our analysis of the feature importance, words such as `http` and `www` stand out among the most important, confirming that hyperlinks are indeed strong indicators of spam content. This result supports Google's recent policy change of banning hyperlinks in YouTube Shorts comments due to their association with spam.

More generally, we see that common words like `check`, `subscribe`, and `please` were also significant contributors to identifying spam. This aligns with our understanding of typical spam behaviors, where users solicit user actions such as clicking links or subscribing to channels. Meanwhile, words relating to music, such as `shakira` and `song`, were often used to identify non-spam comments. This supports our intuition that comments relevant to the video topic are more likely to be legitimate

community engagement. Finally, while this feature is not a word, the inclusion of `DATE_NA` highlights the effectiveness of our feature engineering, where we used the presence of missing date information as a predictor. In conclusion, the random forest model effectively addressed the research questions. In terms of the primary research question, the high precision and overall accuracy of the model demonstrates that it can successfully moderate spam content while ensuring that legitimate comments are not wrongly flagged. To answer the secondary research question, we identified key features such as the presence of hyperlink-related keywords and words like `check`, `subscribe`, and `please` as being indicative of spam. This aligns with known characteristics of spam comments and supports Google's policy decisions.

5 Discussion

5.1 Limitations

Despite the strong performance of the model on our dataset, its reliance on dataset-specific and context-specific features significantly limits its generalizability. The feature `DATE_NA` was highly effective in our specific dataset for detecting spam comments. However, since all YouTube comments include a timestamp, the uniqueness of `DATE_NA` in our dataset cannot be generalized to other datasets or applied effectively in broader spam detection tasks. Additionally, some of the most influential words identified by our model were directly related to the context of the music videos. For example, features like `shakira`, the name of the artist, and `song` are two of the most influential features in our model. While these words were effective indicators within our dataset, they are highly context-specific. Relying on such features means the model may underperform when predicting spam in YouTube comments unrelated to music videos. Thus, the model's dependence on these specific features limits its ability to generalize and detect spam in comments across a diverse range of YouTube content.

Following the lack of generalizability of the developed model, using TF-IDF vectorization to convert comments into numerical features further limited its applicability across different datasets. While TF-IDF is useful for identifying and highlighting essential terms, it does not capture the semantic relationships between words or the context in which they appear. It cannot effectively detect spam comments that rely on contextual cues. For example, we tested our model with the comment:

"this money song is good www"

The vectorization assigns importance to individual terms like *money* and *www*, resulting the model to classify the comment as spam. It lacked the capacity to interpret the context of each word, resulting the model to be biased toward surface-level patterns. This indicates the need for more advanced word representation techniques that can better capture the underlying meaning and intent of comments.

5.2 Future Work

Future work should explore advanced natural language process techniques to address such limitations. Methods such as word embeddings (e.g., Word2Vec, GloVe) and transformer models (e.g., BERT) can capture semantic relationships more effectively than TF-IDF by considering the order and context of words, allowing the model to understand the meaning of phrases within comments. By doing so, the model can identify spam based on semantic content and language patterns common to spam messages instead of relying on specific keywords like *www*. Nonetheless, the current model provides a strong foundation for spam detection while minimizing the risk of falsely flagging legitimate content as spam. Moreover, the model has allowed us to interpret the words that are most indicative of spam, offering valuable insights for future improvements in automated content moderation.

References

Hale, J. (2023, August 10). YouTube Shorts comments plagued by spam: What creators need to know. *Tubefilter*. Retrieved November 14, 2024, from https://www.tubefilter.com/2023/08/10/youtube-shorts-comments-spam/.

Google. (n.d.). What is spam on YouTube? *YouTube Help*. Retrieved November 14, 2024, from https://support.google.com/youtube/answer/9482362?hl=en.

Perez, S. (2024, October 4). YouTube apologizes for falsely banning channels for spam, canceling subscriptions. *TechCrunch*. Retrieved November 14, 2024, from https://techcrunch.com/2024/10/04/youtube-apologizes-for-falsely-banning-channels-for-spam-canceling-subscriptions/.

0. Setup

[1]:
```
# basics
import re
import pandas as pd
import numpy as np
import os
import string
import nltk
from spellchecker import SpellChecker
from collections import Counter
from scipy.stats import fisher_exact

# visualization
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA
from wordcloud import WordCloud
from matplotlib_venn import venn2
from sklearn.tree import plot_tree

# preprocessing
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from nltk.tokenize import TreebankWordTokenizer, PunktSentenceTokenizer,␣
↪word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.data import find
from nltk.tag import pos_tag

# models
from sklearn.model_selection import train_test_split, cross_val_score,␣
↪GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier

# evaluation
from sklearn.metrics import accuracy_score, classification_report,␣
↪confusion_matrix, recall_score, precision_score

# Ignore convergence warnings
import warnings
from sklearn.exceptions import ConvergenceWarning
warnings.filterwarnings("ignore", category=ConvergenceWarning)
```

[2]:
```
# load data
train_df = pd.read_csv('raw-dataset/train.csv')
test_df = pd.read_csv('raw-dataset/test.csv')
```

[3]:
```
# Set seed for reproducibility
np.random.seed(100)
```

1. Exploratory Data Analysis

1.1 Basic Information

[4]:
```
train_df.head()
```

[4]:

	COMMENT_ID	AUTHOR	DATE	CONTENT	VIDEO_NAME	CLASS
0	1	Brandon Pryor	2014-01-19 00:36:25	I dont even watch it anymore i just come here ...	PSY - GANGNAM STYLE(?????) M/V	0
1	2	Chelsea Yun	2015-05-23 07:17:09.691	i hate rap	Eminem - Love The Way You Lie ft. Rihanna	0
2	3	Sofia Aristizabal	2014-09-09 00:43:52	I loved, she is amazing.. OMG your eyes*_*	Katy Perry - Roar	0
3	4	said abdesalam	2015-05-24 07:35:13.754	song is bad	Eminem - Love The Way You Lie ft. Rihanna	0
4	5	crazy girl	2015-05-23 23:26:05.305	tension	LMFAO - Party Rock Anthem ft. Lauren Bennett, ...	0

[5]:
```
print(f"Shape of the training dataframe: {train_df.shape}")
print(f"Shape of the testing dataframe: {test_df.shape}")
```

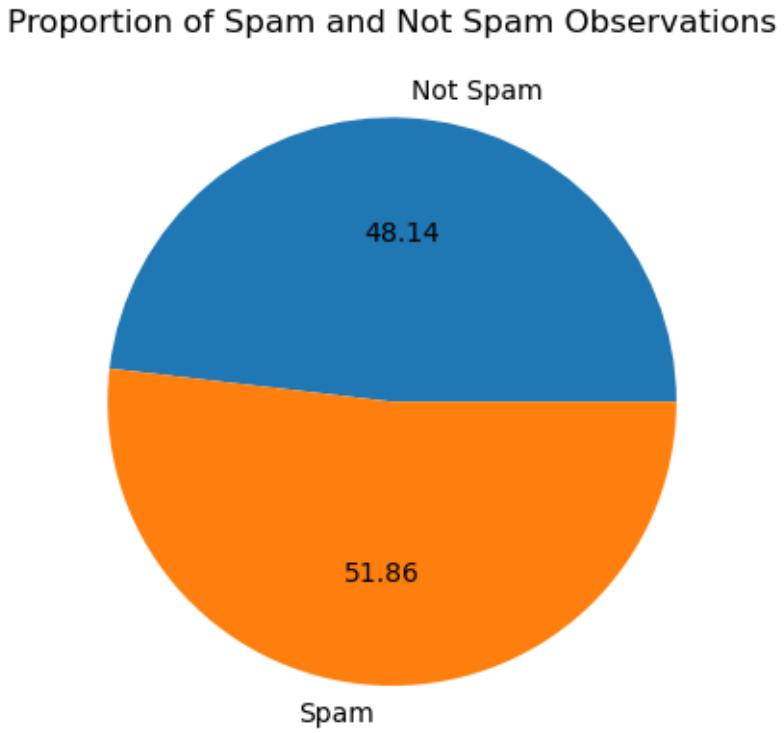
```
Shape of the training dataframe: (1369, 6)
Shape of the testing dataframe: (587, 5)
```

[6]:
```
print(f"Number of duplicates in the training data: {train_df.duplicated().
↪sum()}")
```

```
Number of duplicates in the training data: 0
```

[7]:
```
#PIE PLOT Visualizing how many observations of each class training data has
# Calculate counts for each class
spam_count = train_df[train_df['CLASS'] == 1].shape[0]
not_spam_count = train_df[train_df['CLASS'] == 0].shape[0]
```

```
# Plot with correct labels
plt.pie([not_spam_count, spam_count], labels=['Not Spam', 'Spam'], autopct="%0.
 ↪2f")
plt.title('Proportion of Spam and Not Spam Observations')
plt.show()
```



1.2 Date

[8]:
```
# Check to see how observations with NA date are distributed
missing_dates_by_class = train_df.groupby('CLASS')['DATE'].apply(lambda x: x.
 ↪isna().sum())
print("Missing Dates by Class:")
print(f"Not Spam: {missing_dates_by_class[0]}")
print(f"Spam: {missing_dates_by_class[1]}")
```

```
Missing Dates by Class:
Not Spam: 0
Spam: 170
```

1.3 Video and User Information

[9]:
```
unique_users = train_df['AUTHOR'].nunique()
print(f"Number of unique users in the training data: {unique_users}")
unique_dates = train_df['DATE'].nunique()
print(f"Number of unique dates in the training data: {unique_dates}")
unique_videos = train_df['VIDEO_NAME'].nunique()
print(f"Number of unique videos in the training data: {unique_videos}")
```

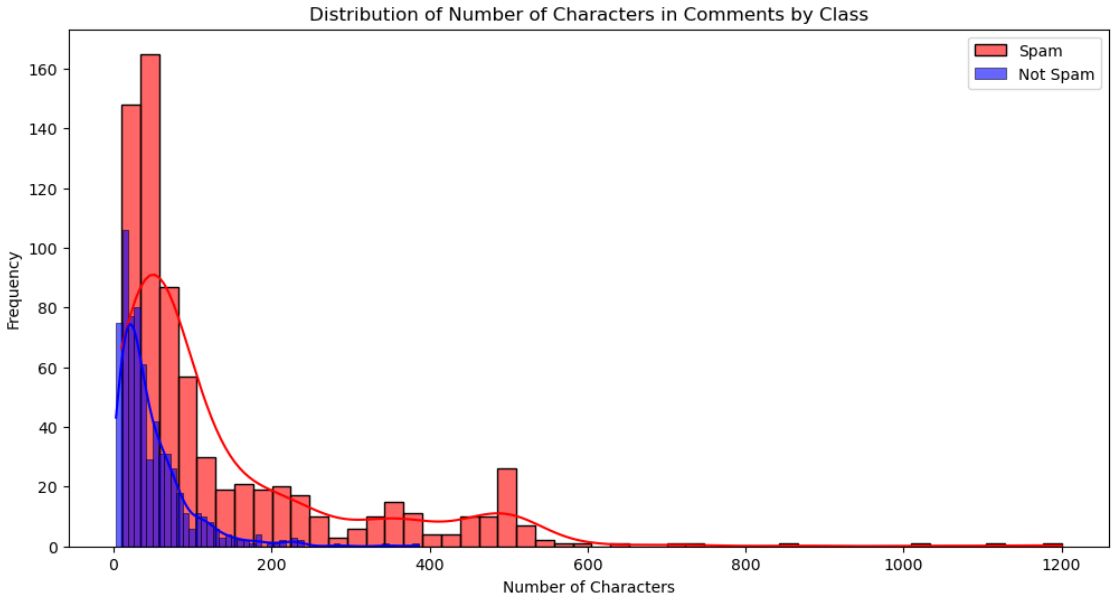
```
Number of unique users in the training data: 1267
Number of unique dates in the training data: 1199
Number of unique videos in the training data: 5
```

1.4 Comment

1.4.1 Character counts

[10]:
```
# check for number of characters in each comment
train_df['n_chars'] = train_df['CONTENT'].apply(len)
print(f"Maximum number of characters in comments: {train_df['n_chars'].max()}")
print(f"Minimum number of characters in comments: {train_df['n_chars'].min()}")
print(f"Mean number of characters in comments: {train_df['n_chars'].mean()}")

# check for number of characters based on class
spam_chars = train_df[train_df['CLASS'] == 1]['n_chars']
not_spam_chars = train_df[train_df['CLASS'] == 0]['n_chars']
print(f"Mean number of characters in spam comments: {spam_chars.mean()}")
print(f"Mean number of characters in not spam comments: {not_spam_chars.mean()}")

# Plot the distribution of the number of characters based on class
plt.figure(figsize=(12, 6))
sns.histplot(spam_chars, bins=50, kde=True, color='red', label='Spam', alpha=0.6)
sns.histplot(not_spam_chars, bins=50, kde=True, color='blue', label='Not Spam',␣
↪alpha=0.6)
plt.title('Distribution of Number of Characters in Comments by Class')
plt.xlabel('Number of Characters')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

```
Maximum number of characters in comments: 1200
Minimum number of characters in comments: 3
Mean number of characters in comments: 95.5989773557341
Mean number of characters in spam comments: 140.80845070422535
Mean number of characters in not spam comments: 46.8907435508346
```



1.4.2 Word counts

[11]:
```
# basic functions
def remove_punctuations_stop_words(text):
    # Remove '\ufeff' if present
    text = text.replace('\ufeff', '')
    # Remove punctuation from the text
    punctuation = string.punctuation.replace("'", "")
    text = text.translate(str.maketrans('', '', punctuation))
    # filter out stopwords
    words = text.split()
    stop_words = set(stopwords.words('english'))
    cleaned_words = [word for word in words if word.lower() not in stop_words]
    return ' '.join(cleaned_words)

def identify_web_related_terms(text):
    text = text.replace('www', ' www ')
    text = text.replace('https', ' https ')
    text = text.replace('  ', ' ')
    return text

def tokenize(text):
    sentence_tokenizer = TreebankWordTokenizer()
    lemmatizer = WordNetLemmatizer()
    tokenized_text = sentence_tokenizer.tokenize(text.lower())
    final = [lemmatizer.lemmatize(word) for word in tokenized_text]
    return final
```

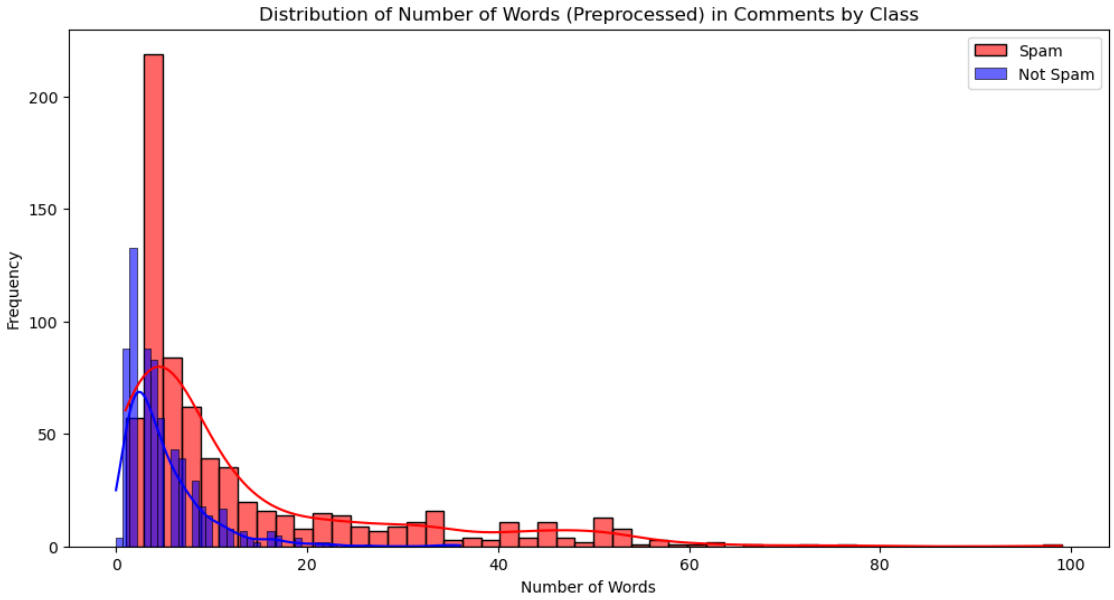
```
def join_tokens(text):
    return ' '.join(text)

def clean_text(text):
    text = remove_punctuations_stop_words(text)
    text = identify_web_related_terms(text)
    text = tokenize(text)
    return join_tokens(text)

def clean_text_eda(text):
    text = remove_punctuations_stop_words(text)
    text = identify_web_related_terms(text)
    text = tokenize(text)
    return text

def token_to_string(tokens):
    text = ' '.join(tokens)
    text = text.replace('  ', ' ')
    return text
```

[12]:
```
train_df['tokenized'] = train_df['CONTENT'].apply(clean_text_eda)
train_df['cleaned'] = train_df['tokenized'].apply(token_to_string)
```

[13]:
```
# Calculate the number of words in each comment
train_df['n_words'] = train_df['tokenized'].apply(len)
print(f"Maximum number of words in comments: {train_df['n_words'].max()}")
print(f"Minimum number of words in comments: {train_df['n_words'].min()}")
print(f"Mean number of words in comments: {train_df['n_words'].mean()}")

# Check for number of words based on class
spam_words = train_df[train_df['CLASS'] == 1]['n_words']
not_spam_words = train_df[train_df['CLASS'] == 0]['n_words']
print(f"Mean number of words in spam comments: {spam_words.mean()}")
print(f"Mean number of words in not spam comments: {not_spam_words.mean()}")

# Plot the distribution of the number of words based on class
plt.figure(figsize=(12, 6))
sns.histplot(spam_words, bins=50, kde=True, color='red', label='Spam', alpha=0.6)
sns.histplot(not_spam_words, bins=50, kde=True, color='blue', label='Not Spam',␣
 ↪alpha=0.6)
plt.title('Distribution of Number of Words (Preprocessed) in Comments by Class')
plt.xlabel('Number of Words')
plt.ylabel('Frequency')
plt.legend()
plt.show()
```

```
Maximum number of words in comments: 99
Minimum number of words in comments: 0
```

```
Mean number of words in comments: 9.16581446311176
Mean number of words in spam comments: 13.06338028169014
Mean number of words in not spam comments: 4.966616084977238
```



1.4.3 Frequent words

[14]:
```
# Initial word analysis before preprocessing
# Make a copy of train_df to avoid modifying the original DataFrame
train_df_copy = train_df.copy()

# Calculate overall most common words
all_text = " ".join(train_df_copy['CONTENT'].dropna())
overall_word_counts = Counter(all_text.split())
overall_common = overall_word_counts.most_common(50)  # Top 50 words

# Function to count the most common words by class
def count_words_by_class(group):
    # Combine all text in this class into a single string
    all_text = " ".join(group.dropna())
    # Tokenize and count words
    word_counts = Counter(all_text.split())
    # Return the most common words
    return word_counts.most_common(50)  # Top 50 words per class

# Group by CLASS and apply the function
class_word_counts = train_df_copy.groupby('CLASS')['CONTENT'].
 ↪apply(count_words_by_class)
```

```
# Create a DataFrame for results
df_common_words = pd.DataFrame({
    'Overall Words': [word for word, _ in overall_common],
    'Overall Counts': [count for _, count in overall_common],
    'Class 0 Words': [word for word, _ in class_word_counts[0]],
    'Class 0 Counts': [count for _, count in class_word_counts[0]],
    'Class 1 Words': [word for word, _ in class_word_counts[1]],
    'Class 1 Counts': [count for _, count in class_word_counts[1]],
})

# Print the DataFrame
with pd.option_context('display.max_rows', None, 'display.max_columns', None):
    print(df_common_words)
```

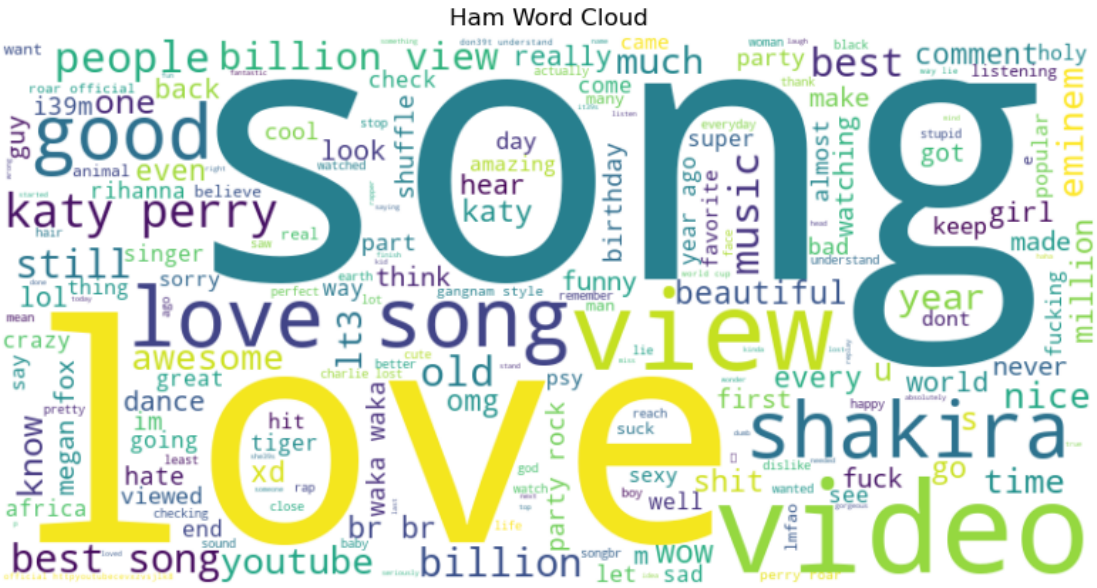
	Overall Words	Overall Counts	Class 0 Words	Class 0 Counts	Class 1 Words	\
0	this	411	the	153	and	
1	the	408	I	138	to	
2	and	404	this	138	out	
3	to	402	is	125	my	
4	I	386	song	88	this	
5	a	333	to	78	a	
6	out	330	love	70	the	
7	my	304	and	70	I	
8	on	266	a	66	on	
9	you	255		63	you	
10	is	194	in	54	Check	
11	of	193	of	50	check	
12	check	184	so	49	.	
13	Check	179	that	47	of	
14	.	178	like	46	video	
15	video	173	i	44	for	
16	it	150	it	41	it	
17	for	150	video	39	me	
18	me	129	you	36	You	
19	i	127	2	35	YouTube:	
20	in	122	This	32	if	
21	like	122	my	30	i	
22	song	116	Katy	28	can	
23		115	was	28	just	
24	so	113	billion	27	like	
25	You	103	for	26	subscribe	
26	just	98	song	24	is	
27	love	94	views	24	MY	
28	if	92	on	23	in	
29	YouTube:	91	she	23	will	
30	can	89	when	23	your	
31	that	87	are	23	so	
32	have	79	me	22	have	

33	will	78	but	22	music
34	your	77	just	21	channel
35	music	75	best	21	guys
36	subscribe	72	Love	20	please
37	be	71	get	19	be
38	MY	70	years	19	money
39	but	69	because	18	at
40	/><br	69	people	18	/><br
41	at	67	not	18	as
42	from	64	with	17	from
43	guys	62	only	17	
44	channel	62	The	17	make
45	please	60	have	15	up
46	are	60	all	15	OUT
47	get	59	her	15	Please
48	The	57	Shakira	15	but
49	with	56	old	15	do

	Class 1 Counts
0	334
1	324
2	323
3	274
4	273
5	267
6	255
7	248
8	243
9	219
10	179
11	178
12	175
13	143
14	134
15	124
16	109
17	107
18	93
19	91
20	84
21	83
22	77
23	77
24	76
25	72
26	69
27	69
28	68

```
29              67
30              66
31              64
32              64
33              63
34              62
35              61
36              60
37              59
38              55
39              55
40              55
41              52
42              52
43              52
44              50
45              50
46              50
47              47
48              47
49              44
```

[15]:
```
ham_wc = WordCloud(width=800, height=400, background_color='white').
 ↪generate(train_df[train_df['CLASS'] == 0]['cleaned'].str.cat(sep=" "))
plt.figure(figsize=(10, 5))
plt.imshow(ham_wc, interpolation="bilinear")
plt.axis("off")
plt.title("Ham Word Cloud")
plt.show()

spam_wc = WordCloud(width=800, height=400, background_color='white').
 ↪generate(train_df[train_df['CLASS'] == 1]['cleaned'].str.cat(sep=" "))
plt.figure(figsize=(10, 5))
plt.imshow(spam_wc, interpolation="bilinear")
plt.axis("off")
plt.title("Spam Word Cloud")
plt.show()
```



[16]:
```
# Filter rows for each class
class_0_words = ' '.join(train_df[train_df['CLASS'] == 0]['cleaned']).split()
class_1_words = ' '.join(train_df[train_df['CLASS'] == 1]['cleaned']).split()

##Word frequency
# Count word occurrences in each class
class_0_word_counts = Counter(class_0_words)
```

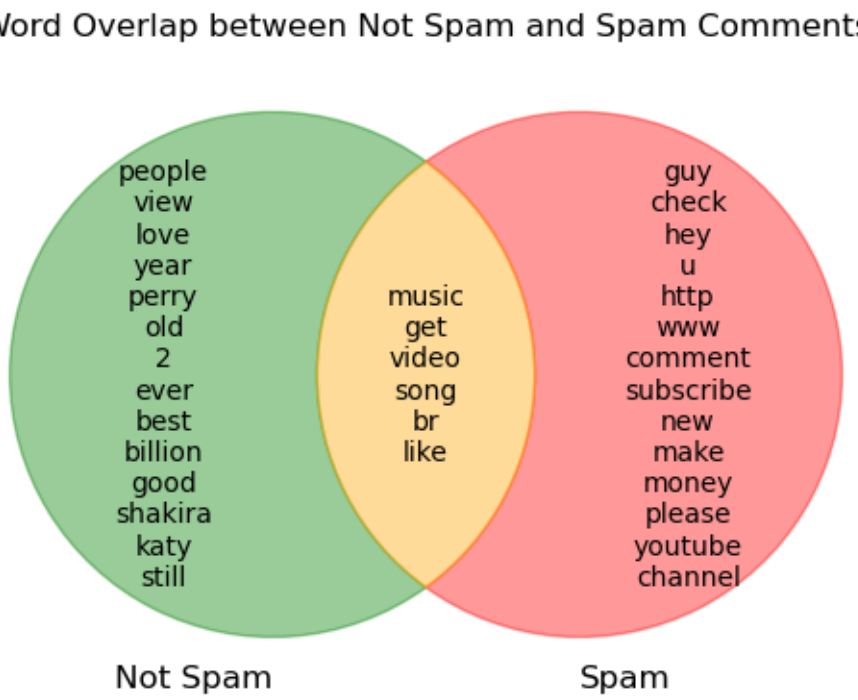
```
class_1_word_counts = Counter(class_1_words)

# Display the most common words in each class
print("Most common words in class 0:")
print(class_0_word_counts.most_common(10))  # Adjust the number to see more or␣
 ↪fewer words
print("\nMost common words in class 1:")
print(class_1_word_counts.most_common(10))

# Find the most frequent words in each class that do not appear in the other␣
 ↪class
class_0_only_words = {word: count for word, count in class_0_word_counts.items()␣
 ↪if word not in class_1_word_counts}
class_1_only_words = {word: count for word, count in class_1_word_counts.items()␣
 ↪if word not in class_0_word_counts}

# Get the top 10 unique words in each class
class_0_unique_top_10 = Counter(class_0_only_words).most_common(10)
class_1_unique_top_10 = Counter(class_1_only_words).most_common(10)

# Display the most frequent words unique to each class
print("\nMost frequent words in class 0 that do not appear in class 1:")
print(class_0_unique_top_10)

print("\nMost frequent words in class 1 that do not appear in class 0:")
print(class_1_unique_top_10)
```

```
Most common words in class 0:
[('song', 162), ('love', 104), ('view', 60), ('like', 60), ('video', 56),
('best', 42), ('billion', 40), ('katy', 38), ('2', 36), ('shakira', 32)]

Most common words in class 1:
[('check', 404), ('video', 198), ('please', 159), ('subscribe', 157),
('youtube', 137), ('channel', 126), ('like', 117), ('http', 111), ('music', 89),
('guy', 85)]

Most frequent words in class 0 that do not appear in class 1:
[('xd', 11), ('fox', 10), ('megan', 9), ('fuck', 9), ('lost', 7), ('viewed', 7),
('charlie', 6), ('end', 6), ('official', 5), ('favorite', 5)]

Most frequent words in class 1 that do not appear in class 0:
[('cover', 41), ('moneygqcom', 33), ('home', 32), ('playlist', 30), ('website',
30), ('working', 30), ('online', 29), ('free', 28), ('mixtape', 28),
('zonepacom', 27)]
```

[17]:
```
# Create a set of the top N words in each class (N = 20 for example)
top_n = 20
```

```
class_0_top_words = {word for word, _ in class_0_word_counts.most_common(top_n)}
class_1_top_words = {word for word, _ in class_1_word_counts.most_common(top_n)}
# Find the overlapping and unique words
overlapping_words = class_0_top_words & class_1_top_words
class_0_unique_words = class_0_top_words - class_1_top_words
class_1_unique_words = class_1_top_words - class_0_top_words

plt.clf()
# Plot the Venn diagram of word overlap
overlap_fig, ax = plt.subplots()
venn = venn2(subsets=(len(class_0_unique_words), len(class_1_unique_words),␣
 ↪len(overlapping_words)), set_labels=('Not Spam', 'Spam'))
venn.get_label_by_id('10').set_text('\n'.join(list(class_0_unique_words)))
venn.get_label_by_id('01').set_text('\n'.join(list(class_1_unique_words)))
venn.get_label_by_id('11').set_text('\n'.join(list(overlapping_words)))
# Set colors manually
venn.get_patch_by_id('10').set_color('green')  # Class 0 (Not Spam)
venn.get_patch_by_id('01').set_color('red')    # Class 1 (Spam)
venn.get_patch_by_id('11').set_color('orange')  # Overlap (optional, can change␣
 ↪to any color)
plt.title("Word Overlap between Not Spam and Spam Comments")
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```



1.4.4 Typos and Emojis

[18]:
```
spell = SpellChecker()
def find_typos(text):
    # Tokenize text and tag with parts of speech
    words = nltk.word_tokenize(text)
    pos_tags = nltk.pos_tag(words)
    typos = [
        word for word, pos in pos_tags
        if word.lower() not in spell
        and pos not in ('NNP', 'NNPS')
        and not word.isdigit()
    ]

    # Join typos list if any, else return empty string
    return ', '.join(typos) if typos else ''


# Function to extract emojis from text
def find_emojis(text):
    # Regular expression pattern to match emojis
    emoji_pattern = re.compile("["
                           u"\U0001F600-\U0001F64F"  # Emoticons
                           u"\U0001F300-\U0001F5FF"  # Symbols & pictographs
                           u"\U0001F680-\U0001F6FF"  # Transport & map symbols
                           u"\U0001F1E0-\U0001F1FF"  # Flags (iOS)
                           u"\U00002700-\U000027BF"  # Dingbats
                           u"\U000024C2-\U0001F251"  # Enclosed characters
                           "]+", flags=re.UNICODE)
    # Find all emojis in the text
    emojis = emoji_pattern.findall(text)
    # Join emojis list if any, else return empty string
    return ' '.join(emojis) if emojis else ''

# Create a table showing the top 10 typos in each class and their counts
def get_top_typos(class_typos):
    all_typos = ', '.join(class_typos).split(', ')
    typo_counts = Counter(all_typos)
    return typo_counts.most_common(10)

# Create a table showing the top 10 emojis in each class and their counts
def get_top_emojis(class_emojis):
    all_emojis = ' '.join(class_emojis).split()
    emoji_counts = Counter(all_emojis)
    return emoji_counts.most_common(11)
```

[19]:
```
# Apply the typo detection function
train_df['typos'] = train_df['cleaned'].apply(find_typos)
```

```
class_0_top_typos = get_top_typos(train_df[train_df['CLASS'] == 0]['typos'])
class_1_top_typos = get_top_typos(train_df[train_df['CLASS'] == 1]['typos'])
# Create a DataFrame to display the top typos in each class
top_typos_df = pd.DataFrame({
    'Not Spam Typos': [typo for typo, _ in class_0_top_typos],
    'Not Spam Counts': [count for _, count in class_0_top_typos],
    'Spam Typos': [typo for typo, _ in class_1_top_typos],
    'Spam Counts': [count for _, count in class_1_top_typos]
})

print(top_typos_df)
top_typos_df

# Create the 'has_typo' column (1 if typos exist, 0 if not)
train_df['has_typo'] = train_df['typos'].apply(lambda x: 1 if len(x) > 0 else 0)

# Calculate the proportion of comments with typos in each class
class_0_proportion_spam = (train_df[train_df['CLASS'] == 0]['has_typo'] == 1).
 ↪mean()
class_1_proportion_spam = (train_df[train_df['CLASS'] == 1]['has_typo'] == 1).
 ↪mean()

print(f"Proportion of comments with typos in Not Spam:␣
 ↪{class_0_proportion_spam}")
print(f"Proportion of comments with typos in Spam: {class_1_proportion_spam}")
```

```
  Not Spam Typos  Not Spam Counts  Spam Typos  Spam Counts
0                            290     youtube          137
1           katy              38        http          111
2        shakira              32                      109
3             br              23         www           84
4        youtube              18          br           70
5         eminem              16        i39m           41
6           waka              16          im           38
7            lt3              15  moneygqcom           33
8             's              12    playlist           30
9            www              12     mixtape           28
Proportion of comments with typos in Not Spam: 0.5599393019726859
Proportion of comments with typos in Spam: 0.8464788732394366
```

[20]:
```
# Typos Fisher's Exact Test
# Apply the typo detection function
train_df['typos'] = train_df['cleaned'].apply(find_typos)

# Create a new column 'has_typo' (1 if typos exist, 0 if not)
train_df['has_typo'] = train_df['typos'].apply(lambda x: 1 if len(x) > 0 else 0)
```

```
# Create the contingency table
contingency_table_typos = pd.crosstab(train_df['has_typo'], train_df['CLASS'],
                                      rownames=['Has Typo'], colnames=['Class'])

# Print the contingency table
print("Contingency Table for Typos vs. Class:")
print(contingency_table_typos)

# Perform Fisher's Exact Test
_, p_value_typos = fisher_exact(contingency_table_typos)
print(f"\nFisher's Exact Test p-value for Typos vs. Class: {p_value_typos:.10f}")
```

```
Contingency Table for Typos vs. Class:
Class       0    1
Has Typo
0         290  109
1         369  601

Fisher's Exact Test p-value for Typos vs. Class: 0.0000000000
```

[21]:
```
train_df['emojis'] = train_df['CONTENT'].apply(find_emojis)
class_0_top_emojis = get_top_emojis(train_df[train_df['CLASS'] == 0]['emojis'])
class_1_top_emojis = get_top_emojis(train_df[train_df['CLASS'] == 1]['emojis'])
# Create a DataFrame to display the top emojis in each class
top_emojis_df = pd.DataFrame({
    'Non Spam Emojis': [emoji for emoji, _ in class_0_top_emojis],
    'Non Spam Counts': [count for _, count in class_0_top_emojis],
    'Spam Emojis': [emoji for emoji, _ in class_1_top_emojis],
    'Spam Counts': [count for _, count in class_1_top_emojis]
})

# Skip the first row and show the next 10 rows
next_10_entries = top_emojis_df.iloc[1:11]

# Print the DataFrame with a title
print("Unique Emoji Clusters by Class")
print(next_10_entries)

# Total number of entries in train_df for each class
total_entries_class_0 = len(train_df[train_df['CLASS'] == 0])
total_entries_class_1 = len(train_df[train_df['CLASS'] == 1])

# Count of no-emoji entries (from the first row of the table)
no_emoji_class_0_count = top_emojis_df.iloc[0]["Non Spam Counts"]
no_emoji_class_1_count = top_emojis_df.iloc[0]["Spam Counts"]
```

```
Unique Emoji Clusters by Class
   Non Spam Emojis  Non Spam Counts Spam Emojis  Spam Counts
```

```
1                               2                       2
2                                2              2
3                       2                2
4                            1                       1
5                            1                 1
6                          1                   1
7                               1                      1
8                               1                      1
9                               1                      1
10                             1                      1
```

[22]:

```python
# Emojis Fisher's Exact Test
# Apply the emoji detection function
train_df['emojis'] = train_df['CONTENT'].apply(find_emojis)

# Create a new column 'has_emoji' (1 if emojis exist, 0 if not)
train_df['has_emoji'] = train_df['emojis'].apply(lambda x: 1 if len(x) > 1 else␣
↪0)

# Create the contingency table
contingency_table = pd.crosstab(train_df['has_emoji'], train_df['CLASS'],
                                rownames=['Has Emoji'], colnames=['Class'])

# Print the contingency table
print("Contingency Table:")
print(contingency_table)

# Perform Fisher's Exact Test
_, p_value = fisher_exact(contingency_table)
print(f"\nFisher's Exact Test p-value: {p_value:.10f}")
```

```
Contingency Table:
Class        0    1
Has Emoji
0          630  695
1           29   15

Fisher's Exact Test p-value: 0.0206600581
```

[23]:

```python
# www or https Fisher's Exact Test
# Create a function to check if a comment contains 'www' or 'http'
def contains_http_www(text):
    return 'www' in text or 'http' in text

# Apply this function to your dataset to create a new column
train_df['has_www_http'] = train_df['CONTENT'].apply(contains_http_www)

# Create the contingency table comparing 'has_www_http' and 'CLASS'
```

```
contingency_table = pd.crosstab(train_df['has_www_http'], train_df['CLASS'])

# Display the contingency table
print("Contingency Table:")
print(contingency_table)

# Perform Fisher's Exact Test
_, p_value = fisher_exact(contingency_table)  # Exclude the "All" row and column

# Print the p-value
print(f"\nFisher's Exact Test p-value: {p_value:.4f}")
```

```
Contingency Table:
CLASS           0    1
has_www_http
False         651  580
True            8  130

Fisher's Exact Test p-value: 0.0000
```

2. Data Preprocessing

2.0 Setup

[24]:
```
train_df = pd.read_csv('raw-dataset/train.csv')
test_df = pd.read_csv('raw-dataset/test.csv')
```

2.1 Feature Engineering

[25]:
```
# date column into binary
train_df['DATE'] = train_df['DATE'].apply(lambda x: 1 if type(x) == float else 0)
test_df['DATE'] = test_df['DATE'].apply(lambda x: 1 if type(x) == float else 0)
```

2.2 Comment Preprocessing

[26]:
```
train_df['cleaned_text'] = train_df['CONTENT'].apply(clean_text)
test_df['cleaned_text'] = test_df['CONTENT'].apply(clean_text)
```

2.3 Word Vectorization: TF-IDF Vectorizer

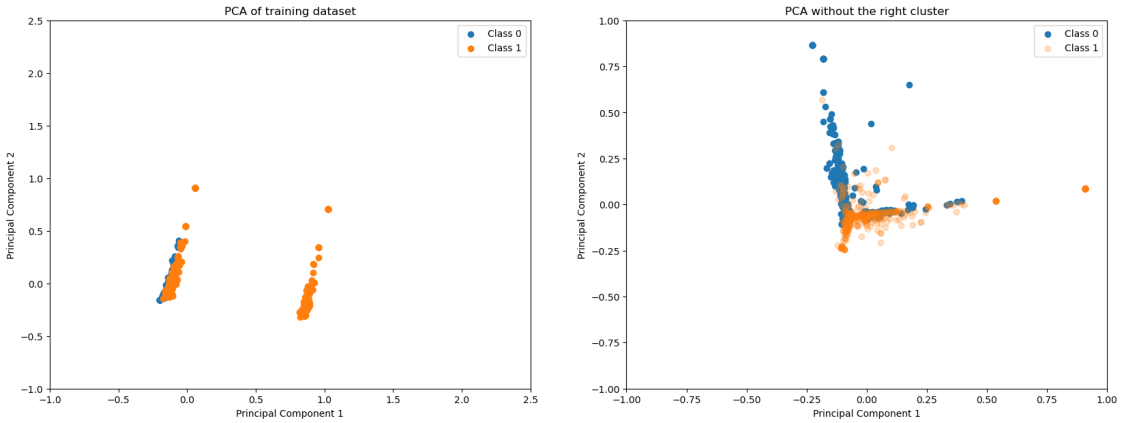
[27]:
```
tf_vectorizer = TfidfVectorizer()
x_1 = tf_vectorizer.fit_transform(list(train_df['cleaned_text'])).toarray()
x_2 = np.array(train_df['DATE']).reshape(-1, 1)
x = np.hstack((x_1, x_2))
y = train_df['CLASS']
```

3. Model Development and Training

3.1 Linear Separability: Principal Component Analysis

[28]:
```
# Create subplots
fig, axes = plt.subplots(1, 2, figsize=(20, 7))

# PCA
pca_tf_idf = PCA(n_components=2)
x_pca_tf_idf = pca_tf_idf.fit_transform(x)

# Plot
axes[0].set_title('PCA of training dataset')
for class_value in np.unique(y):
    axes[0].scatter(x_pca_tf_idf[y == class_value, 0], x_pca_tf_idf[y ==␣
 ↪class_value, 1], label=f'Class {class_value}')
axes[0].set_xlim(-1, 2.5)
axes[0].set_ylim(-1, 2.5)
axes[0].set_xlabel('Principal Component 1')
axes[0].set_ylabel('Principal Component 2')
axes[0].legend()

# Filtered PCA
right_cluster_threshold = 0.5

pca_tf_idf_temp = PCA(n_components=2)
x_pca_temp = pca_tf_idf_temp.fit_transform(x)

left_cluster_mask = x_pca_temp[:, 0] < right_cluster_threshold
x_tf_idf_left_cluster = x[left_cluster_mask]
y_left_cluster = y[left_cluster_mask]

pca_tf_idf_filtered = PCA(n_components=2)
x_pca_filtered = pca_tf_idf_filtered.fit_transform(x_tf_idf_left_cluster)

axes[1].set_title('PCA without the right cluster')
for class_value in np.unique(y_left_cluster):
    if class_value == 1:
        # Make Class 1 semi-transparent
        axes[1].scatter(x_pca_filtered[y_left_cluster == class_value, 0],
                        x_pca_filtered[y_left_cluster == class_value, 1],
                        label=f'Class {class_value}', alpha=0.25)
    else:
        # Class 0 remains opaque
        axes[1].scatter(x_pca_filtered[y_left_cluster == class_value, 0],
                        x_pca_filtered[y_left_cluster == class_value, 1],
                        label=f'Class {class_value}', alpha=1)
```

```
axes[1].set_xlim(-1, 1)
axes[1].set_ylim(-1, 1)
axes[1].set_xlabel('Principal Component 1')
axes[1].set_ylabel('Principal Component 2')
axes[1].legend()

plt.show()
```



3.2 Linear Separability: Hard Margin Support Vector Machine

TF-IDF

[29]:
```
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.15)
```

[30]:
```
hard_margin_svm = SVC(kernel='linear', C=100000000000)

hard_margin_svm.fit(x_train, y_train)
y_pred = hard_margin_svm.predict(x_test)

hard_margin_svm_accuracy_score = accuracy_score(y_test, y_pred)
hard_margin_svm_accuracy_score
```

[30]:
```
0.912621359223301
```

3.3 Model Hyperparameter Tuning

[31]:
```
lr_model = LogisticRegression(random_state=42)

param_grid = {
    'C': [100, 500, 1000],
    'solver': ['lbfgs', 'saga'],
    'max_iter': [100, 1000, 3000]
}
```

```
grid_search_lr = GridSearchCV(estimator=lr_model, param_grid=param_grid, cv=5,␣
↪scoring='accuracy')
grid_search_lr.fit(x_train, y_train)
grid_search_lr.best_params_
```

[31]:
```
{'C': 1000, 'max_iter': 100, 'solver': 'lbfgs'}
```

[32]:
```
svm = SVC(random_state=42)

param_svm_grid = {
    'C': [0.1, 1, 10, 100],
    'kernel': ['linear', 'rbf'],
    'gamma': ['scale', 'auto']
}
grid_search_svm = GridSearchCV(estimator=svm, param_grid=param_svm_grid, cv=5,␣
↪scoring='accuracy', n_jobs=-1)
grid_search_svm.fit(x_train, y_train)
grid_search_svm.best_params_
```

[32]:
```
{'C': 1, 'gamma': 'scale', 'kernel': 'linear'}
```

[33]:
```
rf = RandomForestClassifier(random_state=42)

param_rf_grid = {
    'max_depth': [50, 100, 150, None],
    'min_samples_split': [2, 4, 5, 6, 7, 10],
    'min_samples_leaf': [1, 2, 4]
}

grid_search_random_forest = GridSearchCV(estimator=rf, param_grid=param_rf_grid,␣
↪cv=5, scoring='accuracy', n_jobs=-1)
grid_search_random_forest.fit(x_train, y_train)
grid_search_random_forest.best_params_
```

[33]:
```
{'max_depth': 100, 'min_samples_leaf': 1, 'min_samples_split': 7}
```

3.4 Model Comparison

[34]:
```
lr_best_model = grid_search_lr.best_estimator_
svm_best_model = grid_search_svm.best_estimator_
rf_best_model = grid_search_random_forest.best_estimator_

print('SVC:', grid_search_lr.best_score_)
print('LR:', grid_search_svm.best_score_)
print('RF:', grid_search_random_forest.best_score_)
```

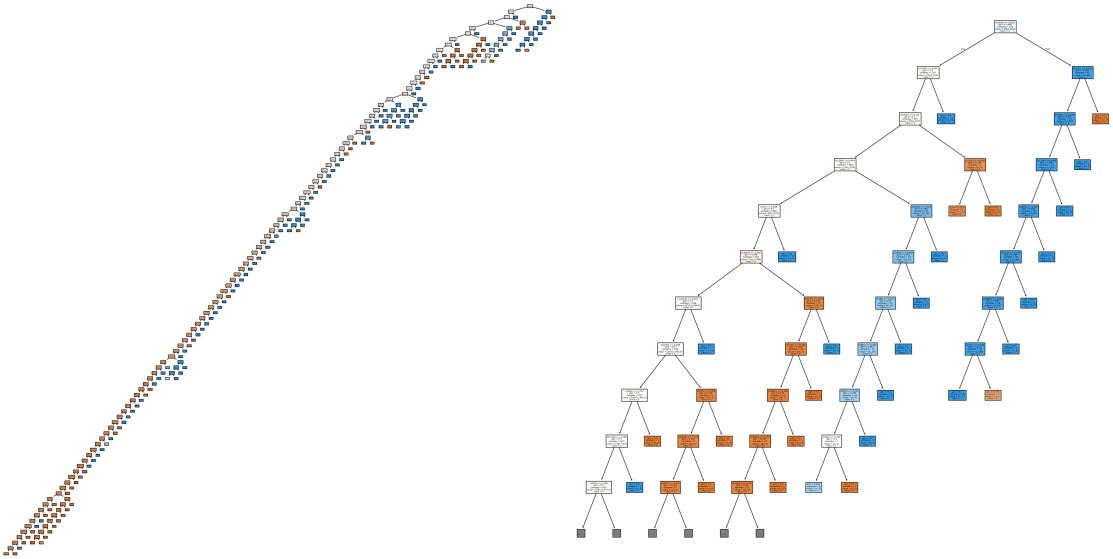
```
SVC: 0.9320704454639633
```

```
LR: 0.9234719550096196
RF: 0.9415458043510434
```

4. Model Evaluation

4.1 Sample Tree Visualization

[35]:
```
rf_best_model = grid_search_random_forest.best_estimator_
sample_tree = rf_best_model.estimators_[0]

fig, axes = plt.subplots(1, 2, figsize=(30, 15))

# Plot the full tree on the left, zoomed-in on the right
axes[0].set_title('Full Decision Tree')
plot_tree(sample_tree, filled=True, class_names=['0', '1'], ax=axes[0])
axes[1].set_title('Zoomed-In Decision Tree')
plot_tree(sample_tree, max_depth=10, filled=True, class_names=['0', '1'],␣
 ↪ax=axes[1])

plt.tight_layout()
plt.show()
```



4.2 Influential Features

[36]:
```
def get_top_n_features(model, feature_names, n):
    # Get feature importances
    importances = model.feature_importances_

    # Get indices of the top n features
```

```
    indices = np.argsort(importances)[-n:][::-1]

    top_features = []
    for i in indices:
        feature_name = "DATE_NA" if i == 3125 else feature_names[i]
        top_features.append((feature_name, importances[i]))

    return top_features

feature_names = tf_vectorizer.get_feature_names_out()
top_20_features = get_top_n_features(rf_best_model, feature_names, 20)
print(f"Top 20 features with Gini scores:")
for feature in top_20_features:
    print(feature)
```

```
Top 20 features with Gini scores:
('check', 0.15918269866012644)
('subscribe', 0.06928994945317653)
('http', 0.04391652882175408)
('DATE_NA', 0.04285564457353054)
('channel', 0.03713704939460256)
('www', 0.03221683734697302)
('please', 0.03158242462850183)
('youtube', 0.02998429575172788)
('video', 0.026218570048555868)
('song', 0.016297776281704503)
('love', 0.01299443528084477)
('subscriber', 0.012571154021043198)
('comment', 0.008990714441932317)
('new', 0.008968737684407434)
('view', 0.008388127984190284)
('follow', 0.007832165769455546)
('money', 0.007781176843633006)
('shakira', 0.006963192207061184)
('share', 0.006735201075482655)
('give', 0.006142743419464154)
```

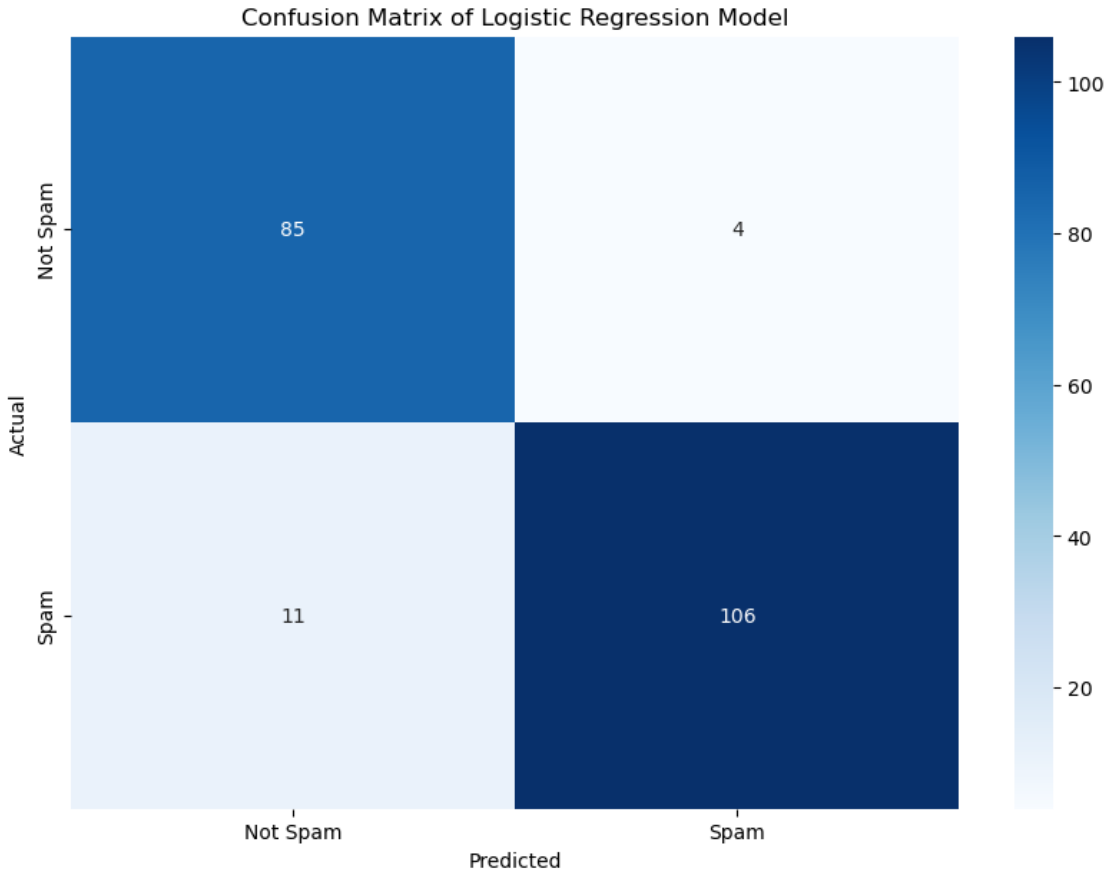
4.3 Metrics Evaluation

4.3.1 Logistic Regression

[37]:
```
best_lr_model = grid_search_lr.best_estimator_
y_val_pred = best_lr_model.predict(x_test)
val_accuracy = accuracy_score(y_test, y_val_pred)
recall = recall_score(y_test, y_val_pred)
precision = precision_score(y_test, y_val_pred)
print(f"Validation Accuracy: {val_accuracy:.4f}")
print(f"Recall: {recall:.4f}")
print(f"Precision: {precision:.4f}")
```

```
conf_matrix = confusion_matrix(y_test, y_val_pred)

TN, FP, FN, TP = conf_matrix.ravel()

# Calculate FNR and FPR
FNR = FN / (FN + TP)
FPR = FP / (FP + TN)

print(f"False Negative Rate (FNR): {FNR}")
print(f"False Positive Rate (FPR): {FPR}")

# Plot the confusion matrix
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Not␣
 ↪Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix of Logistic Regression Model')
plt.show()
```

```
Validation Accuracy: 0.9272
Recall: 0.9060
Precision: 0.9636
False Negative Rate (FNR): 0.09401709401709402
False Positive Rate (FPR): 0.0449438202247191
```

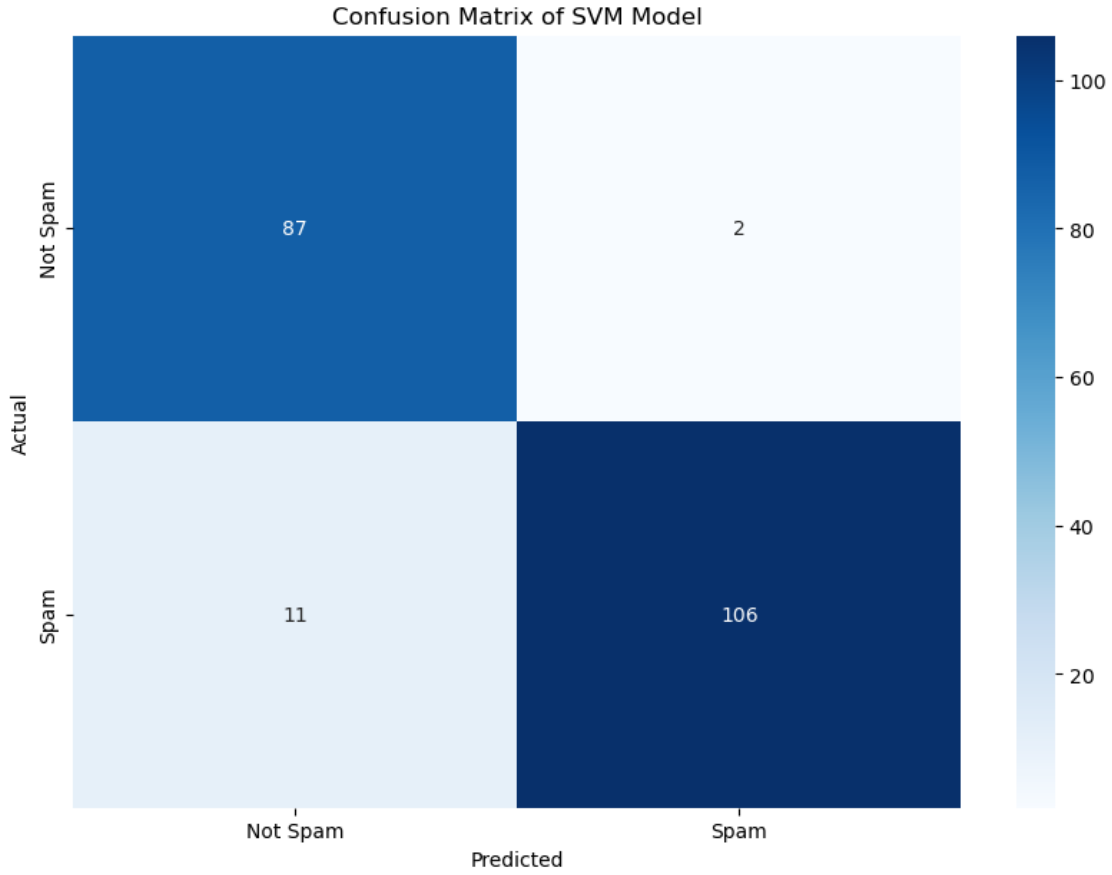


4.3.2 Support Vector Machine

[38]:
```
best_svm_model = grid_search_svm.best_estimator_
y_val_pred = best_svm_model.predict(x_test)
val_accuracy = accuracy_score(y_test, y_val_pred)
recall = recall_score(y_test, y_val_pred)
precision = precision_score(y_test, y_val_pred)
print(f"Test Accuracy: {val_accuracy}")
print(f"Recall: {recall}")
print(f"Precision: {precision}")

conf_matrix = confusion_matrix(y_test, y_val_pred)

TN, FP, FN, TP = conf_matrix.ravel()

# Calculate FNR and FPR
FNR = FN / (FN + TP)
FPR = FP / (FP + TN)

print(f"False Negative Rate (FNR): {FNR}")
```

```
print(f"False Positive Rate (FPR): {FPR}")

# Plot the confusion matrix
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Not␣
 ↪Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix of SVM Model')
plt.show()
```

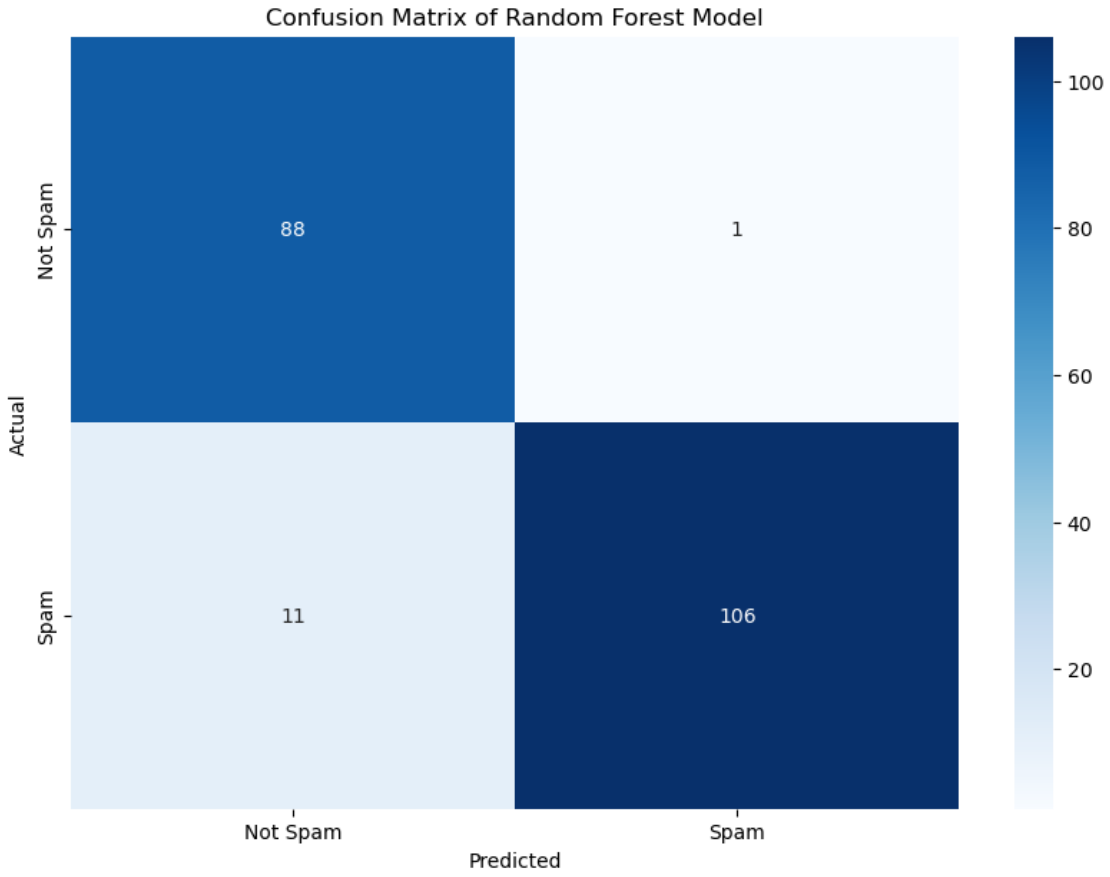
```
Test Accuracy: 0.9368932038834952
Recall: 0.905982905982906
Precision: 0.9814814814814815
False Negative Rate (FNR): 0.09401709401709402
False Positive Rate (FPR): 0.02247191011235955
```



4.3.3 Random Forest Model

[39]:
```
best_model_rf = grid_search_random_forest.best_estimator_
y_val_pred = best_model_rf.predict(x_test)
val_accuracy = accuracy_score(y_test, y_val_pred)
recall = recall_score(y_test, y_val_pred)
precision = precision_score(y_test, y_val_pred)
print(f"Test Accuracy: {val_accuracy}")
print(f"Recall: {recall}")
print(f"Precision: {precision}")

conf_matrix = confusion_matrix(y_test, y_val_pred)

TN, FP, FN, TP = conf_matrix.ravel()

# Calculate FNR and FPR
FNR = FN / (FN + TP)
FPR = FP / (FP + TN)

print(f"False Negative Rate (FNR): {FNR}")
print(f"False Positive Rate (FPR): {FPR}")

# Plot the confusion matrix
plt.figure(figsize=(10, 7))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=['Not␣
 ↪Spam', 'Spam'], yticklabels=['Not Spam', 'Spam'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix of Random Forest Model')
plt.show()
```

```
Test Accuracy: 0.941747572815534
Recall: 0.905982905982906
Precision: 0.9906542056074766
False Negative Rate (FNR): 0.09401709401709402
False Positive Rate (FPR): 0.011235955056179775
```



5. Final Prediction

[40]:
```
test_df['cleaned_text'] = test_df['CONTENT'].apply(clean_text)
test_df['DATE'] = test_df['DATE'].apply(lambda x: 1 if type(x) == float else 0)
tf_vectorizer = TfidfVectorizer()
tf_vectorizer.fit(list(train_df['cleaned_text']))

x_1 = tf_vectorizer.transform(list(train_df['cleaned_text'])).toarray()
x_2 = np.array(train_df['DATE']).reshape(-1, 1)
x = np.hstack((x_1, x_2))
y = train_df['CLASS']

x_test_1 = tf_vectorizer.transform(list(test_df['cleaned_text'])).toarray()
x_test_2 = np.array(test_df['DATE']).reshape(-1, 1)
x_test = np.hstack((x_test_1, x_test_2))
```

[50]:
```
sample_str = "this money song is good www"
sample_cleaned = clean_text(sample_str)
sample_vectorized = tf_vectorizer.transform([sample_cleaned]).toarray()
```

```
sample_date = np.array([0]).reshape(-1, 1)
sample_x = np.hstack((sample_vectorized, sample_date))
prediction = best_model_rf.predict(sample_x)
prediction
```

[50]:
```
array([1])
```

[51]:
```
best_params_final = best_model_rf.get_params()
```

[52]:
```
rf = RandomForestClassifier(**best_params_final)
rf.fit(x, y)
```

[52]:
```
RandomForestClassifier(max_depth=100, min_samples_split=7, random_state=42)
```

[53]:
```
y_pred = rf.predict(x_test)
```

[54]:
```
sol = pd.read_csv("./sample_submission.csv")
sol['CLASS'] = y_pred
sol.head()
sol.to_csv("final_submission.csv", index=False)
```