Optimizing N-grams and Embeddings for YouTube Spam Detection

Huy Dang (1007857277), Peize Zhang (1008759862)
Henry Ren (1009428246), Yunshu Zhang (1007761673)

Dataset: Detect Spam YouTube Comment
Team Name: YouTube Spammers
Final Ranking: 25th
Prediction Score: 0.94014

Abstract

YouTube's moderation faces ongoing challenges with spam comments. While machine learning approaches for spam classification are well-studied, the impact of text preprocessing techniques—specifically tokenization and text-embedding methods—remains underexplored. This study examines how different n-gram sizes and frequency-based embedding techniques (Bag of Words and TF-IDF) affect spam classification performance. Three key questions were addressed: determining the optimal n-gram value; identifying the most effective frequency-based embedding technique; and finding the best model configuration for YouTube spam classification. To achieve this goal, various n-gram values and embedding techniques were compared using the Naive Bayes, Random Forest, and Support Vector Machine classifiers. Results revealed that the optimal n-gram values were model-dependent, TF-IDF consistently outperformed Bag of Words, and Random Forest with TF-IDF embedding was the strongest base configuration - however, the optimal n-gram selection depended on specific classification priorities. These findings provide practical guidance for implementing more effective spam detection systems and contribute to the broader understanding of text pre-processing techniques in social media comment moderation.

Keywords: *Spam Classification, Character N-gram, Bag-of-Words, TF-IDF, Naive Bayes, Random Forest, Support Vector Machine*

1 Introduction

Comment sections are essential for interaction between content creators and their communities on platforms such as YouTube; However, these communication tools can also be misused by those with ulterior motives, specifically spammers. Spam comments come in various forms, such as product advertising, misinformation, phishing, and scamming. In moderate cases, they degrade the user experience, and in extreme cases, they can be outright harmful. This study examines various classifiers for spam detection, focusing on tokenization and text-embedding techniques like n-grams, TF-IDF, and Bag-of-Words. Using Naive Bayes, Support Vector Machine, and Random Forest models, this study evaluated how tokenization and text-embedding approaches can optimize YouTube's spam detection capabilities, thereby improving safety on the platform.

1.1 Literature Review

The prevalence of spam comments on YouTube has led to numerous studies evaluating various models for spam detection. A 2018 study found Naive Bayes to be the most accurate among seven models, while K-Nearest Neighbor performed the worst (Aziz, Mohd Foozy, Shamala & Suradi, 2018). Another study, using additional metrics like AUC and F1 score, identified Random Forest as the top-performing model (Xiao & Liang, 2024). Researchers from the Manipal Institute of Technology further examined n-grams, concluding that SVMs with character-level n-gram ($n = 6$) were the most effective (Aiyar & Shetty, 2018). Spam classification has also been studied for other social media platforms. Neutral Networks have been used for Twitter spam detection (Wu, Liu, Zhang, & Xiang, 2017), SVMs with TF-IDF embeddings for Instagram (Haqimi, Rokhman, & Priyanta, 2019), and feature engineering approaches for Facebook (Soiraya, Thanalerdmongkol, & Chantrapornchai, 2012). While these studies demonstrated the effectiveness of various models in different contexts, they failed to highlight the trade-offs between accuracy and computational cost, nor did they investigate further the impact that different tokenization and vectorization techniques have on overall model performance.

1.2 Objective

While previous research had explored various techniques for spam detection, most studies primarily focused on evaluating different models and their accuracy in spam detection, whereas our research places additional emphasis on the domain of tokenization and text embeddings—the "pre-model"

process of representing text data. Specifically, the following research questions are addressed in our paper: (1) *What n-gram representation performs best for detecting YouTube comment spam, based on our evaluation metrics?* (2) *How does TF-IDF compare to Bag of Words (BoW) as a text representation technique?* (3) *What combination of tokenization method, embedding technique, and model (Naive Bayes, Random Forest, or Support Vector Machine) has the best performance?* Ultimately, this research project aims to enhance the understanding of how different pre-model text representation techniques influence the effectiveness of spam detection models and identify optimal configurations for accurately detecting spam in YouTube comments.

2 Methodology

2.1 Tokenization Technique

An n-gram is a sequence of n consecutive tokens extracted from a larger sequence. For example, given a character-gram of size m, we can extract $m-n+1$ substrings of n consecutive characters. This paper focuses on character n-grams, where each character in a string represents a token. Character n-grams are ideal for short, context-rich texts like YouTube comments because they outperform larger units such as word n-grams (Kanaris et al., 2007) and eliminates the need for language-dependent tools, which can be manipulated by spammers. Therefore, this study focuses on tokenizing via character n-grams and exploring the effectiveness of varying the n-values to optimize spam detection.

2.2 Text-embedding Techniques

Text embedding is a process where text is converted into numerical vectors that capture semantic meanings and relationships to enable models to process and learn from the data. This paper will focus on frequency-based text embedding techniques, TF-IDF and Bag-of-Words (BoW), since there is a strong need for simple, interpretable, and computationally efficient methods in spam detection on platforms like YouTube. These frequency-based embedding techniques offer simpler, more transparent representations of text that are easier to implement and understand. These methods rely on the frequency at which words occur, making them well-suited for tasks like spam classification, where the primary focus is on identifying specific patterns and keywords commonly found in spam content.

Bag-of-Words (BoW) is a text representation technique that represents a document by counting the frequency of each "token" (word or term) in the text. The corpus is represented as a sparse matrix, where each document corresponds to a row, and each column corresponds to a token in the vocabulary (Hasan, Matin, Bansal, & Uddin, 2021).

Term Frequency-Inverse Document Frequency (TF-IDF) builds on BoW by incorporating weights that reflect the importance of each token. Specifically, tokens that appear frequently across many documents are deemed less important, as they carry less unique information. Each token is assigned a weight based on the following formula (Hasan et al., 2021):

$$TF - IDF(d, t, N) = TF(t, d) \times IDF(t, D)$$

where

$$TF(t, d) = \frac{\text{Number of Times Token t appear in Document d}}{\text{Total Number of Tokens in Document d}}$$

$$IDF(t, N) = \log\left(\frac{\text{Total Number of Documents in the Corpus N}}{\text{Number of Documents containing Term t}}\right)$$

2.3 Model Architectures

We consider Multinomial Naive Bayes, Random Forest, and Support Vector Machine as potential models because they are the most common models used for classification tasks (Sohana, Rupa, &

Rahman, 2021). Naive Bayes is popular due to its simplicity, especially in text classification tasks (Zhang, 2004). Random Forest and Support Vector Machine, on the other hand, provide robust performance and high prediction accuracy in real-world datasets (Aiyar & Shetty, 2018). Each model brings unique strengths, making them all suitable candidates for our classification problem.

Naive Bayes

Naive Bayes is built upon Bayes' Theorem, which allows us to compute the posterior probability of a class given the observed predictors (James, Witten, Hastie, & Tibshirani, 2021). In a binary classification setting with classes C_1 and C_2, the prior probability of class C_k is denoted as $P(C_k)$ with $k = \{1, 2\}$, and the likelihood of observing predictor x_i given class C_k is $P(x_i|C_k)$. Using Bayes' Theorem, the posterior probability is proportional to:

$$P(C_k \mid x_i) \propto P(C_k)P(x_i \mid C_k)$$

Multinomial Naive Bayes assigns the class which has the highest posterior probability:

$$\hat{y} = \arg\max_{k \in \{1,2\}} (C_k) \prod_{i=1}^{n} P(x_i \mid C_k)$$

Support Vector Machine

Support Vector Machine (SVM) finds the optimal decision boundary (hyperplane) that maximizes the margin between classes (James et al., 2021). The boundary is defined as:

$$w^T x + b = 0$$

where x is the input, while w and b are derived by the optimization problem:

$$\min_{(w,b)} \|w\|_2^2 \quad \text{such that} \quad y_i \left(w^T x + b\right) \geq 1 \quad \forall i \in [n]$$

For non-linearly separable data, SVM uses hinge loss and kernel functions to transform data into higher dimensions, enabling it to model complex relationships (James et al., 2021). This model is effective in high-dimensional spaces and requires tuning of hyperparameters like regularization term C and kernel type to improve performance.

Random Forest

Random Forest (RF) is an ensemble method that builds multiple decision trees, each trained on a bootstrap sample of the data (James et al., 2021). At each tree split, a random subset of predictors is considered, and the final class prediction is determined by a majority vote. By reducing the correlation between trees, RF lowers variance and improves robustness compared to single decision trees. Therefore, increasing the number of trees improves the performance of the model, at the expense of increased computational cost.

2.4 Evaluation Metrics

To evaluate the embeddings and models, the following two metrics were used: F1-Score and Inference Time. We aim to develop a method that makes accurate predictions quickly.

Our model was trained using stratified k-fold cross-validation. This technique divides the dataset into k "folds", while preserving class distribution, iterating through the folds, with every iteration using one fold for validation and the remaining k - 1 folds for training (Kavitha et al., 2023). This method

reduces variability between each test iteration and provides consistent evaluation, thereby improving the reliability of our metric, even for balanced datasets.

F1-Score, defined as the harmonic mean of precision and recall, measures accuracy in spam detection (Taha & Hanbury, 2015). A high F1 score ensures both the effective removal of spam (True Positive) and the preservation of legitimate content (True Negative). It follows the following formula:

$$F1 = 2\left(\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}\right)$$

Inference Time (in seconds) is the time taken for a model to make predictions on a set of new data. Since YouTube has a massive user base, the filtering of spam comments must be fast to avoid delays of unsent comments. Thus, inference time is an important criterion to be considered.

Optimizing one of the F1-score or inference times comes at the cost of worsening the other. A model with a high F1-score but long inference time is computationally costly, whereas a model with a low F1-score but short inference time is ineffective. Therefore, these two metrics should be balanced in a realistically applicable model. To analyze this trade-off, F1-score was plotted against the inverse of Inference Time, which visualized the trade-off for the comparison of the models. This metric is further discussed in Section 4.3.

3 Experimental Setup

3.1 Dataset

The dataset was obtained from the Kaggle competition "Detect spam youtube comment" and contains YouTube comments classified into spam or not spam (Detect spam youtube comment, n.d.) The dataset consists of 1,956 data points in total, divided into a training dataset with 1,370 points and a testing dataset with 586 points. Each data point includes features such as `COMMENT_ID`, `AUTHOR`, `DATE`, and `CONTENT`. The training dataset also includes classification labels (`CLASS`) of the comments for supervised learning tasks. The training dataset maintains a roughly balanced distribution between spam (1) and non-spam (0) classes.

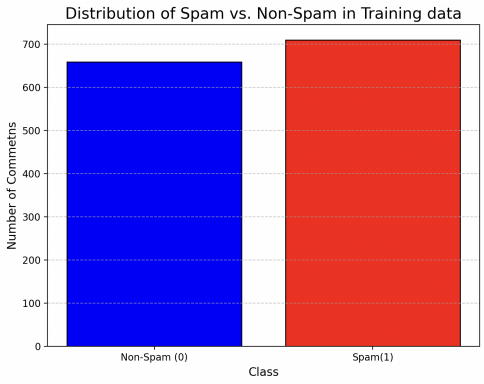


Figure 1: Distribution of Spam vs. Non-Spam in Training Data

3.2 Implementation Details

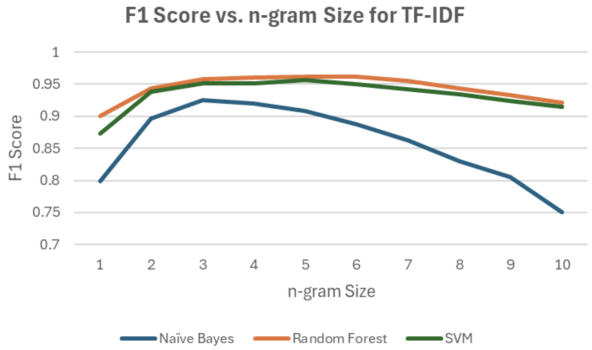
Since the data was provided in a .zip file, the libraries `Pandas` and `Zipfile` were used to extract the training and testing datasets, along with their relevant features (McKinney, 2011; Python Software Foundation, 2024). The library `Scikit-learn` was used to implement the models (Pedregosa et al., 2011). Only the `CONTENT` column was extracted for the training dataset, while both `COMMENT_ID` and `CONTENT` were extracted for the testing dataset. The other columns did not contribute to the feature

representation of the text, and thus, were irrelevant for comparing text embeddings. The models were trained on the training dataset and then used to make predictions on the `CLASS` of the data points in the testing dataset, based on their vectorized `CONTENT`. The hyper-parameters of the three models were kept consistent during the entirety of the modeling procedure to ensure a fair comparison between the different text embedding and tokenization techniques. For RF, we set `n_estimators = 100` and `max_depth = None`, and for SVM, we used a linear kernel. Further, no form of conventional preprocessing techniques such as stemming, lemmatization, or stop-word-removal was applied, as not all the steps were mandatory and the use of pre-compiled stop-list may negatively affect classification performance (Aiyar, 2018).

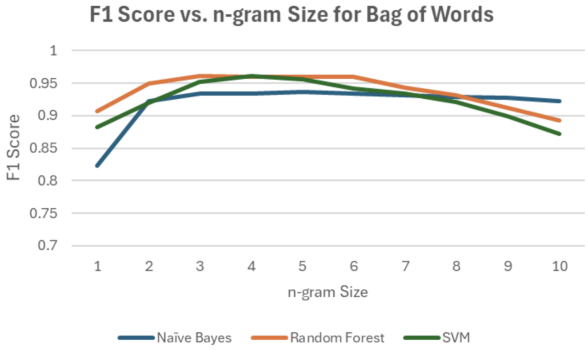
4 Results

4.1 Comparison of N-grams

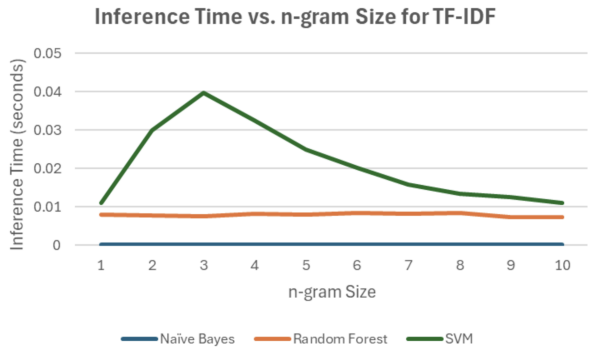
In order to explore the impact that different character n-gram sizes had on the F1-score and inference time of the chosen models, ten experiments were conducted, each using a different n-value from 1 to 10, respectively.



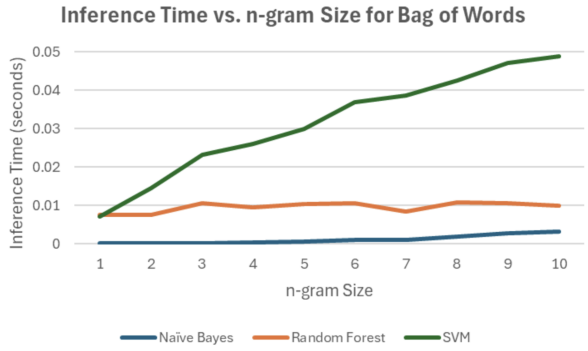
(a) F1 v.s. n using TF-IDF



(b) F1 v.s. n using BoW



(c) Inference Time v.s. n using TF-IDF



(d) Inference Time v.s. n using BoW

Figure 2: Performance comparison of different models with varying n-gram sizes

Figures 2a and 2b reveal a consistent pattern: F1-scores increased as n-gram size increased, reaching a peak at $n = 3$ or 4, but then decreased as n-gram sizes continued to grow. This pattern can be attributed to two key factors: (1) insufficient contextual capture at lower n-values, where tokens represented limited character subsets, and (2) overfitting at higher n-values, where tokens became too specific to training data patterns. Additionally, the expansion of feature space with increasing n-values introduced dimensionality challenges, particularly when training data was limited. The inference time analysis (Figures 2c and 2d) demonstrated distinct behavioral patterns across different classifier-embedding combinations. With TF-IDF, SVM peaked at $n = 3$ and declined sharply, likely

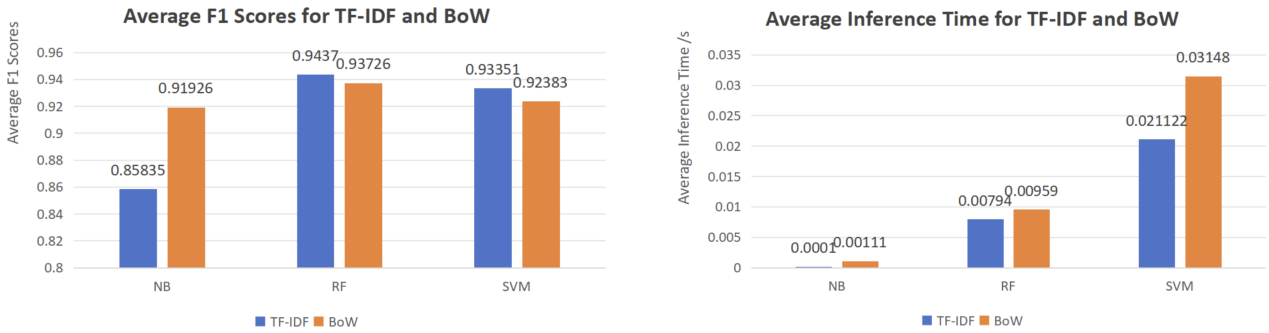
due to the curse of dimensionality. On the other hand, Naive Bayes and Random Forest maintained consistent inference times due to their efficient feature processing mechanisms - independent probability calculations for Naive Bayes and feature sampling in Random Forest's tree structure. For BoW embeddings, RF maintained a stable inference time, whereas both SVM and Naive Bayes showed gradual increases as n increased. This difference could be due to BoW's creation of denser feature spaces compared to TF-IDF, which required more extensive computational resources for SVM. The optimal n-size should strike a balance between F1-score and inference time, and it differed depending on the model:

- Naive Bayes and Random Forest performed optimally at $n = 3$ to 4, effectively balancing F1-scores with consistent inference times;
- SVM with Bag of Words achieved peak performance at $n = 3$, avoiding the sharp inference time increases observed at higher values;
- SVM with TF-IDF showed optimal results at $n = 10$, maintaining peak F1-score performance while minimizing computational overhead.

Overall, the optimal n-value for character n-grams varied depending on the model and text embedding method. For Naive Bayes and RF, $n = 3$ and $n = 4$ were ideal. For SVM, the optimal n-value depended on the embedding: $n = 3$ for BoW and $n = 10$ for TF-IDF.

4.2 Comparison of Embedding Techniques

This section compares BoW and TF-IDF embeddings across all models, focusing on average F1-score and inference time to provide a clear, efficient, and practical analysis. By using averages instead of examining performance at each n-value, the approach saves time, reduces the impact of outliers, and highlights overall trends, making the comparison more robust.



(a) Average F1-Score Comparison

(b) Average Inference Time Comparison

Figure 3: Performance comparison between TF-IDF and BoW across different models

Figure 3a shows average F1-scores for models using TF-IDF and BoW embeddings:

- RF and SVM: TF-IDF achieved higher average F1-scores due to its ability to downweight common tokens and rely on informative ones, enhancing stability on large datasets.
- Naive Bayes: BoW performed better, as it directly counted token occurrence, which provided more accurate prior and likelihood estimations which are crucial for Naive Bayes' Bayes Theorem-based approach.

Figure 3b demonstrates that TF-IDF resulted in shorter average inference times across all models. This is due to its reweighting mechanism, which prunes unimportant terms and thus reduces computational cost—similar to ridge regression's efficiency over OLS (James et al., 2021).

In general, TF-IDF was more computationally efficient and could result in higher average F1-score. However, for Naive Bayes, this advantage was not observed due to the model's reliance on accurate

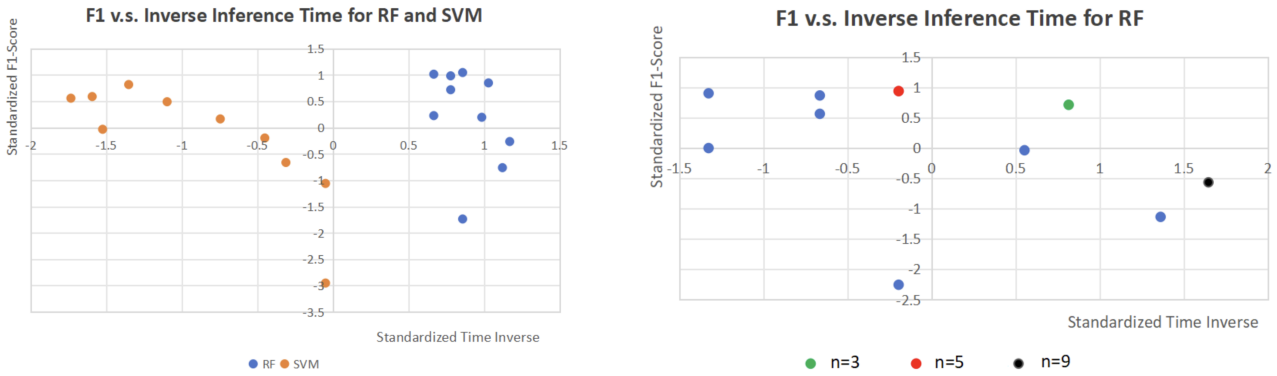
probability estimates for prior and likelihood calculations. BoW directly counted token occurrences, providing a more straightforward and accurate estimation of token probabilities, which is crucial for the Bayes Theorem-based algorithm. In contrast, TF-IDF's reweighting can distort these probabilities, leading to less effective modeling for Naive Bayes.

4.3 General Model Performance Evaluation

This section evaluates model performance across various n-values and embedding techniques. Figure 2 shows that Naive Bayes consistently yielded a relatively poor F1 score despite its low inference time, making it unsuitable for spam detection; As a result, Naive Bayes was excluded from comparison. Similarly, Section 4.2 indicates that TF-IDF generally outperformed BoW, so our comparison was limited to only models using TF-IDF embeddings.

The final comparison evaluated SVM and RF models with TF-IDF under different n-grams. To assess performance, a trade-off analysis was conducted between F1 score and inference time, plotting F1 against the inverse of inference time. Depending on the context, clients [1] may prioritize F1 score, inference time, or a balance of both. Therefore, our recommendations will address these varying priorities rather than focusing on a single metric.

To make the comparison, F1-score and inference time were standardized to fit into a common scale, which made the model comparison simpler. Standardized F1-score and standardized inference-time inverse were then plotted for RF and SVM under different configurations.



(a) RF v.s. SVM using TF-IDF

(b) Random Forest Using TF-IDF

Figure 4: Performance comparison using standardized metrics

Figure 4a compares the performance of RF and SVM with TF-IDF using n-values ranging from 1 to 10. Our observations indicate that all data points corresponding to model RF have positive standardized inverse inference time, i.e. a shorter inference time, while maintaining roughly the same F1 score compared to RF. This suggests that, on average, RF was better than SVM under TF-IDF embeddings.

Figure 4b compares RF models of different n-values. Given a "client" of our model, their model priority was categorized into either (1) High F1 in compensation for High Inference Time; (2) Balance between F1 and Inference Time; and (3) Low F1 in compensation for Low Inference Time. Client (1) should implement RF with TF-IDF and $n = 5$ (red), as it yielded the highest F1-score out of all the data points; Client (2) should implement model RF with TF-IDF and $n = 3$ (green), as it struck a clear balance between F1 and inference time; Client (3) should implement RF with TF-IDF and $n = 9$ (black), as it has the lowest inference time, yet still yields a decent F1 performance.

2

[1] A client refers to an individual or organization seeking to implement our model for their specific objectives.
[2] IT refers to Inference Time.

Client	Model	Text-Embedding	N-gram	Inference Time (s)	F1 (%)
High F1, High IT	RF	TF-IDF	5	0.008	0.962
Balanced	RF	TF-IDF	3	0.0076	0.9576
Low F1, Low IT	RF	TF-IDF	9	0.0073	0.9327

Table 1: *Model Recommendations based on Client Need*

5 Conclusion

5.1 Discussion

Our findings provide nuanced insights into the research questions. First, there is no universally "best" n-gram, as optimal values vary by model. For NB and RF, $n = 3$ and $n = 4$ performed best, while for SVM, the choice depended on embedding: $n = 3$ for BoW and $n = 10$ for TF-IDF. Second, TF-IDF generally outperformed BoW due to its efficient weighting mechanism and comparable or better F1-scores. Lastly, the best overall model was Random Forest with TF-IDF, but n-gram choice depended on priorities: 9-gram for low inference time, 5-gram for higher F1-scores, and 3-gram for a balance. These findings highlight the interplay between text embeddings and tokenization in YouTube spam classification.

5.2 Limitations & Further Research

Ultimately, our study contained several limitations. In sections 4.1 and 4.2, n-grams and text-embeddings were compared using F1-score and inference time individually, but there was no measurement for a trade-off. Thus, including a trade-off metric during the evaluation would have enhanced reliability. In section 4.2, to simplify comparisons, our analysis leveraged averaging F1 and inference times across n-values. However, this approach assumed similar patterns between models, and any datapoint anomalies would have skewed our result and made the metrics less reliable. Moreover, in section 4.3, standardized F1-scores and inversed inference-time were plotted to provide client recommendations, but this analysis relied on intuition rather than a formal metric to make client recommendations, which reduced the rigor of our analysis. Finally, throughout our research, specific model hyperparameters were used for each model, and our model analyses were based on those configurations. Different hyperparameter choices could significantly alter model behavior and results, meaning our conclusions may lack robustness across varying setups.

For future research into the same topic, researchers could could adopt more robust evaluation metrics to compare F1-scores and inference time. Expanding the study to include advanced text embeddings like BERT, Word2Vec, and transformer-based methods may better capture semantics. Researchers could also explore sophisticated models such as XGBoost, LightGBM, LSTMs, and CNNs for improved accuracy. Lastly, a hyperparameter sensitivity analysis is recommended to ensure conclusions are robust and broadly applicable.

6 Kaggle

Kaggle was assumed to be a "client" that prioritizes optimizing F1-score rather than inference time. Therefore, the prediction submitted was generated by model RF with TF-IDF embedding and $n = 5$, as it yielded the highest F1-score. In the end, the final Kaggle score (94%) differed from our cross-validation accuracy (96.2%) by over 2%. This may be because we used stratified k-fold cross-validation to ensure balanced splits, while the testing dataset might not be as balanced. Additionally, reusing the same dataset during cross-validation could have led to slight overfitting, which could have reduced our model's ability to generalize to unseen data.

Acknowledgement

We would like to express our gratitude to the following contributors for their valuable efforts in this project:

- Huy Dang: Topic proposal, model implementations, objective, dataset, tokenization and embedding techniques, implementation details, comparison of n-grams, conclusion.
- Peize Zhang: Trade-off metric proposal, model architecture, evaluation metrics, comparison of n-grams, comparison of embedding techniques, general model performance.
- Henry Ren: Data visualization, manuscript revision.
- Yunshu Zhang: Introduction, literature review, references.

We would also like to thank Professor Xin Bing and TA Fred Haochen Song for providing suggestions and feedback for our research paper.

Access to Full Code

https://github.com/huyxdang/STA314-Project

References

[1] Ahuja, R., Chug, A., Kohli, S., Gupta, S., & Ahuja, P. (2019). The Impact of Features Extraction on the Sentiment Analysis. *Procedia Computer Science*, 152 (Complete), 341–348.

[2] Aiyar, S., & Shetty, N. P. (2018). N-Gram Assisted Youtube Spam Comment Detection. *Procedia Computer Science*, 132, 174–182.

[3] Aziz, A., Mohd Foozy, C. F., Shamala, P., & Suradi, Z. (2018). Youtube spam comment detection using support vector machine and K–nearest neighbor. *Indonesian Journal of Electrical Engineering and Computer Science*, 12, 607–611.

[4] Detect spam youtube comment. (n.d.). *Kaggle*. Retrieved December 8, 2024, from `https://www.kaggle.com/competitions/detect-spam-youtube-comment/leaderboard`

[5] Haqimi, N., Rokhman, N., & Priyanta, S. (2019). Detection Of Spam Comments On Instagram Using Complementary Naïve Bayes. *IJCCS (Indonesian Journal of Computing and Cybernetics Systems*, 13, 263.

[6] Hasan, T., Matin, A., Bansal, J. C., & Uddin, M. S. (2021). Extract Sentiment from Customer Reviews: A Better Approach of TF-IDF and BOW-Based Text Classification Using N-Gram Technique. In *Proceedings of International Joint Conference on Advances in Computational Intelligence* (pp. 231–244). Singapore: Springer Singapore.

[7] James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021). *An Introduction to Statistical Learning: with Applications in R* (Second Edition. ed.). New York, NY: Springer US.

[8] Kanaris, I., Kanaris, K., Houvardas, I., & Stamatatos, E. (2007). WORDS VERSUS CHARACTER N-GRAMS FOR ANTI-SPAM FILTERING. *International Journal on Artificial Intelligence Tools*, 16(06), 1047–1067.

[9] Kavitha, A., Suganya, P., Prakash, A. G., Jeevan, S., & Kumar, R. V. (2023). A Survey on Credit Card Fraud Detection Using Holdout Cross Validation and Stratified K-fold Cross-Validation. *J. Sci*, (1), 1–9.

[10] McKinney, W. (2011). pandas: a Foundational Python Library for Data Analysis and Statistics. *Python High Performance Science Computer*.

[11] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., et al. (2011). Scikit-learn: Machine learning in Python. *The Journal of Machine Learning Research*, 12, 2825–2830.

[12] Python Software Foundation. (2024). zipfile (Version 3.13) [Computer software]. *GitHub*. `https://docs.python.org/3/library/zipfile.html`

[13] Sohana, J. F., Rupa, R. J., & Rahman, M. (2021). Bengali Stop Word Detection Using Different Machine Learning Algorithms. In *Proceedings of International Joint Conference on Advances in Computational Intelligence* (pp. 131–142). Singapore: Springer Singapore.

[14] Soiraya, M., Thanalerdmongkol, S., & Chantrapornchai, C. (2012). Using a Data Mining Approach: Spam Detection on Facebook. *International Journal of Computer Applications*, 58.

[15] Taha, A. A., & Hanbury, A. (2015). Metrics for evaluating 3D medical image segmentation: analysis, selection, and tool. *BMC Medical Imaging*, 15(1), 29–29.

[16] Wu, T., Liu, S., Zhang, J., & Xiang, Y. (2017). Twitter spam detection based on deep learning. *Proceedings of the Australasian Computer Science Week Multiconference*. https://doi.org/10.1145/3014812.3014815

[17] Xiao, A. S., & Liang, Q. (2024). Spam detection for Youtube video comments using machine learning approaches. *Machine Learning with Applications*, 16, 100550.

[18] Zhang, H. (2004). The optimality of naive Bayes. *American Association for Aritifical Intelligence*.

Appendix

Submitted Model

```
import pandas as pd
import time
import zipfile
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, f1_score
from sklearn.ensemble import RandomForestClassifier

# This is the code for Random Forest, TF-IDF embedding, character n-gram with n = 5. It was ch

# Set your n-value here:
n = 5

# Load data
# Load data
z = zipfile.ZipFile('/Users/huydang/Desktop/STA314/Project/youtube_comments.zip') # Change fil
train_data = pd.read_csv(z.open('train.csv'))  # Training data
test_data = pd.read_csv(z.open('test.csv'))  # Test data


# Split training data into features and labels
X = train_data['CONTENT'].values  # Text content
Y = train_data['CLASS'].values  # Labels

# Create TF-IDF representation with character n-gram
```

```
vectorizer = TfidfVectorizer(analyzer='char', ngram_range=(n, n), max_features=5000)
X_tfidf = vectorizer.fit_transform(X)

# Instantiate RF model
model = RandomForestClassifier(
    n_estimators=100,  # Number of trees
    max_depth=None,  # No maximum depth
    n_jobs=-1  # Use all available cores
)

# Fit the model on the entire training dataset
model.fit(X_tfidf, Y)

# Transform the test data
X_test_tfidf = vectorizer.transform(test_data['CONTENT'].values)

# Predict labels for the test dataset
test_predictions = model.predict(X_test_tfidf)

# Create a submission DataFrame
submission_df = pd.DataFrame({
    "COMMENT_ID": test_data['COMMENT_ID'],  # Assuming 'ID' column is present in test data
    "CLASS": test_predictions
})

# Save the submission DataFrame to a CSV file
submission_df.to_csv('submission.csv', index=False)

print("Submission file 'submission.csv' created successfully.")
```

Bag-of-Word Models

Naive Bayes

```
import pandas as pd
import time
import zipfile
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, f1_score
from sklearn.naive_bayes import MultinomialNB
from nltk.stem import PorterStemmer
import nltk

# Set your n-value here:
n = 10

# Load data
z = zipfile.ZipFile('/Users/huydang/Desktop/STA314/Project/youtube_comments.zip')  # Change fi
train_data = pd.read_csv(z.open('train.csv'))  # Training data
test_data = pd.read_csv(z.open('test.csv'))  # Test data

# Extract features and labels
```

```
X_train = train_data['CONTENT'].values  # Text content for training
Y_train = train_data['CLASS'].values  # Labels for training
X_test = test_data['CONTENT'].values  # Text content for testing
test_ids = test_data['COMMENT_ID'].values  # Comment IDs for the test data

# Create Bag of Words representation with character n-grams
vectorizer = CountVectorizer(analyzer='char', ngram_range=(n, n), max_features=5000)
X_train_bow = vectorizer.fit_transform(X_train)
X_test_bow = vectorizer.transform(X_test)

# Stratified K-Fold Cross-Validation
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
fold_metrics = []

# Training and validation loop
for fold, (train_idx, val_idx) in enumerate(skf.split(X_train_bow, Y_train)):
    print(f"Fold {fold + 1}")
    X_train_fold, X_val_fold = X_train_bow[train_idx], X_train_bow[val_idx]
    Y_train_fold, Y_val_fold = Y_train[train_idx], Y_train[val_idx]

    # Measure training time
    start_train_time = time.time()
    model = MultinomialNB()
    model.fit(X_train_fold, Y_train_fold)
    train_time = time.time() - start_train_time

    # Measure inference time
    start_inference_time = time.time()
    Y_val_pred = model.predict(X_val_fold)
    inference_time = time.time() - start_inference_time

    # Calculate metrics
    accuracy = accuracy_score(Y_val_fold, Y_val_pred)
    f1 = f1_score(Y_val_fold, Y_val_pred, average='weighted')

    # Record metrics
    fold_metrics.append({
        "Fold": fold + 1,
        "Accuracy": accuracy,
        "F1-Score": f1,
        "Inference Time (s)": inference_time
    })

# Create a DataFrame to display metrics
metrics_df = pd.DataFrame(fold_metrics)
print(metrics_df)

# Average metrics across folds
average_metrics = metrics_df.drop(columns=["Fold"]).mean(axis=0).to_dict()
print("\nAverage Metrics Across Folds:")
for metric, value in average_metrics.items():
    print(f"{metric}: {value:.4f}")
```

Random Forest

```
import pandas as pd
import time
import psutil
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, f1_score
from sklearn.ensemble import RandomForestClassifier
from nltk.stem import PorterStemmer
import nltk
import zipfile

# Set your n-value here:
n = 10

# Load data
z = zipfile.ZipFile('/Users/huydang/Desktop/STA314/Project/youtube_comments.zip')  # Change fi
train_data = pd.read_csv(z.open('train.csv'))  # Training data
test_data = pd.read_csv(z.open('test.csv'))  # Test data

# Extract features and labels
X_train = train_data['CONTENT'].values  # Text content for training
Y_train = train_data['CLASS'].values  # Labels for training
X_test = test_data['CONTENT'].values  # Text content for testing
test_ids = test_data['COMMENT_ID'].values  # Comment IDs for the test data

# Create Bag of Words representation with character n-grams
vectorizer = CountVectorizer(analyzer='char', ngram_range=(n, n), max_features=5000)
X_train_bow = vectorizer.fit_transform(X_train)
X_test_bow = vectorizer.transform(X_test)

# Stratified K-Fold Cross-Validation
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
fold_metrics = []

# Training and validation loop
for fold, (train_idx, val_idx) in enumerate(skf.split(X_train_bow, Y_train)):
    print(f"Fold {fold + 1}")
    X_train_fold, X_val_fold = X_train_bow[train_idx], X_train_bow[val_idx]
    Y_train_fold, Y_val_fold = Y_train[train_idx], Y_train[val_idx]

    # Measure training time
    start_train_time = time.time()
    model = RandomForestClassifier(
        n_estimators=100,  # Number of trees
        max_depth=None,  # No maximum depth
        random_state=42,
        n_jobs=-1  # Use all available cores
    )
    model.fit(X_train_fold, Y_train_fold)
    train_time = time.time() - start_train_time
```

```
    # Measure inference time
    start_inference_time = time.time()
    Y_val_pred = model.predict(X_val_fold)
    inference_time = time.time() - start_inference_time

    # Calculate metrics
    accuracy = accuracy_score(Y_val_fold, Y_val_pred)
    f1 = f1_score(Y_val_fold, Y_val_pred, average='weighted')

    # Record metrics
    fold_metrics.append({
        "Fold": fold + 1,
        "Accuracy": accuracy,
        "F1-Score": f1,
        "Inference Time (s)": inference_time
    })

# Create a DataFrame to display metrics
metrics_df = pd.DataFrame(fold_metrics)
print(metrics_df)

# Average metrics across folds
average_metrics = metrics_df.drop(columns=["Fold"]).mean(axis=0).to_dict()
print("\nAverage Metrics Across Folds:")
for metric, value in average_metrics.items():
    print(f"{metric}: {value:.4f}")
```

Suppor Vector Machine

```
import pandas as pd
import time
import psutil
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, f1_score
from sklearn.svm import SVC
import zipfile
from nltk.stem import PorterStemmer
import nltk

# Set your n-value here:
n = 10

# Load data
z = zipfile.ZipFile('/Users/huydang/Desktop/STA314/Project/youtube_comments.zip')  # Change fi
train_data = pd.read_csv(z.open('train.csv'))  # Training data
test_data = pd.read_csv(z.open('test.csv'))  # Test data

# Extract features and labels
X_train = train_data['CONTENT'].values  # Text content for training
Y_train = train_data['CLASS'].values  # Labels for training
X_test = test_data['CONTENT'].values  # Text content for testing
test_ids = test_data['COMMENT_ID'].values  # Comment IDs for the test data
```

```
# Create Bag of Words representation with character n-grams
vectorizer = CountVectorizer(analyzer='char', ngram_range=(n, n), max_features=5000)
X_train_bow = vectorizer.fit_transform(X_train)
X_test_bow = vectorizer.transform(X_test)


# Stratified K-Fold Cross-Validation
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
fold_metrics = []

# Training and validation loop
for fold, (train_idx, val_idx) in enumerate(skf.split(X_train_bow, Y_train)):
    print(f"Fold {fold + 1}")
    X_train_fold, X_val_fold = X_train_bow[train_idx], X_train_bow[val_idx]
    Y_train_fold, Y_val_fold = Y_train[train_idx], Y_train[val_idx]

    # Measure training time
    start_train_time = time.time()
    model = SVC(kernel='linear', probability=True, random_state=42)
    model.fit(X_train_fold, Y_train_fold)
    train_time = time.time() - start_train_time

    # Measure inference time
    start_inference_time = time.time()
    Y_val_pred = model.predict(X_val_fold)
    inference_time = time.time() - start_inference_time

    # Calculate metrics
    accuracy = accuracy_score(Y_val_fold, Y_val_pred)
    f1 = f1_score(Y_val_fold, Y_val_pred, average='weighted')

    # Record metrics
    fold_metrics.append({
        "Fold": fold + 1,
        "Accuracy": accuracy,
        "F1-Score": f1,
        "Inference Time (s)": inference_time
    })

# Create a DataFrame to display metrics
metrics_df = pd.DataFrame(fold_metrics)
print(metrics_df)

# Average metrics across folds
average_metrics = metrics_df.drop(columns=["Fold"]).mean(axis=0).to_dict()
print("\nAverage Metrics Across Folds:")
for metric, value in average_metrics.items():
    print(f"{metric}: {value:.4f}")
```

TF-IDF Models

Naive Bayes

```
import pandas as pd
import zipfile
import time
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, f1_score
from sklearn.naive_bayes import MultinomialNB

# Set your n-value here:
n = 10

# Load data
# Load data
z = zipfile.ZipFile('/Users/huydang/Desktop/STA314/Project/youtube_comments.zip') # Change fil
train_data = pd.read_csv(z.open('train.csv'))  # Training data
test_data = pd.read_csv(z.open('test.csv'))  # Test data


# Split training data into features and labels
X = train_data['CONTENT'].values  # Text content
Y = train_data['CLASS'].values  # Labels

# Create TF-IDF representation
vectorizer = TfidfVectorizer(analyzer='char', ngram_range=(n, n), max_features=5000)
X_tfidf = vectorizer.fit_transform(X)

# Stratified 10-Fold Cross-Validation
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
fold_metrics = []

# Training and validation loop
for fold, (train_idx, val_idx) in enumerate(skf.split(X_tfidf, Y)):
    print(f"Fold {fold + 1}")
    X_train_fold, X_val_fold = X_tfidf[train_idx], X_tfidf[val_idx]
    Y_train_fold, Y_val_fold = Y[train_idx], Y[val_idx]

    # Measure training time
    start_train_time = time.time()
    model = MultinomialNB()
    model.fit(X_train_fold, Y_train_fold)
    train_time = time.time() - start_train_time

    # Measure inference time
    start_inference_time = time.time()
    Y_val_pred = model.predict(X_val_fold)
    inference_time = time.time() - start_inference_time

    # Calculate metrics
    accuracy = accuracy_score(Y_val_fold, Y_val_pred)
```

```
    f1 = f1_score(Y_val_fold, Y_val_pred, average='weighted')

    # Record metrics
    fold_metrics.append({
        "Fold": fold + 1,
        "Accuracy": accuracy,
        "F1-Score": f1,
        "Inference Time (s)": inference_time
    })

# Create a DataFrame to display metrics
metrics_df = pd.DataFrame(fold_metrics)
print(metrics_df)

# Average metrics across folds
average_metrics = metrics_df.drop(columns=["Fold"]).mean(axis=0).to_dict()
print("\nAverage Metrics Across Folds:")
for metric, value in average_metrics.items():
    print(f"{metric}: {value:.4f}")
```

Random Forest

```
import pandas as pd
import time
import zipfile
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, f1_score
from sklearn.ensemble import RandomForestClassifier

# Set your n-value here:
n = 10

# Load data
# Load data
z = zipfile.ZipFile('/Users/huydang/Desktop/STA314/Project/youtube_comments.zip') # Change fil
train_data = pd.read_csv(z.open('train.csv'))  # Training data
test_data = pd.read_csv(z.open('test.csv'))  # Test data


# Split training data into features and labels
X = train_data['CONTENT'].values  # Text content
Y = train_data['CLASS'].values  # Labels

# Create TF-IDF representation
vectorizer = TfidfVectorizer(analyzer='char', ngram_range=(n, n), max_features=5000)
X_tfidf = vectorizer.fit_transform(X)

# Stratified K-Fold Cross-Validation
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
fold_metrics = []

# Training and validation loop
```

```
for fold, (train_idx, val_idx) in enumerate(skf.split(X_tfidf, Y)):
    print(f"Fold {fold + 1}")
    X_train_fold, X_val_fold = X_tfidf[train_idx], X_tfidf[val_idx]
    Y_train_fold, Y_val_fold = Y[train_idx], Y[val_idx]

    # Measure training time
    start_train_time = time.time()
    model = RandomForestClassifier(
        n_estimators=100,  # Number of trees
        max_depth=None,  # No maximum depth
        random_state=42,
        n_jobs=-1  # Use all available cores
    )
    model.fit(X_train_fold, Y_train_fold)
    train_time = time.time() - start_train_time

    # Measure inference time
    start_inference_time = time.time()
    Y_val_pred = model.predict(X_val_fold)
    inference_time = time.time() - start_inference_time

    # Calculate metrics
    accuracy = accuracy_score(Y_val_fold, Y_val_pred)
    f1 = f1_score(Y_val_fold, Y_val_pred, average='weighted')

    # Record metrics
    fold_metrics.append({
        "Fold": fold + 1,
        "Accuracy": accuracy,
        "F1-Score": f1,
        "Inference Time (s)": inference_time
    })

# Create a DataFrame to display metrics
metrics_df = pd.DataFrame(fold_metrics)
print(metrics_df)

# Average metrics across folds
average_metrics = metrics_df.drop(columns=["Fold"]).mean(axis=0).to_dict()
print("\nAverage Metrics Across Folds:")
for metric, value in average_metrics.items():
    print(f"{metric}: {value:.4f}")
```

Support Vector Machine

```
import pandas as pd
import time
import zipfile
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score, f1_score
from sklearn.svm import SVC
```

```
# Set your n-value here:
n = 10

# Load data
z = zipfile.ZipFile('/Users/huydang/Desktop/STA314/Project/youtube_comments.zip') # Change fil
train_data = pd.read_csv(z.open('train.csv'))  # Training data
test_data = pd.read_csv(z.open('test.csv'))  # Test data

# Extract features and labels
X_train = train_data['CONTENT'].values  # Text content for training
Y_train = train_data['CLASS'].values  # Labels for training
X_test = test_data['CONTENT'].values  # Text content for testing
test_ids = test_data['COMMENT_ID'].values  # Comment IDs for the test data

# Create TF-IDF representation
vectorizer = TfidfVectorizer(analyzer='char', ngram_range=(n, n), max_features=5000)
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)

# Stratified K-Fold Cross-Validation
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
fold_metrics = []

# Training and validation loop
for fold, (train_idx, val_idx) in enumerate(skf.split(X_train_tfidf, Y_train)):
    print(f"Fold {fold + 1}")
    X_train_fold, X_val_fold = X_train_tfidf[train_idx], X_train_tfidf[val_idx]
    Y_train_fold, Y_val_fold = Y_train[train_idx], Y_train[val_idx]

    # Measure training time
    start_train_time = time.time()
    model = SVC(kernel='linear', probability=True, random_state=42)
    model.fit(X_train_fold, Y_train_fold)
    train_time = time.time() - start_train_time

    # Measure inference time
    start_inference_time = time.time()
    Y_val_pred = model.predict(X_val_fold)
    inference_time = time.time() - start_inference_time

    # Calculate metrics
    accuracy = accuracy_score(Y_val_fold, Y_val_pred)
    f1 = f1_score(Y_val_fold, Y_val_pred, average='weighted')

    # Record metrics
    fold_metrics.append({
        "Fold": fold + 1,
        "Accuracy": accuracy,
        "F1-Score": f1,
        "Inference Time (s)": inference_time
    })
```

```
# Create a DataFrame to display metrics
metrics_df = pd.DataFrame(fold_metrics)
print(metrics_df)

# Average metrics across folds
average_metrics = metrics_df.drop(columns=["Fold"]).mean(axis=0).to_dict()
print("\nAverage Metrics Across Folds:")
for metric, value in average_metrics.items():
    print(f"{metric}: {value:.4f}")
```