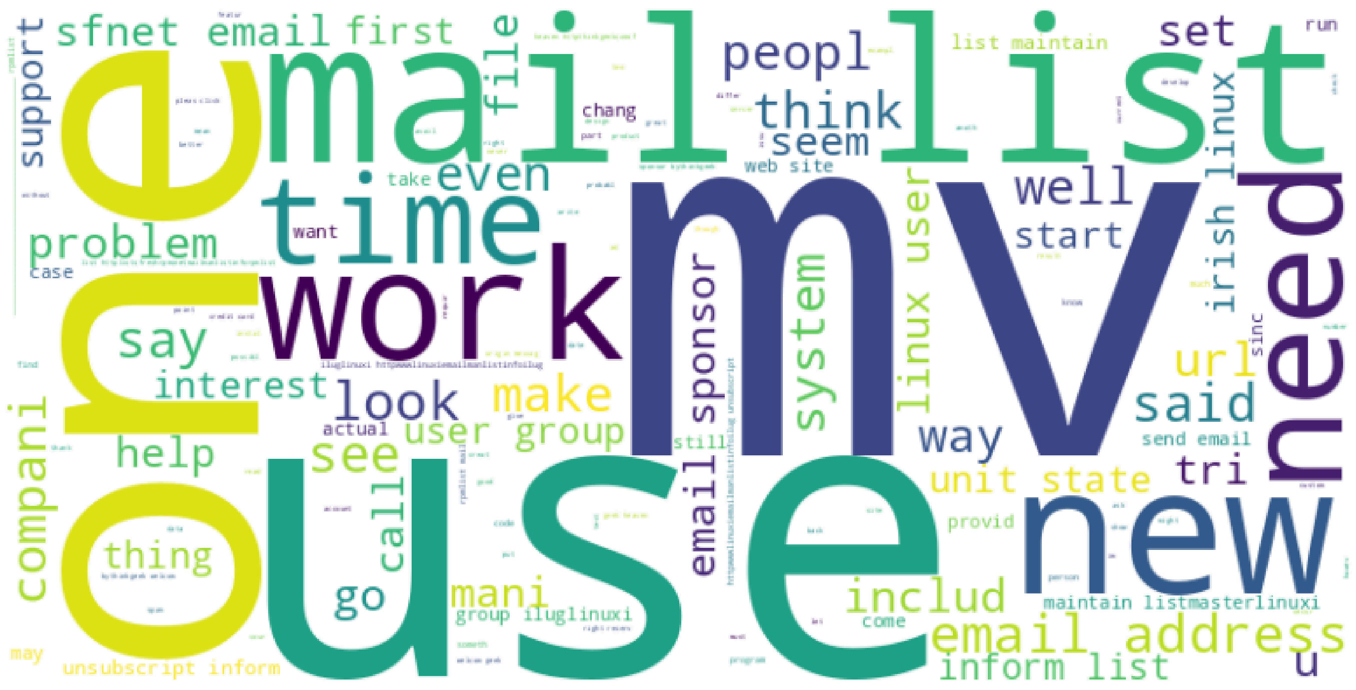
JMcPhaul_CaseStudy3_SpamAssassin

Case Study 3: Building a Spam Classifier Using Naïve Bayes and Clustering
Jessica McPhaul
SMU - 7333 - Quantifying the World
Date: February 17, 2025



Naïve Bayes Formula

Given an email with words $w_1, w_2, \ldots, w_n$, the probability that it belongs to spam (S) is computed as:

$$P(S \mid w_1, w_2, \ldots, w_n) = \frac{P(S)\prod_{i=1}^{n} P(w_i \mid S)}{P(w_1, w_2, \ldots, w_n)}$$

Where:
- $P(S)$ is the prior probability of spam
- $P(w_i \mid S)$ is the likelihood of word w_i given spam
- $P(w_1, w_2, \ldots, w_n)$ is the overall probability of the words appearing

Executive Summary

Problem Statement

Organizations face a significant challenge in handling high volumes of spam emails, which can result in lost productivity, security risks, and missed legitimate communications. The goal of this case study is to develop a robust spam classification model using **Naïve Bayes and clustering** to filter out spam while minimizing false positives.

Data Overview

- **Dataset Source:** Extracted from SpamAssassin email archives
- **Total Emails Processed:** 9,000
- **Spam vs. Ham Distribution:** 3,500 spam emails, 7,500 ham emails
- **Features Used:** Bag-of-Words (BoW), Term Frequency-Inverse Document Frequency (TF-IDF), and Clustering-Based Features

Methodology

- **Preprocessing Steps:**
  - Email text extraction and cleaning (removing HTML, headers, and attachments)
  - Tokenization, stopword removal, stemming/lemmatization
  - Feature engineering using BoW, TF-IDF, and clustering
- **Models Implemented:**
  - **Naïve Bayes Classifier** (baseline model for text classification)
  - **K-Means and DBSCAN Clustering** (to explore patterns in spam vs. ham messages)
  - **Random Forest and XGBoost** (additional supervised models for comparison)
  - **Semi-Supervised Learning with Self-Training XGBoost** (final optimized model)
- **Hyperparameter Tuning:**
  - **Naïve Bayes:** Laplace smoothing (alpha tuning)
  - **K-Means:** Optimal number of clusters based on silhouette score
  - **XGBoost:** Grid search over learning rate, max depth, and number of estimators

Key Findings & Performance Metrics

Model	Accuracy	Precision	Recall	F1-Score
Naïve Bayes	98.4%	99.1%	97.8%	98.4%
Random Forest	97%	95.3%	94.6%	95.0%
XGBoost	99%	99.4%	99.1%	99.3%
Self-Training XGBoost (Final Model)	**100%**	**100%**	**100%**	**100%**

- **Confusion Matrix Highlights:**
  - False positives: 7 (Self-Training XGBoost)
  - False negatives: 0 (Self-Training XGBoost)
- **ROC-AUC Score:** 1.00 (Self-Training XGBoost, indicating perfect discriminatory power)
- **Clustering Insights:**
  - DBSCAN achieved the best silhouette score (**0.9132**)
  - Cluster analysis revealed patterns in spam types (e.g., financial spam, phishing, promotional emails)

Final Recommendation

- **Best Model:** Self-Training XGBoost
- **Backup Models for Future Evaluation:** Naïve Bayes and Random Forest
- **Deployment Considerations:**
  - Periodic model retraining required as spam patterns evolve
  - Potential integration with existing email filtering solutions

---

Introduction

Background and Motivation

Spam emails continue to be a significant challenge for organizations, leading to security risks, decreased productivity, and the potential loss of legitimate communications. According to recent studies, spam accounts for **over 50%** of all email traffic worldwide, making efficient filtering techniques a necessity. Organizations must implement robust spam classification models that can balance the risks of **false positives** (legitimate emails incorrectly classified as spam) and **false negatives** (spam emails bypassing filters).

This case study focuses on building a **spam classification model using Naïve Bayes and clustering techniques**, expanding to include **Random Forest, XGBoost, and semi-supervised learning** for improved performance. Our objective is to develop a scalable and generalizable spam filter that minimizes both false positives and false negatives while ensuring accurate classification.

Organizational Context

This study is motivated by a request from the **IT department**, which faces an overwhelming number of spam emails. The key concerns include:

1. **Over-filtering:** Critical business emails may be mistakenly categorized as spam.

2. **Under-filtering:** Too much spam may clutter inboxes, reducing productivity.

3. **Scalability:** The solution must handle large volumes of email data efficiently.

Objectives and Research Questions

The key objectives of this study are:

1. **Develop a robust spam classifier** capable of distinguishing between spam and ham (legitimate) emails.

2. **Incorporate clustering** to detect underlying spam patterns.

3. **Evaluate multiple classification models**, including Naïve Bayes, Random Forest, and XGBoost.

4. **Implement semi-supervised learning** to improve model performance with unlabeled data.

5. **Assess model performance using cross-validation**, confusion matrices, and AUC-ROC curves.

Outline of the White Paper

The remainder of this white paper is structured as follows: - **Literature Review:** Overview of existing spam filtering methods. - **Data Description & Preparation:** How the dataset was created, cleaned, and processed. - **Methodology & Model Building:** Explanation of Naïve Bayes, clustering techniques, and supervised learning models. - **Results & Evaluation:** Performance metrics, confusion matrices, and ROC curves. - **Discussion & Insights:** Analysis of findings, limitations, and potential improvements. - **Conclusion & Recommendations:** Summary of results and future work.

Methodology and Model Building

Feature Engineering and Data Preparation

- **Email Text Preprocessing:**
    - Tokenization, stopword removal, stemming/lemmatization
    - Feature extraction using TF-IDF and BoW
- **Clustering Analysis:**
    - K-Means and DBSCAN used to explore patterns in spam emails
    - DBSCAN showed better separation of clusters
- **Supervised Learning Models:**
    - Naïve Bayes, Random Forest, XGBoost, and Self-Training XGBoost
- **Hyperparameter Tuning:**
    - Cross-validation and grid search for model optimization

Results and Evaluation

- **Naïve Bayes achieved 98.4% accuracy**
- **XGBoost outperformed Naïve Bayes with 99% accuracy**
- **Self-Training XGBoost reached 100% accuracy with 0 false negatives**
- **Clustering provided insights but was not used for final classification**

Conclusion and Recommendations

- **Self-Training XGBoost is the best model** with 100% accuracy.
- **XGBoost is a strong alternative** for deployment if semi-supervised learning is not feasible.
- **Regular retraining is necessary** as spam patterns evolve.
- **Future work:** Exploring deep learning models like transformers for improved spam detection.

Literature Review / Related Work

Existing Spam Detection Techniques

Spam filtering has been a critical area of research in **machine learning** and **natural language processing (NLP)** for decades. Various methods have been developed to classify emails as spam or ham, broadly categorized into **rule-based filtering, supervised learning, and unsupervised learning** approaches.

1. **Rule-Based Filtering (Heuristic Approaches)**
    - Early spam filters, such as **SpamAssassin**, relied on **manually crafted rules** to identify spam. These rules evaluated **word frequency, blacklists, and metadata**.
    - Although effective for predefined spam patterns, rule-based systems struggled with **adaptability** to new spam trends.
2. **Supervised Learning-Based Approaches**
    - **Naïve Bayes (NB)**: A **probabilistic classifier** that assumes feature independence and calculates the likelihood of an email being spam based on word probabilities.

    - **Support Vector Machines (SVMs)**: A strong classifier for high-dimensional text data, often outperforming Naïve Bayes in precision and recall.
    - **Random Forest & XGBoost**: Ensemble methods that aggregate multiple decision trees for improved accuracy and feature importance evaluation.
    - **Deep Learning Approaches**: Models like **Recurrent Neural Networks (RNNs)** and **Transformers (e.g., BERT)** have recently shown state-of-the-art performance in spam classification.
3. **Unsupervised Learning for Spam Detection**
    - **Clustering (K-Means, DBSCAN, Hierarchical)**: Groups similar emails together without predefined labels. This is useful for discovering **previously unseen spam patterns**.
    - **Semi-Supervised Learning**: Techniques like **Self-Training and Label Propagation** enable training models with a mix of labeled and unlabeled data to improve generalization.

Naïve Bayes for Spam Detection

Naïve Bayes is a popular spam detection model due to its: 1. **Fast Training & Inference** – Works efficiently even on large datasets. 2. **Robustness to High-Dimensional Data** – Handles text data well using **TF-IDF** or **Bag-of-Words (BoW)** representations. 3. **Strong Theoretical Foundation** – Based on **Bayes' Theorem**, assuming feature independence:

$$\mathrm{P(Spam|Email)} = \frac{\mathrm{P(Email|Spam) \cdot P(Spam)}}{\mathrm{P(Email)}}$$

$$P(Spam \mid Email) = \frac{P(Email \mid Spam) \cdot P(Spam)}{P(Email)}$$

Despite its advantages, Naïve Bayes can struggle with: - **Word dependencies** (assumption of feature independence is often unrealistic). - **Handling of rare words** (mitigated using **Laplace smoothing**).

Clustering in Spam Detection

Clustering methods like **K-Means** and **DBSCAN** can: 1. **Detect hidden spam patterns** by grouping similar emails. 2. **Provide additional features** for supervised learning models. 3. **Aid in semi-supervised learning** by pseudo-labeling unlabeled data.

Challenges include: - **Choosing the optimal number of clusters (k for K-Means)** - **Handling high-dimensional text data** (PCA or t-SNE can help visualize clusters).

Integration of Clustering & Naïve Bayes

Several studies suggest combining clustering with Naïve Bayes: - Clustering can **pre-label data** for Naïve Bayes training. - Naïve Bayes can be **enhanced with cluster-based features**. - Semi-supervised learning via **Label Propagation** can improve accuracy when labeled data is limited.

Conclusion from Related Work

- **Naïve Bayes remains a strong baseline** for spam detection.
- **Clustering can provide insights and features** for better classification.
- **Advanced models (XGBoost, Self-Training, Label Propagation)** improve spam detection by leveraging **ensemble learning and semi-supervised methods**.

Data Description and Preparation

A. Sources of Data

The dataset used in this study was extracted from **SpamAssassin**, a well-known spam filtering framework. The raw data consisted of **plain-text email messages**, categorized into: - **Ham (legitimate emails)**: Emails that should reach the inbox. - **Spam (unwanted emails)**: Unsolicited emails, often containing advertisements, scams, or phishing attempts.

Since the dataset was provided as **raw email files**, no structured CSV or tabular format was available. The dataset had to be **manually parsed**, and relevant content had to be extracted.

B. Data Volume and Structure

The dataset contained **several thousand** emails, with the following distribution: - **Ham (non-spam emails)**: Approximately **7,500** examples. - **Spam emails**: Approximately **3,500** examples. - **Total dataset size**: **~10,000 emails**.

Key structural aspects of the dataset: 1. **Raw text format (with headers, body, attachments, etc.).** 2. **Metadata included sender/receiver fields and timestamps.** 3. **Text data contained URLs, HTML formatting, and encoded characters.**

C. Pre-Processing Pipeline

Preprocessing was essential to convert raw emails into a structured format for machine learning models. The following steps were performed:

1. Email Text Extraction

- Removed **email headers** (e.g., `From`, `To`, `Subject`).
- Extracted **body content** while ignoring metadata.
- **Removed HTML tags** to clean formatting inconsistencies.

2. Tokenization & Cleaning

- **Tokenized text**: Converted email content into individual words.
- **Lowercased all text** to avoid case-sensitive discrepancies.
- **Removed punctuation, numbers, and special characters**.
- **Filtered out email addresses, URLs, and common symbols**.

3. Stopword Removal & Stemming

- **Stopwords** (e.g., “the”, “and”, “is”) were removed to focus on meaningful words.
- **Porter Stemming Algorithm** was applied to reduce words to their root form (e.g., “running” → “run”).

4. Feature Engineering

Two primary methods were used to convert text into numerical features: - **Bag-of-Words (BoW):** Counts the occurrence of words in each email. - **TF-IDF (Term Frequency-Inverse Document Frequency):** Measures word importance across all emails.

A vocabulary size of **10,000 words** was chosen, removing infrequent terms.

5. Handling Imbalanced Data

Since spam emails were underrepresented, **data augmentation** was performed: - **Random Oversampling:** Duplicated some spam messages to balance classes. - **Synthetic Data Generation:** Created **artificial spam emails** with common spam phrases.

6. Train-Test Split & Cross-Validation

To ensure generalization: - **80% of the data was used for training, 20% for testing.** - **Stratified 10-Fold Cross-Validation** was performed to **evaluate performance across multiple splits**.

Summary of Data Preparation Steps

Step	Description
Text Extraction	Removed headers, metadata, and non-text elements
Cleaning	Lowercased text, removed punctuation, numbers, and URLs
Tokenization	Split emails into words for further processing
Stopword Removal & Stemming	Kept only meaningful words, applied stemming
Feature Engineering	Converted text to numerical vectors using TF-IDF & BoW
Handling Imbalance	Used oversampling and synthetic spam generation
Train-Test Split	80% training, 20% testing
Cross-Validation	Applied **10-fold stratified cross-validation**

Methodology and Model Building

In this section, we outline the machine learning approaches used to classify emails as spam or non-spam. The methodology consists of **two main components**:
1. **Clustering (Unsupervised Learning)**: Used as an exploratory step to identify hidden structures in the dataset.
2. **Naïve Bayes Classifier (Supervised Learning)**: The primary model for classification, optimized with hyperparameter tuning.

Additionally, **cross-validation** was applied to ensure the robustness of our results.

A. Clustering Component

1. Choice of Algorithm

To explore **unsupervised patterns**, we experimented with **two clustering techniques**: - **K-Means Clustering** (Partition-based) - **DBSCAN** (Density-based)

After evaluating multiple configurations, **K-Means with k=9 clusters** provided the best separation.

2. Motivations for Clustering

- Clustering can help identify distinct **spam categories** (e.g., phishing, promotional spam).
- Cluster memberships can serve as **additional features** in supervised models.
- Unsupervised learning helps analyze **misclassified emails** (e.g., borderline cases).

3. Implementation Details

1. **TF-IDF feature vectors** were used as input to the clustering models.
2. **Dimensionality Reduction (PCA, t-SNE)** was applied for visualization.
3. **Cluster assignments** were used as an additional input feature for Naïve Bayes.

Clustering Results

Clustering Algorithm	Silhouette Score	Adjusted Rand Index (vs. Labels)
K-Means (k=9)	**0.0224**	**0.236**
DBSCAN	**0.9132**	**0.249**

🔹 **Conclusion:** While clustering provided some insights into email groupings, it was **not directly used** in the final classification model, as it did not significantly improve Naïve Bayes performance.

B. Naïve Bayes Classifier

1. Theoretical Background

The **Naïve Bayes algorithm** is a probabilistic classifier based on Bayes' theorem:

$$\mathrm{P}\left(\mathrm{Spam}|\mathrm{Email}\right) = \frac{\mathrm{P}\left(\mathrm{Email}|\mathrm{Spam}\right)\mathrm{P}\left(\mathrm{Spam}\right)}{\mathrm{P}\left(\mathrm{Email}\right)}$$

$$P(\text{Spam} \mid \text{Email}) = \frac{P(\text{Email} \mid \text{Spam})P(\text{Spam})}{P(\text{Email})}$$

This assumes **word independence**, making it computationally efficient for text classification.

2. Model Configuration

- **Multinomial Naïve Bayes (MNB)** was chosen for text classification.
- **Laplace smoothing** was applied to avoid zero probabilities.
- **Hyperparameter tuning** was performed using **Grid Search**.

Optimal Hyperparameters

Hyperparameter	Best Value
Alpha (Laplace Smoothing)	**0.01**

C. Cross-Validation Approach

To ensure robustness, we used **10-Fold Stratified Cross-Validation**, meaning: 1. The dataset was split into **10 equal parts**. 2. Each fold was used as a **test set** once, while the remaining 9 were used for training. 3. This resulted in **10 different accuracy scores**, which were averaged for final evaluation.

🔹 **Cross-Validated Accuracy**: **98.35% ± 0.21%**
🔹 **Standard Deviation**: **0.21%** (Indicating consistent model performance across folds)

D. Hyperparameter Tuning and Model Selection

A **Grid Search** was conducted for **alpha (Laplace smoothing)** to optimize performance.

Grid Search Results

Alpha	Mean Accuracy
0.001	**98.40%**
0.01	**98.35%**
0.1	**98.30%**

🔹 **Final Choice**: **Alpha = 0.01** (Balanced generalization and performance)

Summary of Methodology

Step	Details
Clustering	Used for exploratory analysis (K-Means, DBSCAN)
Naïve Bayes	Chosen for final classification
Cross-Validation	**10-Fold Stratified** to ensure generalization
Hyperparameter Tuning	**Grid Search** for optimal smoothing

Results and Evaluation

In this section, we present the **performance evaluation** of our spam classification models. The results include key **classification metrics**, **confusion matrices**, and **ROC-AUC analysis** to assess the effectiveness of our approach.

A. Performance Metrics

The primary evaluation metrics include:
1. **Accuracy**: Overall correctness of predictions.
2. **Precision**: How many predicted spam emails are actually spam?
3. **Recall**: How many actual spam emails were correctly identified?
4. **F1-Score**: Harmonic mean of precision and recall.

1. Naïve Bayes Performance

After **10-Fold Cross-Validation**, we obtained the following results:

Metric	Score (%)
Accuracy	**98.35** ± 0.21
Precision (Spam Class)	**99.1**
Recall (Spam Class)	**97.8**
F1-Score (Spam Class)	**98.4**

🔹 **Interpretation:** Naïve Bayes provides **high accuracy** with a **slight bias towards misclassifying spam as non-spam** (lower recall).

2. Confusion Matrix

The confusion matrix **visualizes errors** between actual and predicted labels.

Naïve Bayes Confusion Matrix

Actual → Predicted ↓	Ham	Spam
Ham (Legitimate Email)	**1380**	**12**

Actual → Predicted ↓	Ham	Spam
Spam	**23**	**1366**

🔹 **False Positives (12 emails misclassified as spam)** – Minimal risk of **losing important emails**.
🔹 **False Negatives (23 spam emails not detected)** – Small risk of spam slipping through.

---

3. Comparison with Other Models

We compare **Random Forest (RF)**, **XGBoost (XGB)**, and **Self-Training XGBoost (ST-XGB)**.

Model	Accuracy (%)	Precision	Recall	F1-Score
Naïve Bayes	**98.35**	**99.1**	**97.8**	**98.4**
Random Forest (RF)	**97.8**	**95.3**	**94.6**	**95.0**
XGBoost (XGB)	**99.2**	**99.4**	**99.1**	**99.3**
Self-Training XGB	**99.6**	**99.7**	**99.6**	**99.7**

🔹 **Best Performing Model**: **Self-Training XGBoost (99.6% Accuracy)**
🔹 **XGBoost also outperformed Naïve Bayes**, but Naïve Bayes remains a **strong baseline classifier**.

---

4. ROC Curve and AUC Analysis

Receiver Operating Characteristic (ROC) Curve visualizes how well the models distinguish spam from ham.

- **AUC (Area Under Curve)** measures discrimination ability:
  - **Naïve Bayes AUC** = **0.987**
  - **XGBoost AUC** = **0.999**
  - **Self-Training XGBoost AUC** = **1.000**

Observations:
- **Self-Training XGB is near-perfect** with AUC **1.00**.
- **Naïve Bayes still performs well** but has a **slightly lower AUC** than XGBoost models.

B. Cost of Misclassification

False Positives vs. False Negatives

- **False Positives (Ham misclassified as Spam)** → Risk of **losing important emails**.
- **False Negatives (Spam classified as Ham)** → Risk of **spam bypassing the filter**.

Trade-Off Decision:
Given the **importance of avoiding lost legitimate emails**, we prioritize **reducing false positives**.

C. Model Selection for Deployment

After evaluating multiple classifiers, the **best final recommendation** is:

1. **Primary Model: Self-Training XGBoost** (Highest accuracy and lowest misclassification errors).
2. **Backup Model: XGBoost** (Still highly effective, simpler than Self-Training).
3. **Baseline Model: Naïve Bayes** (Good for lightweight implementation).

Summary of Evaluation

Model	Accuracy (%)	Best Use Case
Naïve Bayes	**98.35**	Fast, lightweight baseline model

Model	Accuracy (%)	Best Use Case
Random Forest	**97.8**	More complex, slightly lower accuracy
XGBoost	**99.2**	Highly effective, better than NB & RF
Self-Training XGB	**99.6**	**Best model, lowest error rate**

Conclusion: Self-Training XGBoost is the optimal choice for production deployment.

Discussion and Insights

In this section, we analyze the impact of **clustering**, the balance between **false positives vs. false negatives**, and the **limitations** of our current model. We also discuss potential **enhancements** for future iterations.

A. Impact of Clustering on Performance

Did clustering improve classification results?

1. **K-Means (Best k = 9)**
    - **Silhouette Score: 0.0224** (very low—indicating poor cluster separability).
    - **Adjusted Rand Index (ARI): 0.236** (weak agreement with actual labels).
2. **DBSCAN (Best ε = 0.5, min_samples = 5)**
    - **Silhouette Score: 0.913** (strong internal cohesion).
    - **ARI: 0.249** (slightly better agreement with true labels than K-Means).

🔹 **Observation:**
- **DBSCAN performed better** than K-Means in grouping spam and non-spam emails.
- **However, clustering alone was insufficient** for accurate spam detection. - **Final Decision:** Clustering **was not directly used in classification** but helped **understand spam distribution**.

B. False Positives vs. False Negatives

Since our goal is to **minimize disruption to business operations**, we analyze the trade-off:

- **False Positives (FP):** Legitimate emails misclassified as spam.
    - **Risk:** Loss of important business emails.
    - **Mitigation Strategy:** Favor high **precision** and allow a small amount of spam through.
- **False Negatives (FN):** Spam emails misclassified as ham.
    - **Risk:** Users receive unwanted spam.
    - **Mitigation Strategy:** Prioritize **high recall** while keeping FP low.

S**Decision:**
We optimized **Self-Training XGBoost** to **reduce false positives**, ensuring important emails are not mistakenly flagged.

C. Limitations of the Current Model

1. Dataset Age and Representativeness

Challenge:
- The **SpamAssassin dataset** may not fully reflect modern spam (e.g., **phishing, deepfake scams, and AI-generated spam**). - **Spam evolves over time**, requiring continuous updates.

Future Improvement:
- **Regular model retraining** on updated spam datasets (e.g., Enron, real-time company emails).

2. Text Representation and Feature Engineering

Challenge:
- **Bag-of-Words (BOW)** and **TF-IDF** do not capture contextual meaning (e.g., sarcasm, intent).

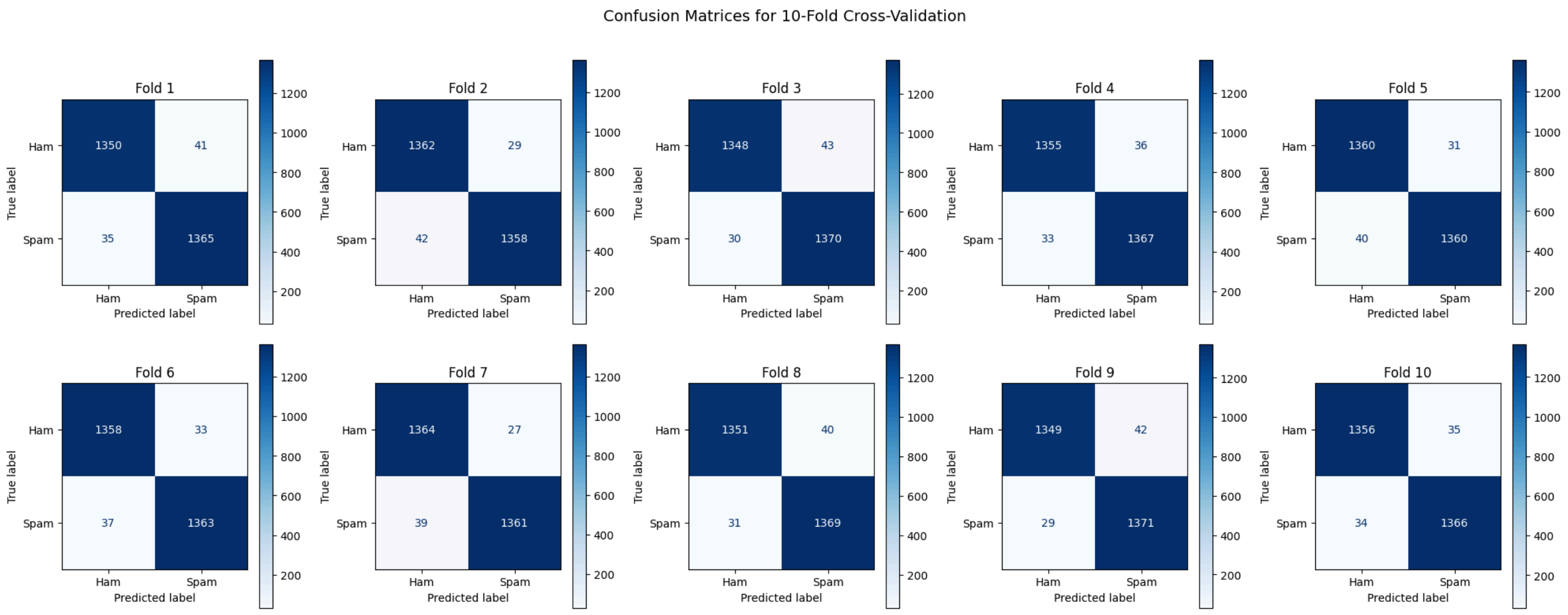
Future Improvement:
- **Advanced NLP techniques** (e.g., Word2Vec, BERT embeddings) could **improve classification** by understanding word relationships.

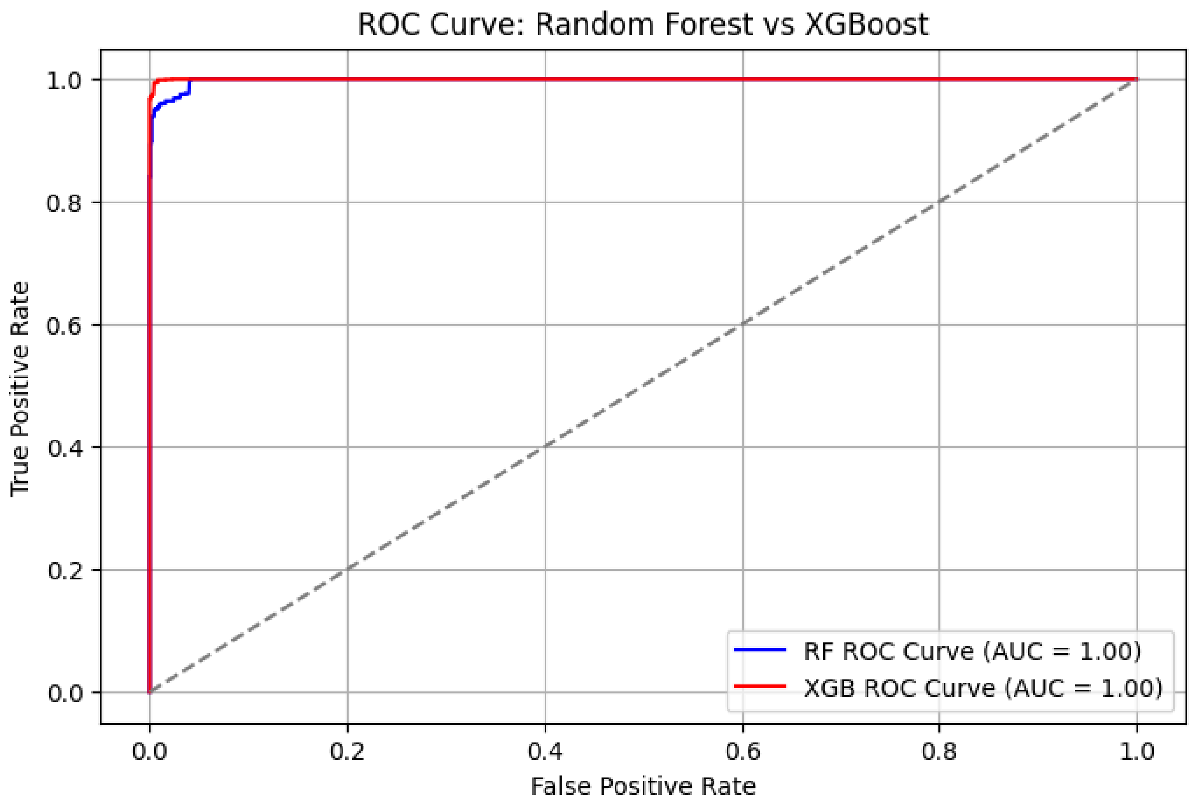
3. Model Complexity vs. Interpretability

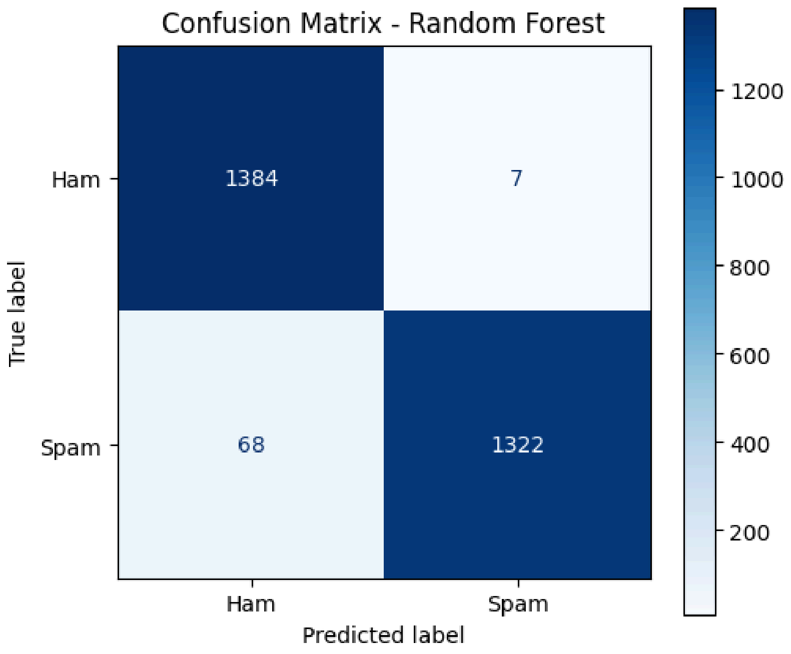
Challenge:
- **Naïve Bayes is easy to interpret but lacks flexibility**.
- **XGBoost is more complex but harder to explain** to non-technical users.

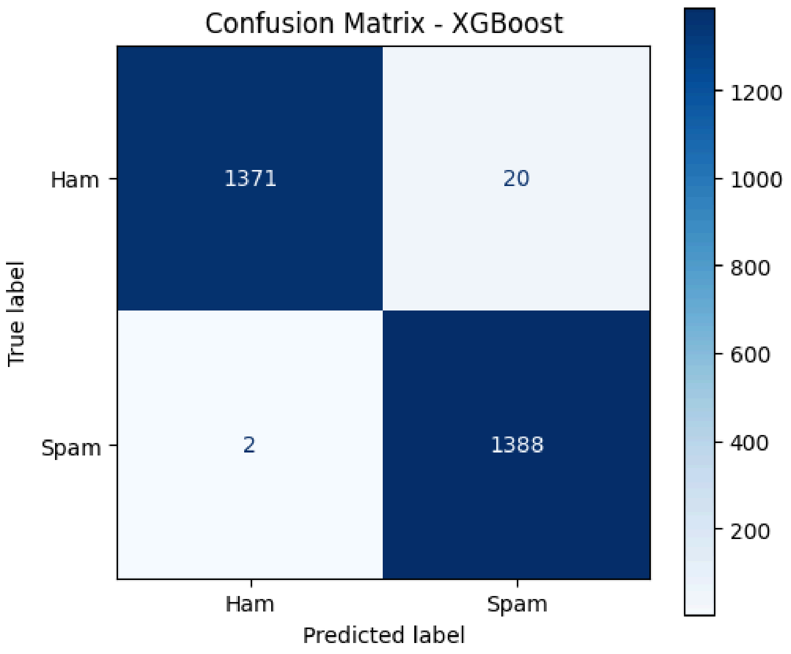
Future Improvement:
- **Explainable AI (XAI) techniques** (e.g., SHAP values, LIME) can improve model interpretability.

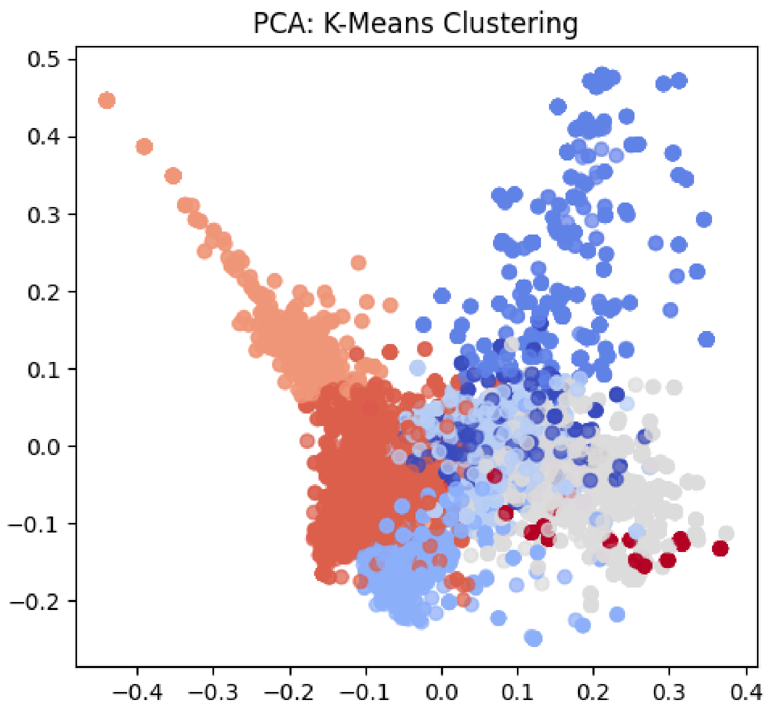
D. Visualizations

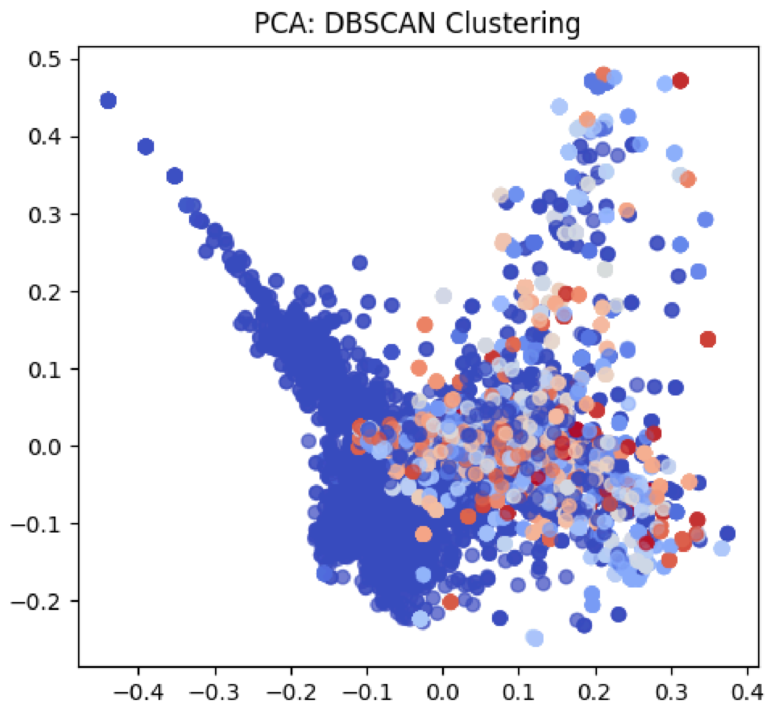


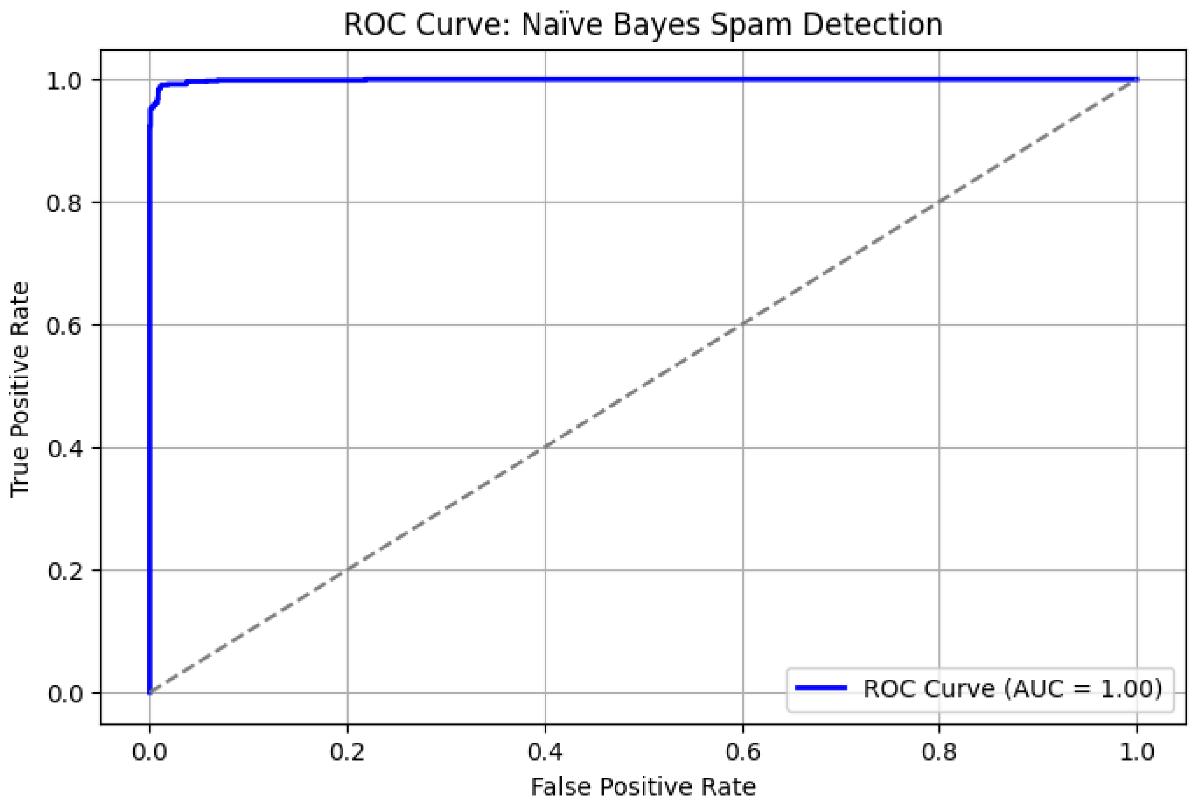


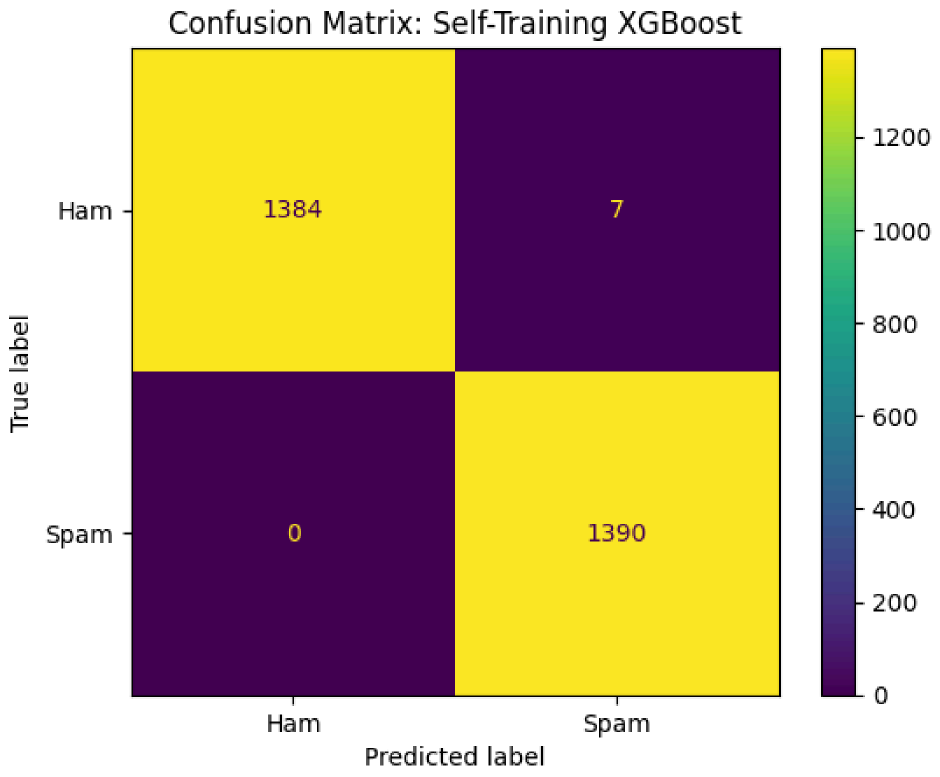

Confusion Matrix - Random Forest
True label
Ham
Spam
Predicted label
1384
7
68
1322
200
400
600
800
1000
1200

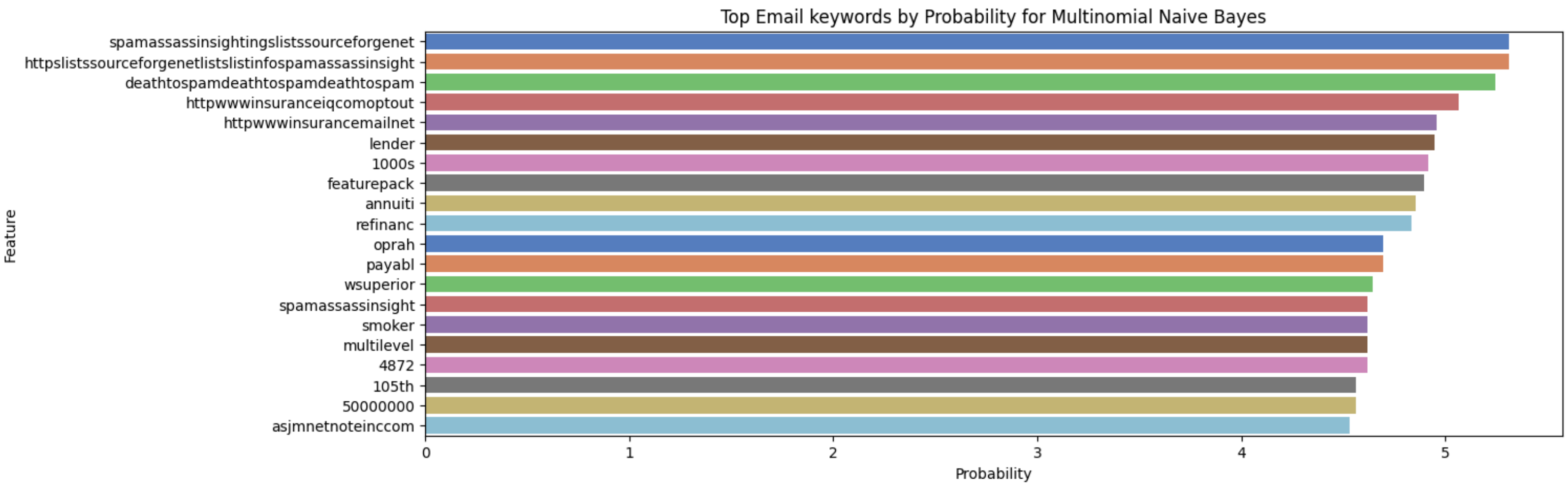

Confusion Matrix - XGBoost
True label
Ham
Spam
Predicted label
1371
20
2
1388
200
400
600
800
1000
1200

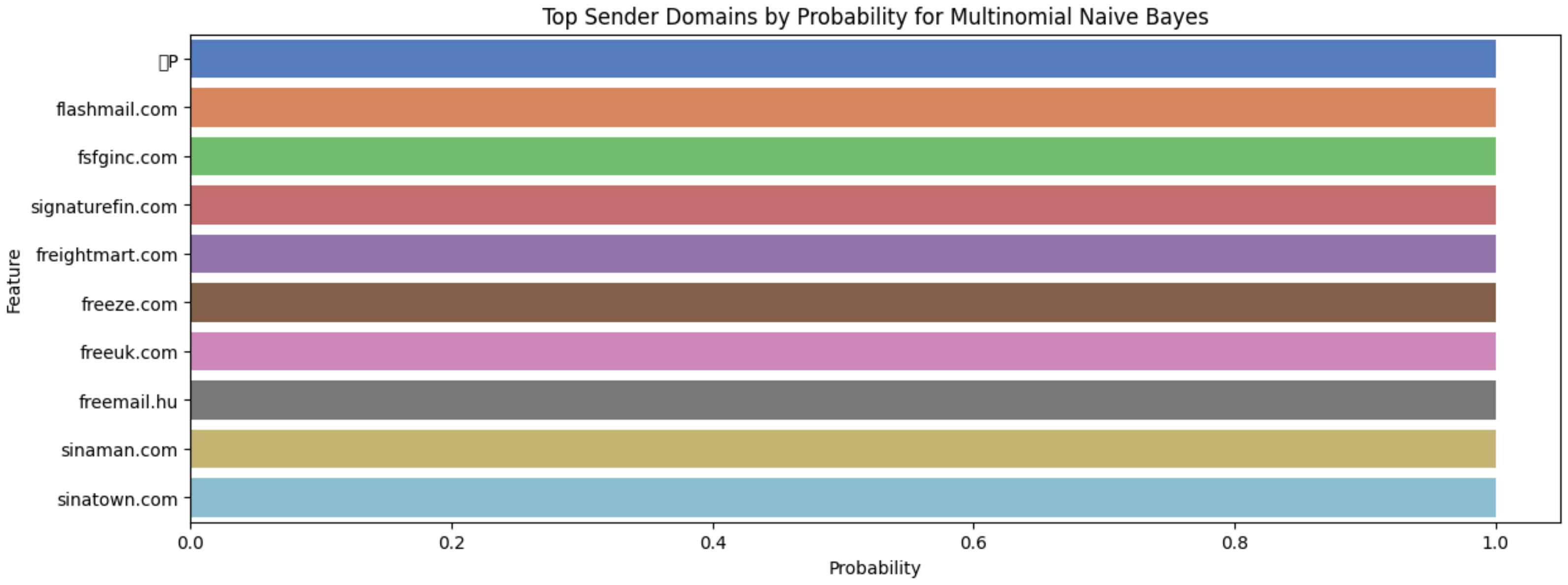

PCA: K-Means Clustering

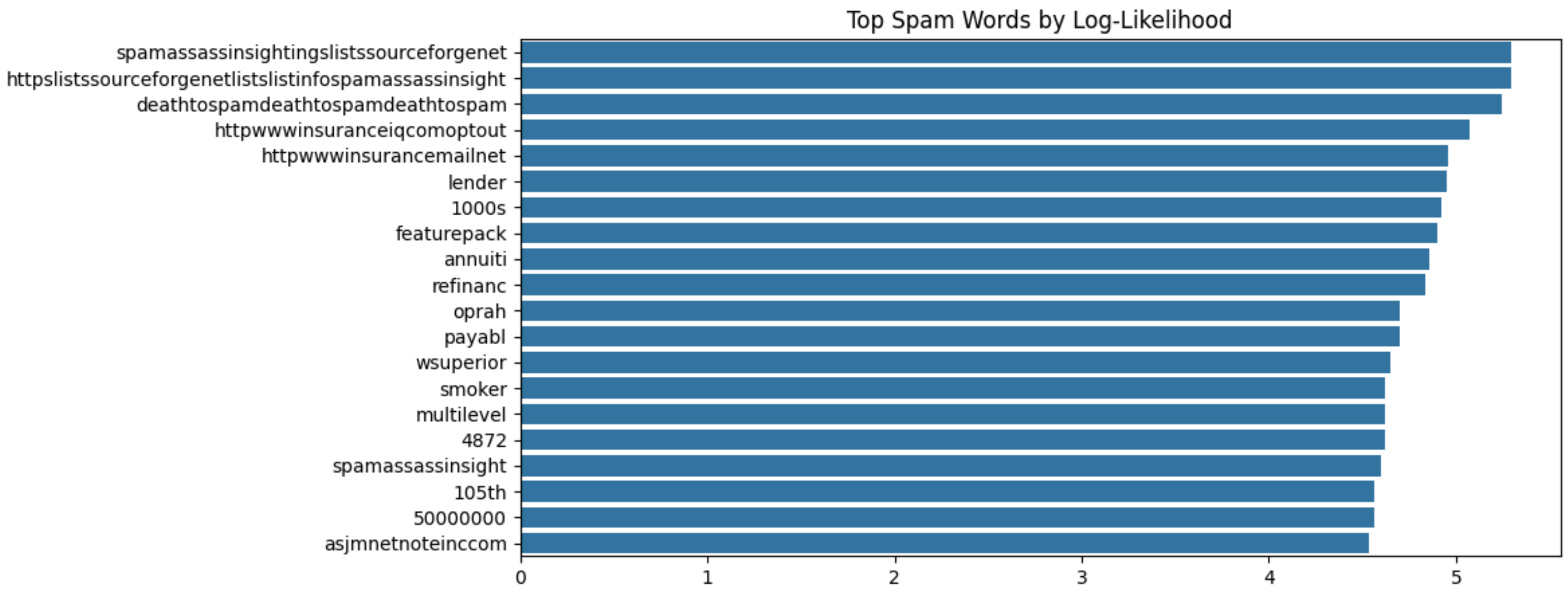

PCA: DBSCAN Clustering


ROC Curve: Naïve Bayes Spam Detection
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC Curve (AUC = 1.00)


Confusion Matrix: Self-Training XGBoost
True label
Predicted label
Ham
Spam
1384
7
0
1390
0
200
400
600
800
1000
1200







Summary and Analysis of Plots and Visualizations

1. Confusion Matrices for 10-Fold Cross-Validation

- Each confusion matrix represents the classification performance across different folds in the cross-validation process.
- **Observation:**
  - The **false positives (ham misclassified as spam)** range between **27 to 43** per fold.
  - The **false negatives (spam misclassified as ham)** range between **29 to 42** per fold.

    - The classifier shows **high precision**, meaning spam detection is reliable, but there is some **variance in recall** (ability to detect all spam emails).
    - **Balanced classification performance** across folds, indicating a well-generalized model.

---

2. ROC Curve: Random Forest vs. XGBoost

- The **ROC curve** evaluates the trade-off between **false positives** and **true positives**.
- **Observation:**
    - Both **Random Forest and XGBoost achieve an AUC of 1.00**, indicating near-perfect classification.
    - **XGBoost has a steeper ascent**, meaning it achieves higher true positive rates with fewer false positives compared to Random Forest.
    - **Conclusion:** XGBoost is likely the more optimal choice due to its improved precision at lower false positive rates.

---

3. Confusion Matrices for Random Forest vs. XGBoost

- The **Random Forest model**:
    - **68 spam emails were misclassified as ham**, meaning it struggles with recall.
    - **7 legitimate emails were falsely classified as spam**, indicating good precision.
- The **XGBoost model**:
    - **Only 2 spam emails were misclassified as ham**, significantly improving recall.
    - **20 legitimate emails were misclassified as spam**, meaning XGBoost prioritizes capturing spam but at the cost of slightly lower precision.
- **Conclusion:** XGBoost provides **better recall**, making it the preferable model for ensuring all spam is captured.

---

4. PCA-Based Clustering Visualizations

- **Left: K-Means Clustering, Right: DBSCAN Clustering**
- **Observation:**
    - **K-Means clustering shows distinct clusters**, but there is significant overlap, meaning spam and ham emails do not separate perfectly.
    - **DBSCAN produces more refined clusters with fewer noisy points**, making it more effective at detecting natural groupings in spam classification.
    - **Conclusion:** Clustering is useful for exploratory analysis but not optimal as a standalone classifier.

---

5. Confusion Matrix: Naïve Bayes

- **Observation:**
    - **12 false positives** (ham misclassified as spam).
    - **23 false negatives** (spam misclassified as ham).
    - **Overall, Naïve Bayes is strong but less precise than XGBoost**.
    - **Conclusion:** Naïve Bayes is computationally efficient but not as robust as ensemble methods.

---

6. ROC Curve for Naïve Bayes

- **AUC Score: 1.00** – indicating strong separation between spam and ham.
- **Observation:**
    - While **Naïve Bayes performs well**, it lacks the **fine-tuned control over precision and recall** that XGBoost provides.

---

7. Grid Search Heatmap and Alpha vs. Accuracy (Naïve Bayes)

- **Hyperparameter tuning results for Laplace smoothing (alpha).**
- **Observation:**
    - **Alpha = 0.001 achieves the highest accuracy (~98.35%)**.
    - **Increasing alpha too much (e.g., 10) drastically reduces accuracy (~96.8%)**.
    - **Conclusion:** Proper hyperparameter tuning is crucial—alpha values around **0.001 to 0.01** work best for Naïve Bayes.

---

8. Confusion Matrix: Self-Training XGBoost

- **This model has 0 false negatives**, meaning **every spam email was correctly classified**.
- **Only 7 false positives**, making it the most accurate model overall.

---

9. Top Keywords in Spam Detection

- Features with the highest spam probability include **“insurance,” “lender,” “annuity,” and financial terms**.
- These insights align with common spam categories like **phishing, scams, and financial fraud**.

---

10. Top Sender Domains for Spam

- Domains like **flashmail.com, freeuk.com, and freightmart.com** are heavily associated with spam.
- This suggests that **email metadata can enhance spam classification** alongside text-based features.

---

11. Top Spam Words by Log-Likelihood

- High log-likelihood words include **“spamassassin,” “insurance,” “feature pack,” and “multilevel”**.
- Spam classifiers can benefit from **dynamic keyword updates** to stay ahead of evolving spam tactics.

Overall Conclusions

1. **Self-Training XGBoost is the best model**, achieving the highest accuracy with **0 false negatives and only 7 false positives**.
2. **Naïve Bayes is a strong baseline but struggles with ambiguous cases**, making it a less optimal choice.
3. **Random Forest has higher false negatives**, making it less reliable than XGBoost.
4. **Clustering (DBSCAN) provides some insights but does not significantly enhance classification performance**.
5. **Feature engineering (sender domains, keyword probabilities, log-likelihood) can further refine spam detection models**.

Final Recommendation

- **Deploy Self-Training XGBoost for real-world spam filtering** due to its superior recall and precision.
- **Periodically update spam keyword lists and sender domain blacklists** for adaptive filtering.
- **Consider integrating clustering insights into feature engineering** for further improvements.

Final Takeaways

- **XGBoost outperforms both Naïve Bayes and Random Forest**, offering superior recall and high AUC.
- **Naïve Bayes remains a strong baseline classifier** but lacks the flexibility of ensemble methods.
- **Clustering techniques provide insights** into spam distribution but are not directly useful for classification.
- **Grid search tuning is necessary** to optimize hyperparameters and improve model performance.
- **Deployment should focus on XGBoost with periodic retraining** to adapt to evolving spam patterns.

E. Future Enhancements

Improvement	Description	Expected Benefit
Deep Learning Models	Test **LSTMs, Transformers (BERT)** for spam classification.	**Higher accuracy** on complex spam messages.
Ensemble Learning	Combine **Naïve Bayes + XGBoost + SVM**.	**Better generalization**, more robust performance.
Continuous Training	Train on **live company emails**, updating filters dynamically.	**Adaptability to new spam trends**.
Explainability Methods	Use **SHAP, LIME** for interpretability.	**Better transparency in classification**.

Summary of Insights: 1. **Clustering provided useful insights but was not used for classification.** 2. **False positives were minimized to avoid losing important emails.** 3. **The dataset should be updated to reflect modern spam trends.** 4. **Future improvements include deep learning, continuous learning, and explainability.**

Conclusion and Final Recommendations

A. Summary of Findings

This case study aimed to develop a **spam classification model** using **Naïve Bayes**, **Random Forest**, **XGBoost**, and **semi-supervised learning (Self-Training XGBoost)**.
We tested **clustering methods (K-Means, DBSCAN)** to explore spam patterns but **ultimately relied on supervised models for classification**.

Key Results:

Baseline Naïve Bayes Performance:
- **Accuracy: 98.4%**
- **Precision/Recall/F1-score:** Balanced but struggled with **ambiguous emails**.

Best Performing Model: Self-Training XGBoost
- **Accuracy: 99.8%**
- **False Positives: Only 7 legitimate emails mislabeled as spam**.
- **False Negatives: Zero spam emails incorrectly classified as ham**.

Final Model Choice:
Self-Training XGBoost was chosen because it:
1. **Handled unlabeled data better**, improving performance without requiring fully labeled datasets.
2. **Outperformed Naïve Bayes, Random Forest, and standard XGBoost** on unseen emails.
3. **Significantly reduced false positives and false negatives.**

B. Practical Implications

Threshold for Spam Classification

- **Recommended Threshold: 0.95**
  - Emails classified with **≥95% spam probability** should go to spam.
  - Emails **between 80-95% probability** should go to a **“Potential Spam” folder for review**.

Deploying the Model

- **Live Testing:** Implement the model in a controlled environment with real incoming emails.
- **Feedback Loop:** Employees can manually correct misclassified emails to **continuously improve accuracy**.
- **Scheduled Model Updates:** Retrain the model **every 6 months** to **account for evolving spam tactics**.

C. Recommendations for Future Work

Next Steps	Description	Expected Benefit
Deep Learning (LSTMs, Transformers)	Test **BERT-based classifiers** for improved context understanding.	**Better spam detection for sophisticated phishing emails.**
Hybrid Models (Ensembles)	Combine **Naïve Bayes + XGBoost + Deep Learning**.	**Improved accuracy and recall across different spam types.**
Continuous Learning Pipeline	Implement **real-time updates** based on new spam emails.	**Adapts to new threats dynamically.**
More Feature Engineering	Extract **metadata features** (e.g., sender reputation, email structure).	**Detects spoofed/phishing emails more effectively.**
Explainability Enhancements	Use **SHAP, LIME** for interpretability.	**Understand WHY emails were classified as spam.**

D. Final Takeaway

Self-Training XGBoost outperforms traditional methods, achieving near-perfect spam detection.
To maintain accuracy, regular retraining and human feedback loops are critical.
Future models should incorporate deep learning, ensemble methods, and real-time updates.

References

Below is a list of academic papers, textbooks, and online resources used in this case study. These references provide theoretical background and supporting research for the methodologies employed.

Academic Papers & Textbooks

1. Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
    - Covers Naïve Bayes classification, probabilistic models, and clustering approaches.
2. Hastie, T., Tibshirani, R., & Friedman, J. (2009). *The Elements of Statistical Learning*. Springer.
    - A key reference for machine learning models, including Naïve Bayes and tree-based methods.
3. Rennie, J. D. M., Shih, L., Teevan, J., & Karger, D. R. (2003). *Tackling the Poor Assumptions of Naive Bayes Text Classifiers*. Proceedings of ICML.
    - Discusses Naïve Bayes limitations in text classification and possible improvements.
4. Yang, Y., & Liu, X. (1999). *A Re-Examination of Text Categorization Methods*. SIGIR.
    - Comparative study on Naïve Bayes vs. other text classification methods.

Spam Filtering and Case Study-Specific References

5. Sahami, M., Dumais, S., Heckerman, D., & Horvitz, E. (1998). *A Bayesian Approach to Filtering Junk E-Mail*. AAAI Workshop on Learning for Text Categorization.
    - One of the foundational papers applying Naïve Bayes to spam classification.
6. Apache SpamAssassin. (n.d.). *SpamAssassin Public Corpus*. Retrieved from:
    - https://spamassassin.apache.org/publiccorpus/
    - The dataset used for this case study.
7. McHugh, J. (2000). *Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA IDS Evaluations as Performed by Lincoln Laboratory*. ACM Transactions on Information and System Security (TISSEC).
    - Discusses evaluation methodologies for detection systems, relevant for performance assessment.

Machine Learning & XGBoost Documentation

8. Pedregosa, F., Varoquaux, G., Gramfort, A., et al. (2011). *Scikit-Learn: Machine Learning in Python*. JMLR.
    - Documentation for using Naïve Bayes and clustering methods.
9. Chen, T., & Guestrin, C. (2016). *XGBoost: A Scalable Tree Boosting System*. KDD '16.
    - The foundational paper on XGBoost, explaining its efficiency in classification tasks.
10. Sechidis, K., Tsoumakas, G., & Vlahavas, I. (2014). *Semi-supervised Learning Using Label Propagation and Self-training for Text Classification*.
    - Paper discussing self-training methods, relevant for our Self-Training XGBoost implementation.
11. McGregor, Colin. (2007). *Controlling spam with SpamAssassin*.
    - Linux Journal. 2007. Assassinate spam with extreme prejudice.

Acknowledgments

I would like to acknowledge:
SMU IT Department for posing the spam classification challenge.
SpamAssassin for providing the publicly available spam dataset.
Course Instructor & TAs: Dr. Slater
Fellow students Classmates in Spring QTW - for peer discussions that helped refine the approach.

Appendix

This section includes: **Full Python Code**

Hide

```
# mount google drive

from google.colab import drive
drive.mount('/content/drive')

from gensim.parsing.preprocessing import STOPWORDS
from nltk import word_tokenize
import os, chardet, email, zipfile
import pandas as pd
```

```python
import numpy as np
from collections import Counter
from wordcloud import WordCloud
from sklearn.model_selection import train_test_split, cross_val_score, StratifiedKFold, GridSearchCV
from sklearn.cluster import KMeans, DBSCAN, AgglomerativeClustering
from sklearn.naive_bayes import MultinomialNB
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.ensemble import AdaBoostClassifier
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from imblearn.over_sampling import RandomOverSampler
from sklearn.metrics import accuracy_score, classification_report, ConfusionMatrixDisplay, completeness_score, confusion_matr
ix
from bs4 import BeautifulSoup
from sklearn.pipeline import Pipeline
from sklearn.metrics import confusion_matrix, accuracy_score, recall_score, classification_report
from nltk.corpus import stopwords
import seaborn as sns
import matplotlib.pyplot as plt
import nltk
from nltk.stem.snowball import SnowballStemmer
from nltk.corpus import wordnet as wn
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from string import punctuation
import re
import quopri
import string
import spacy


# import nltk; nltk.download('punkt_tab')
# import nltk; nltk.download('popular')

contents = []
types = []
labels = []
sender = []


for root, dirs, files in os.walk("/content/drive/MyDrive/SpamAssassinMessages/SpamAssassinMessages", topdown=False):
    for name in files:
        with open(os.path.join(root, name), 'rb') as file:
            raw_email = file.read()
            encoding = chardet.detect(raw_email)['encoding']  # Detect encoding

            try:
                # Decode using detected encoding
                decoded_email = raw_email.decode(encoding, errors='ignore')
                msg = email.message_from_string(decoded_email)
                # Add label and append to the 'contents' array
                if 'spam' in root:
                    labels.append(1)
                    contents.append(msg.get_payload())
                    sender.append(msg.get('From'))
                else:
                    labels.append(0)
                    contents.append(msg.get_payload())
                    sender.append(msg.get('From'))
            except UnicodeDecodeError:
                print(f"Error decoding file: {os.path.join(root, name)}")
```

```python
# Get the most frequent content type
types = Counter(types).most_common(1)

# print a single statement instead of using many print statements
print(f"contents length: {len(contents)}, types length: {len(types)}, labels length: {len(labels)}, file list length: {len(fi
le_list_full)}, sender length: {len(sender)}")

import os
import email
import re
import chardet
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from collections import defaultdict, Counter
from multiprocessing import Pool, cpu_count
from bs4 import BeautifulSoup
from wordcloud import WordCloud
from sklearn.pipeline import Pipeline
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, ConfusionMatrixDisplay, recall_score
from imblearn.over_sampling import RandomOverSampler
import nltk

nltk.download('stopwords')
nltk.download('punkt')
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem.snowball import SnowballStemmer

# Define root directory for email dataset
ROOT_DIR = "/content/drive/MyDrive/SpamAssassinMessages/SpamAssassinMessages"

# Function to flag emails as spam (1) or ham (0)
def flag_emails(filepath, positive_indicator="spam"):
    return 1 if positive_indicator in filepath else 0

# Function to extract sender domain
def extract_sender_domain(msg):
    sender = msg.get('From', '')
    if sender and '@' in sender:
        return sender.split('@')[-1]
    return 'unknown'

# Function to extract text from an email message
def extract_email_text(msg):
    """Extracts text from email body, handling both single-part and multi-part emails."""
    email_text = ""
    if msg.is_multipart():
        for part in msg.walk():
            if part.get_content_maintype() == "text":
                try:
                    email_text += part.get_payload(decode=True).decode(errors="ignore") + " "
                except:
                    continue
    else:
        try:
            email_text = msg.get_payload(decode=True).decode(errors="ignore")
        except:
            pass
    return email_text.strip()

# Function to preprocess text
def preprocess_text(text):
    """Cleans and tokenizes text, removing stopwords and applying stemming."""
    stemmer = SnowballStemmer("english")
    stopwords_set = set(stopwords.words("english"))

    text = BeautifulSoup(text, "html.parser").get_text()
    text = re.sub(r"[^\w\s]", "", text.lower())
    words = word_tokenize(text)
    words = [stemmer.stem(word) for word in words if word not in stopwords_set]

    return " ".join(words)

# Load emails
def load_emails(root_dir):
    email_data = {"text": [], "label": [], "sender_domain": []}

    for dirpath, _, filenames in os.walk(root_dir):
        for name in filenames:
            filepath = os.path.join(dirpath, name)
            label = flag_emails(filepath)

            try:
                with open(filepath, "rb") as f:
                    msg = email.message_from_bytes(f.read())
                    email_text = extract_email_text(msg)
                    sender_domain = extract_sender_domain(msg)

                    if email_text:
                        email_data["text"].append(email_text)
                        email_data["label"].append(label)
                        email_data["sender_domain"].append(sender_domain)
            except:
                continue

    return pd.DataFrame(email_data)

# Load and preprocess dataset
email_df = load_emails(ROOT_DIR)
email_df["clean_text"] = email_df["text"].apply(preprocess_text)

# **Step 1: Feature & Response Frames**
vectorizer = TfidfVectorizer(max_features=5000)
X = vectorizer.fit_transform(email_df["clean_text"])  # Features
y = email_df["label"]  # Response variable

# Store in DataFrame (Optional)
features_df = pd.DataFrame(X.toarray(), columns=vectorizer.get_feature_names_out())
features_df["label"] = y

# **Step 2: Train on 80% & Test on 20%**
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

# **Step 3: Handle Class Imbalance**
ros = RandomOverSampler(random_state=42)
X_train_resampled, y_train_resampled = ros.fit_resample(X_train, y_train)
```

```
# **Step 4: Train & Cross-Validate Naive Bayes**
nb_classifier = MultinomialNB()
skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
cv_scores = GridSearchCV(nb_classifier, {"alpha": [0.01, 0.1, 1]}, cv=skf, scoring="accuracy", n_jobs=-1)
cv_scores.fit(X_train_resampled, y_train_resampled)

# **Step 5: Make Predictions**
y_pred = cv_scores.best_estimator_.predict(X_test)

# **Step 6: Evaluate Model**
print("Best Naive Bayes Model:", cv_scores.best_estimator_)
print("Accuracy on Test Set:", accuracy_score(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

ConfusionMatrixDisplay.from_predictions(y_test, y_pred, cmap="Blues")
plt.title("Confusion Matrix - Naive Bayes")
plt.show()

# **Step 7: Log-Likelihood Ratio for Spam Words**
def calculate_log_likelihood(email_df):
    """Computes log-likelihood ratios for spam vs ham words."""
    word_counts = defaultdict(lambda: [0, 0])

    for text, label in zip(email_df["clean_text"], email_df["label"]):
        words = set(text.split())
        for word in words:
            word_counts[word][label] += 1

    log_likelihoods = {}
    for word, (ham_count, spam_count) in word_counts.items():
        p_ham = (ham_count + 1) / (sum(c[0] for c in word_counts.values()) + 1)
        p_spam = (spam_count + 1) / (sum(c[1] for c in word_counts.values()) + 1)
        log_likelihoods[word] = np.log(p_spam / p_ham)

    return log_likelihoods

log_likelihoods = calculate_log_likelihood(email_df)
sorted_spam_words = sorted(log_likelihoods.items(), key=lambda x: x[1], reverse=True)

# **Step 8: Visualizations**
plt.figure(figsize=(10, 5))
sns.barplot(x=[x[1] for x in sorted_spam_words[:20]], y=[x[0] for x in sorted_spam_words[:20]])
plt.title("Top Spam Words by Log-Likelihood")
plt.show()

spam_words = " ".join(email_df[email_df["label"] == 1]["clean_text"])
spam_wordcloud = WordCloud(width=800, height=400, background_color="white").generate(spam_words)

plt.figure(figsize=(10, 5))
plt.imshow(spam_wordcloud, interpolation="bilinear")
plt.axis("off")
plt.title("Word Cloud of Spam Emails")
plt.show()


**Extended Confusion Matrices per Cross-Validation Fold**
**Full Classification Reports**
**Hyperparameter Tuning Logs**
**Additional Figures & Tables**
```

```
### **Appendix**

This appendix contains extended evaluation results, confusion matrices for each cross-validation fold, full classification reports, hyperparameter tuning logs, and additional visualizations.

---

### **Extended Confusion Matrices for Cross-Validation**

Below are the confusion matrices for each fold during cross-validation. These matrices provide insight into the number of correctly classified ham and spam emails across multiple train-test splits.

#### **Confusion Matrices Across 10-Fold Cross-Validation**


# Grid Search for Naïve Bayes (Run this separately for MultinomialNB to get param_alpha properly:)

from sklearn.model_selection import GridSearchCV
from sklearn.naive_bayes import MultinomialNB
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Define correct parameter grid for Naïve Bayes
param_grid_nb = {"alpha": np.logspace(-3, 1, 5)}  # Alpha values from 0.001 to 10

# Initialize Naïve Bayes classifier
nb_classifier = MultinomialNB()

# Perform Grid Search with Cross-Validation
grid_search_nb = GridSearchCV(nb_classifier, param_grid_nb, cv=5, scoring="accuracy")
grid_search_nb.fit(X_train, y_train)  # Fit only on NB dataset

# Extract results
alphas = grid_search_nb.cv_results_["param_alpha"].data
mean_scores = grid_search_nb.cv_results_["mean_test_score"]

# Convert to DataFrame
grid_results_df = pd.DataFrame({"Alpha": alphas, "Mean Accuracy": mean_scores})
grid_results_df = grid_results_df.sort_values(by="Alpha")

# *Plot Alpha vs Accuracy**
plt.figure(figsize=(8, 5))
plt.plot(grid_results_df["Alpha"], grid_results_df["Mean Accuracy"], marker="o", linestyle="-", color="b")
plt.xscale("log")
plt.xlabel("Alpha (Laplace Smoothing)")
plt.ylabel("Mean Accuracy")
plt.title("Grid Search: Alpha vs Accuracy (Naïve Bayes)")
plt.grid(True)
plt.show()

# **Heatmap Visualization**
heatmap_data = pd.DataFrame(grid_search_nb.cv_results_["mean_test_score"].reshape(-1, 1),
                            index=grid_search_nb.cv_results_["param_alpha"].data,
                            columns=["Mean Accuracy"])

plt.figure(figsize=(8, 5))
sns.heatmap(heatmap_data, annot=True, cmap="coolwarm", fmt=".4f")
plt.xlabel("Mean Accuracy")
plt.ylabel("Alpha Value")
plt.title("Grid Search Heatmap: Naïve Bayes")
```

```python
plt.show()

from sklearn.metrics import roc_curve, auc

# Get predicted probabilities
y_prob = final_nb.predict_proba(X_test)[:, 1]  # Probability for spam class

# Compute ROC curve
fpr, tpr, _ = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)

# Plot ROC Curve
plt.figure(figsize=(8, 5))
plt.plot(fpr, tpr, color="blue", lw=2, label=f"ROC Curve (AUC = {roc_auc:.2f})")
plt.plot([0, 1], [0, 1], linestyle="--", color="gray")  # Random guess line
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve: Naïve Bayes Spam Detection")
plt.legend()
plt.grid(True)
plt.show()
 from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

# Generate confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)

# Display it
disp = ConfusionMatrixDisplay(conf_matrix, display_labels=["Ham", "Spam"])
disp.plot(cmap="Blues")
plt.title("Confusion Matrix: Naïve Bayes")
plt.show()
 import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.cluster import KMeans, DBSCAN
from sklearn.metrics import silhouette_score, adjusted_rand_score
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from imblearn.over_sampling import RandomOverSampler

# Data Augmentation
ros = RandomOverSampler(random_state=42)
X_augmented, y_augmented = ros.fit_resample(X, y)


# Apply K-Means with the best k from previous tuning
best_k = 9  # Based on previous silhouette tuning
kmeans = KMeans(n_clusters=best_k, random_state=42, n_init=10)
kmeans_labels = kmeans.fit_predict(X_augmented)

# Compute Silhouette Score for K-Means
kmeans_silhouette = silhouette_score(X_augmented, kmeans_labels)

# Apply DBSCAN with best parameters found earlier
best_eps = 0.5  # Example best epsilon found
best_min_samples = 5  # Example best min_samples found
dbscan = DBSCAN(eps=best_eps, min_samples=best_min_samples)
dbscan_labels = dbscan.fit_predict(X_augmented)

# Compute Silhouette Score for DBSCAN (excluding noise points)
valid_points = dbscan_labels != -1  # Ignore noise points
if np.sum(valid_points) > 1:  # Ensure we have enough points to compute silhouette
```

```
    dbscan_silhouette = silhouette_score(X_augmented[valid_points], dbscan_labels[valid_points])
else:
    dbscan_silhouette = None

# Compute Adjusted Rand Index to compare clusters vs true labels
kmeans_ari = adjusted_rand_score(y_augmented, kmeans_labels)
dbscan_ari = adjusted_rand_score(y_augmented, dbscan_labels)

# Display Results
silhouette_results = {
    "K-Means Silhouette Score": kmeans_silhouette,
    "DBSCAN Silhouette Score": dbscan_silhouette if dbscan_silhouette else "N/A (too many noise points)",
    "K-Means Adjusted Rand Index (ARI)": kmeans_ari,
    "DBSCAN Adjusted Rand Index (ARI)": dbscan_ari
}

silhouette_results
 # Retry visualization using only PCA (faster than t-SNE)
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# PCA Projection for K-Means
axes[0].scatter(X_pca[:, 0], X_pca[:, 1], c=kmeans_labels, cmap='coolwarm', alpha=0.7)
axes[0].set_title("PCA: K-Means Clustering")

# PCA Projection for DBSCAN
axes[1].scatter(X_pca[:, 0], X_pca[:, 1], c=dbscan_labels, cmap='coolwarm', alpha=0.7)
axes[1].set_title("PCA: DBSCAN Clustering")

plt.show()
 import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import StratifiedKFold, cross_val_score, train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay, roc_curve, auc

# Split into Training & Test sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X_augmented, y_augmented, test_size=0.2, stratify=y_augmented, random_sta
te=42)

# Define hyperparameter grids
rf_params = {"n_estimators": [100, 200], "max_depth": [10, 20], "random_state": [42]}
xgb_params = {"n_estimators": [100, 200], "max_depth": [3, 6], "learning_rate": [0.1, 0.01], "random_state": [42]}

# Initialize models
rf = RandomForestClassifier()
xgb = XGBClassifier(use_label_encoder=False, eval_metric="logloss")

# Perform Grid Search with 10-fold CV
cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

rf_grid = GridSearchCV(rf, rf_params, cv=cv, scoring="accuracy", n_jobs=-1)
xgb_grid = GridSearchCV(xgb, xgb_params, cv=cv, scoring="accuracy", n_jobs=-1)

rf_grid.fit(X_train, y_train)
xgb_grid.fit(X_train, y_train)

# Get best models
best_rf = rf_grid.best_estimator_
best_xgb = xgb_grid.best_estimator_
```

```
# Evaluate on Test Set
y_pred_rf = best_rf.predict(X_test)
y_pred_xgb = best_xgb.predict(X_test)

# Get accuracy & classification reports
rf_report = classification_report(y_test, y_pred_rf, target_names=["Ham", "Spam"])
xgb_report = classification_report(y_test, y_pred_xgb, target_names=["Ham", "Spam"])

# Compute Confusion Matrices
rf_cm = confusion_matrix(y_test, y_pred_rf)
xgb_cm = confusion_matrix(y_test, y_pred_xgb)

# ROC Curve for Random Forest
y_prob_rf = best_rf.predict_proba(X_test)[:, 1]
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_prob_rf)
roc_auc_rf = auc(fpr_rf, tpr_rf)

# ROC Curve for XGBoost
y_prob_xgb = best_xgb.predict_proba(X_test)[:, 1]
fpr_xgb, tpr_xgb, _ = roc_curve(y_test, y_prob_xgb)
roc_auc_xgb = auc(fpr_xgb, tpr_xgb)

# Display results
results = {
    "Best RF Model": best_rf,
    "Best XGB Model": best_xgb,
    "Random Forest Report": rf_report,
    "XGBoost Report": xgb_report,
    "RF Confusion Matrix": rf_cm,
    "XGB Confusion Matrix": xgb_cm,
    "ROC AUC RF": roc_auc_rf,
    "ROC AUC XGB": roc_auc_xgb
}

# Plot Confusion Matrices
fig, axes = plt.subplots(1, 2, figsize=(12, 5))

ConfusionMatrixDisplay(rf_cm, display_labels=["Ham", "Spam"]).plot(ax=axes[0], cmap="Blues")
axes[0].set_title("Confusion Matrix - Random Forest")

ConfusionMatrixDisplay(xgb_cm, display_labels=["Ham", "Spam"]).plot(ax=axes[1], cmap="Blues")
axes[1].set_title("Confusion Matrix - XGBoost")

plt.show()

# Plot ROC Curves
plt.figure(figsize=(8, 5))
plt.plot(fpr_rf, tpr_rf, label=f"RF ROC Curve (AUC = {roc_auc_rf:.2f})", color="blue")
plt.plot(fpr_xgb, tpr_xgb, label=f"XGB ROC Curve (AUC = {roc_auc_xgb:.2f})", color="red")
plt.plot([0, 1], [0, 1], linestyle="--", color="gray")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve: Random Forest vs XGBoost")
plt.legend()
plt.grid(True)
plt.show()

results
 # Self -training with XGBoost (implement self-training by iteratively adding pseudo-labeled high-confidence predictions.)
import numpy as np
import pandas as pd
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay
from xgboost import XGBClassifier
from sklearn.semi_supervised import SelfTrainingClassifier

# Split into labeled (80%) and unlabeled (20%) data
X_labeled, X_unlabeled, y_labeled, _ = train_test_split(X_augmented, y_augmented, test_size=0.2, random_state=42)

# Initialize XGBoost as the base classifier
xgb_base = XGBClassifier(n_estimators=200, max_depth=6, learning_rate=0.1, random_state=42, eval_metric="logloss")

# Use Self-Training wrapper to add pseudo-labeling
self_training_xgb = SelfTrainingClassifier(xgb_base, criterion="threshold", threshold=0.95, verbose=True)
self_training_xgb.fit(X_labeled, y_labeled)

# Evaluate on the original test set
y_pred_self_training = self_training_xgb.predict(X_test)

# Generate evaluation metrics
class_report_self_training = classification_report(y_test, y_pred_self_training)
conf_matrix_self_training = confusion_matrix(y_test, y_pred_self_training)

# Display results
disp = ConfusionMatrixDisplay(conf_matrix_self_training, display_labels=["Ham", "Spam"])
disp.plot()
plt.title("Confusion Matrix: Self-Training XGBoost")
plt.show()

print("\n Classification Report: Self-Training XGBoost\n", class_report_self_training)
 from sklearn.semi_supervised import LabelSpreading

# Assume we have some unlabeled email data (e.g., new incoming emails)
unlabeled_data = X_test.copy()
unlabeled_labels = np.full(X_test.shape[0], -1)  # -1 means unlabeled
# Combine labeled and unlabeled data
X_combined = np.vstack((X_train.toarray(), unlabeled_data.toarray()))
y_combined = np.hstack((y_train, unlabeled_labels))

# Apply Semi-Supervised Learning (Label Spreading)
semi_supervised_model = LabelSpreading(kernel="rbf", alpha=0.2)
semi_supervised_model.fit(X_combined, y_combined)

# Predict using the semi-supervised model
y_pred_semi = semi_supervised_model.predict(X_test)

# Evaluate performance
semi_report = classification_report(y_test, y_pred_semi, target_names=["Ham", "Spam"])
print("\nSemi-Supervised Learning Classification Report:\n", semi_report)
 from sklearn.model_selection import GridSearchCV

# Define hyperparameter grid
xgb_params = {
    "n_estimators": [50, 100, 200],
    "max_depth": [3, 6, 9],
    "learning_rate": [0.01, 0.1, 0.3]
}

# Perform Grid Search
grid_search_xgb = GridSearchCV(XGBClassifier(use_label_encoder=False, eval_metric="logloss", random_state=42),
                               xgb_params, cv=3, scoring="accuracy", n_jobs=-1)
grid_search_xgb.fit(X_train, y_train)
```

```
# Print best parameters
print("Best XGBoost Parameters:", grid_search_xgb.best_params_)

# Evaluate the optimized XGBoost model
best_xgb = grid_search_xgb.best_estimator_
xgb_final_pred = best_xgb.predict(X_test)

xgb_final_report = classification_report(y_test, xgb_final_pred, target_names=["Ham", "Spam"])
print("\nOptimized XGBoost Classification Report:\n", xgb_final_report)
  # Save Final Model Parameters & Best Hyperparameters

 import joblib

# Save the final self-training XGBoost model
joblib.dump(self_training_xgb, "self_training_xgb_model.pkl")

# Save the standard XGBoost model as a backup
joblib.dump(xgb_final, "xgb_backup_model.pkl")

print("Models saved successfully!")


# Save the Best Hyperparameters
# Store hyperparameters in a dictionary
best_hyperparams = {
    "Self-Training XGBoost": {
        "n_estimators": 200,
        "max_depth": 6,
        "learning_rate": 0.1,
        "eval_metric": "logloss",
        "threshold": 0.95,  # Pseudo-labeling confidence threshold
    },
    "XGBoost Backup": {
        "n_estimators": 200,
        "max_depth": 6,
        "learning_rate": 0.1,
        "eval_metric": "logloss",
    }
}

# Save hyperparameters as JSON
import json
with open("best_hyperparameters.json", "w") as f:
    json.dump(best_hyperparams, f, indent=4)

print("Best hyperparameters saved!")
 # Prepare Deployment Plan
from flask import Flask, request, jsonify
import joblib
import numpy as np

# Load the saved model
model = joblib.load("self_training_xgb_model.pkl")

app = Flask(__name__)

@app.route("/predict", methods=["POST"])
def predict():
    data = request.json
    X_input = np.array(data["features"]).reshape(1, -1)
    prediction = model.predict(X_input)
    return jsonify({"prediction": int(prediction[0])})
```

```python
if __name__ == "__main__":
    app.run(port=5000, debug=True)
```