PROGRAMMING ASSIGNMENT: NAIVE BAYES SPAM FILTER

Frilles, Roljohn C. rjthehokage@gmail.com	Macaraeg, Paul Angelo O. paulmacaraeg24@gmail.com	Manansala, Roan P. manansalaroan11@gmail.com	Manucom, Precious Grace Deborah deborahgrace0118@gmail.com

Executive Summary

This study explores the development of a spam detection system using the Multinomial Naive Bayes classifier applied to the TREC06p dataset, which contains over 37,000 labeled emails. After preprocessing the emails into a bag-of-words representation and applying Laplace smoothing, the model achieved an accuracy of 94.01%, a precision of 97.45%, a recall of 93.34%, and an F1-score of 95.35%. Parameter tuning showed that a lower lambda value ($\alpha = 0.005$) yielded the best precision and recall. Lastly, to evaluate the classifier's performance under dimensionality reduction, only the top 200 most informative words, identified using the log-odds ratio, were used for classification. This reduced feature set resulted in a recall of 98.95% and a precision of 80.08%, indicating that while the model becomes more sensitive to spam, it may also misclassify more legitimate emails. These results demonstrate the model's strong performance and adaptability, even under feature constraints, making it suitable for practical spam filtering tasks or as a reliable baseline for more complex systems.

1 Introduction

Spam, or unsolicited bulk email, represents a significant nuisance and security risk in digital communication. Effectively filtering spam is crucial for maintaining user productivity, conserving network resources, and protecting users from malicious content. Among various machine learning techniques applied to this problem, the Naive Bayes classifier stands out due to its simplicity, efficiency, and surprisingly strong performance, particularly on text classification tasks like spam detection. This project implements a Naive Bayes classifier specifically tailored for differentiating legitimate emails ('ham') from spam, utilizing the TREC06p dataset as a benchmark corpus.

The foundation of the classifier lies in **Bayes' Theorem**, which provides a way to calculate the probability of a hypothesis (e.g., an email being spam) given the observed evidence (e.g., the words in the email). For an email represented by a feature vector **x** (containing words $x_1, x_2, ..., x_v$, where v is the vocabulary size), the probability of it belonging to a class c (where c is either 'spam' or 'ham') is given by:

$$P(c \mid x) = \frac{P(x \mid c)\, P(c)}{P(x)} \quad \text{(Eq. 1)}$$

Where:

- $P(c \mid x)$ is the **posterior probability**: the probability that the email belongs to class c given its words **x**. We want to determine this.
- $P(c \mid x)$ is the **likelihood**: the probability of observing the words **x** given that the email belongs to a class c.

- $P(c)$ is the **prior probability**: the overall probability of an email belonging to the class c, independent of its content.
- $P(x)$ is the **evidence**: the overall probability of observing the words **x**. We note here that since $P(x)$ it is the same for both classes (i.e., ‘spam’ and ‘ham’) when comparing posterior probabilities, it can be ignored for classification purposes.

The “naive” aspect of the Naive Bayes classifier comes from the simplifying assumption that all features (i.e., words in the email) are **conditionally independent** given the class. This means the presence or probability of one word does not affect the presence or likelihood of another word, given that we know whether the email is spam or ham. While this assumption rarely holds in reality (words in language are often correlated), it drastically simplifies the computation of the likelihood term:

$$P(x \mid c) \approx \prod_{i=1}^{v} P(x_i \mid c) \quad \text{(Naive Bayes Assumption)}$$

Thus, the classification decision rule becomes finding the class c that maximizes the posterior probability, which is proportional to:

$$P(c \mid x) \propto P(c) \prod_{i=1}^{v} P(x_i \mid c) \quad \text{(Eq. 2 - Classification Rule)}$$

In this implementation, we adopt a Multinomial Naive Bayes approach. Instead of just considering the presence or absence of words, we consider their frequency. The likelihood $P(x_i \mid c)$, representing the probability of a specific word w (where $w = x_i$) given class c, is estimated from the training data based on the frequency of the word within documents of that class:

$$P(w \mid c) = \frac{Count(w,c)}{TotalWords(c)}$$

Where:

- $Count(w, c)$ is the total number of times word w appears in all training documents belonging to class c.
- $TotalWords(c)$ is the total number of words (including repetitions) in all training documents belonging to class c.

A critical challenge arises when a word encountered during testing was not seen in the training data for a particular class. This would lead to a zero probability [$P(w \mid c) = 0$], causing the entire product in Eq. 2 to become zero, regardless of other evidence. To address this, **Laplace (Additive) Smoothing** is employed. A small smoothing parameter α (often referred to as lambda λ, in literature) is added to each word count, and the denominator is adjusted accordingly to ensure that the probabilities sum to 1. The smoothed likelihood becomes:

$$P(w \mid c) = \frac{Count(w,c) + \alpha}{TotalWords(c) + \alpha\,|V|} \quad \text{(Eq. 3 - Smoothed Likelihood)}$$

Where:

- α is the smoothing parameter (e.g., 1 for standard Laplace smoothing, but other values like 0.5, 0.1, 0.005 are explored).
- $|V|$ is the size of the total vocabulary (the number of unique words across all training documents).

During prediction, calculating the product of many small probabilities can lead to numerical underflow. To mitigate this, computations are performed using the sum of **log probabilities**:

$$\log P(c \mid x) \propto \log P(c) + \sum_{i=1}^{v} \log P(x_i \mid c)$$ (Eq. 4 - Log Probability Calculation)

The class assigned to a new email is the one yielding the higher log probability score.

1.1 Objectives

1. To implement a Multinomial Naive Bayes classifier from scratch for spam email filtering.
2. To preprocess and prepare the TREC06p email dataset for training and evaluation.
3. To calculate and report prior and smoothed conditional probabilities.
4. To evaluate the classifier's performance using standard metrics (Precision, Recall, Accuracy, F1-score).
5. To investigate the impact of Laplace smoothing parameter (α/λ) values on performance.
6. To implement and evaluate a feature selection method based on informative words (log odds ratio) to potentially improve the classifier.

1.2 Scope and Limitations

The project encompasses data loading, preprocessing (i.e., tokenization, lowercasing), manual implementation of Multinomial Naive Bayes training (priors, likelihoods with Laplace smoothing), prediction using log probabilities, evaluation, and feature selection based on word informativeness using the TREC06p dataset.

The "naive" assumption of conditional independence is a primary limitation. The preprocessing is relatively simple (only alphabetic tokens, no stemming/lemmatization, ignoring headers/metadata explicitly in the manual implementation). The evaluation is performed on a single dataset (TREC06p). Feature representation is limited to word frequencies (bag-of-words). Also, word order and context are ignored.

2 Case Context and Data Description

2.1 Problem Statement

The core problem is to develop and evaluate a supervised machine learning model capable of automatically classifying email messages into one of two categories: legitimate ('ham') or unsolicited spam, based solely on the textual content of the emails. The goal is to achieve high accuracy in classification, particularly minimizing the misclassification of legitimate emails as spam (false positives) while effectively identifying actual spam.

2.2 Dataset Overview

- **Source**: The dataset used is the TREC06p (Text Retrieval Conference 2006 Public Spam Corpus), a standard benchmark dataset for spam filtering research. As seen in the `naive_bayes.ipynb` notebook, the data is in a directory structure (`trec06p-cs280`) containing email files and a central `labels` file mapping file paths to 'ham' or 'spam' labels.
- **Number of records and attributes**: The notebook code successfully reads and processes **37, 758** email files (as indicated by `len(texts)` output `(37758, 37758)`). The data is split into a training set (80%, approximately 30,207 emails) and a test set (20%, approximately 7,551 emails). The primary attributes are the words derived from the email text, forming a potentially large vocabulary. The target attribute is the binary class label ('ham' or 'spam'). The notebook calculates the size of the ham and spam vocabularies derived from the training set as 137,155 and 59,675 unique words, respectively, with a total combined training vocabulary (`total_vocabulary`) of **174, 386** unique words.
- **Key features**: The defining features for classification are the individual words (tokens) extracted from the body of emails. The frequency of these words within each class (ham/spam) is used to build the Naive Bayes model. While email headers and metadata exist, the implemented preprocessing focuses on the tokenized body content.
- **Preprocessing steps**: The preprocessing pipeline, as implemented in `naive_bayes.ipynb` includes:
    1. **File Reading**: Loading email content from individual files, handling potential character encoding issues using the `read_file` function (employing `errors=" replace"`).
    2. **Label Association**: Reading the `labels` file to associate each email file path with its correct 'ham' or 'spam' classification.
    3. **Path Correction**: Adjusting file paths (`path.replace('../', 'trec06p-cs280/;, 1)`).
    4. Tokenization: Parsing the email content using the parse_documents function. This involves:
        - Using regular expressions (`re.findall(r'\b[a-zA-z]+\b', doc)`) to extract sequences consisting only of alphabetic characters, effectively treating these as words and discarding numbers, punctuation (except potentially trailing ones handled implicitly by word boundaries), and other symbols. This aligns with the assignment's definition of a word.
        - Converting all extracted tokens to **lowercase** (`token.lower()`) to ensure words like "Free" and "free" are treated as the same feature.
    5. Data Splitting: Dividing the dataset (both the email texts and their corresponding labels) into an 80% training set and a 20% testing set to allow for model training and unbiased evaluation.

3 Methodology

3.1 Data Mining Techniques Used

- **Classification**: The primary technique is Naive Bayes classification, specifically the Multinomial variant, suitable for text data based on word counts.
- **Text Preprocessing**: Essential techniques include tokenization (splitting text into words), lowercasing, and filtering (keeping only alphabetic tokens).
- **Feature Extraction**: A "bag-of-words" model is implicitly used, where features are the unique words in the training vocabulary, and their values are derived from word counts.
- **Feature Engineering/Selection**: A method based on word informativeness (log odds ratio) is used to select a subset of the most discriminative words, aiming to improve performance or efficiency.
- **Model Training**: Calculating prior probabilities and smoothed class conditional probabilities (likelihoods) from the training data.
- **Model Evaluation**: Using standard metrics like Precision, Recall, F1-Score, and Accuracy, often derived from a confusion matrix (calculating True Positives, False Positives, False Negatives, True Negatives).
- **Parameter Tuning**: Investigating the effect of the Laplace smoothing parameter(α/λ) on model performance.

3.2 Tools and Technologies

- **Programming Language**: Python 3.
- **Core Libraries**:
  - `pathlib`: For handling file system paths.
  - `re`: For regular expression-based tokenization.
  - `collections.defaultdict`: For efficient word counting during vocabulary creation and frequency analysis.
  - `math`: For logarithm calculations (log probabilities).
  - `pandas`: Used for organizing word counts and calculating probabilities into DataFrames, facilitating calculations like summing counts and deriving probabilities per word.
- **Machine Learning Libraries**:
  - `sklearn.model_selection`: Used for train_test_split.
  - `sklearn.metrics`: Crucial for evaluating the classifier using `accuracy_score`, `precision_score`, `recall_score`, `f1_score`, `confusion_matrix`, and `classification_report`.
- **Visualization**: `matplotlib.pyplot`: Used to generate bar charts visualizing the frequency distribution of top words in ham and spam emails.
- **Development Environment**: Jupyter Notebook (i.e., the `.ipynb` file provided).

3.3 Process Description

This section shows the sequential steps to implement, train, and evaluate the Naive Bayes spam filter. The process closely follows the specific tasks detailed in the *programming* *assignment* guidelines so that all required components are addressed systematically. This overall pipeline is visually summarized in Figure 1 below.

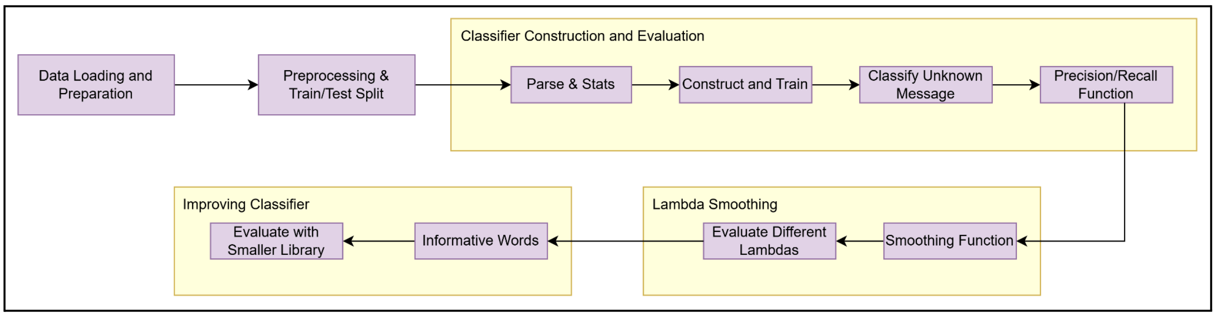

Data Loading and Preparation
Preprocessing & Train/Test Split
Classifier Construction and Evaluation
Parse & Stats
Construct and Train
Classify Unknown Message
Precision/Recall Function
Lambda Smoothing
Smoothing Function
Evaluate Different Lambdas
Improving Classifier
Informative Words
Evaluate with Smaller Library

Figure 1. Process flow of the Naive Bayes spam-filtering pipeline

1. **Data Loading and Preparation**: Email file paths and labels were loaded from the TREC06p dataset structure. Emails were read into memory, handling potential encoding errors. Ham and spam emails were separated based on labels.
2. **Preprocessing & Train/Test Split**: Emails were preprocessed using the `parse_documents` function (regex tokenization `[a-zA-Z]+`, lowercasing). The data was split into 80% for training and 20% for testing.
3. **Classifier Construction and Evaluation** (Assignment 2.1):
   - *(Task 1: Parse & Stats)* Vocabulary V was formed from unique words in the *training set* (`get_unique_tokens`). Word counts for ham (`ham_word_counts`) and spam (`spam_word_counts`) were computed using `defaultdict`. Prior probabilities (`prior_ham`, `prior_spam`) were calculated based on the ratio of ham/spam documents in the *training set*.
   - *(Task 2: Construct & Train)* A Naive Bayes Classifier model was constructed (using the `NaiveBayesClassifier` class). Training involved calculating the conditional probabilities $P(word \mid ham)$ and $P(word \mid spam)$ from the word counts and total words per class (initially without smoothing, later with). These probabilities were stored (e.g., in pandas DataFrames and dictionaries).
   - *(Task 3: Classify Unknown Message)* A prediction function (`predict` or `model.predict`) was implemented. It takes a tokenized unseen email, calculates the log posterior probability for 'ham' and 'spam' using the trained priors and conditional probabilities (Eq.4), and returns the class with the higher score. This was tested on the test set (`X_test`).
   - *(Task 4: Precision/Recall Function)* A function (`compute_precision_recall` or using `sklearn.metrics`) was implemented to calculate precision and recall based on the definitions:

$$\text{Precision} = \frac{TP}{TP + FP}, \qquad \text{Recall} = \frac{TP}{TP + FN}$$

     where TP = spam predicted as spam, FP = ham predicted as spam, FN = spam predicted as ham.
4. **Lambda Smoothing** (Assignment 2.2):

- *(Task 1: Smoothing Function)* A modified prediction function (`predict_with_laplace_smoothing` or integrated into `NaiveBayesClassifier`) was implemented. It calculates smoothed likelihoods using Eq.3 ($(count + \alpha)/(total\_words + \alpha|V|)$) for words present in the vocabulary and calculates a probability for unseen words as $\alpha/(total\_words + \alpha|V|)$.
- *(Task 2: Evaluate Different Lambdas)* The classifier was trained and evaluated (calculating precision and recall on the test set) using five different values for the smoothing parameter alpha (λ): 2.0, 1.0, 0.5, 0.1, 0.005. Results were printed to determine which value yielded the best precision/recall trade-off.

5. **Improving Classifier** (Assignment 2.3):
    - *(Task 1: Informative Words)* Using the best lambda found, an `informative_words` function was implemented. It calculated the log odds ratio

    $$\log \frac{P(word \mid spam)}{P(word \mid ham)}$$

    For each word in the vocabulary, using the smoothed probabilities. This approach, leveraging log-likelihood ratios to determine feature importance, is discussed by Hovold (2006) as a method for feature selection in spam filtering. A positive score indicates the word is more indicative of spam, while a negative score suggests it's more indicative of ham. The words were then sorted by the absolute value of this score to rank them by their informativeness, and the top 200 were identified. Subsequently, the top 10 words most indicative of spam (highest positive scores) and the top 10 most indicative of ham (most negative scores) were printed from this list.
    - *(Task 2: Evaluate with Smaller Vocabulary)* A modified prediction process (`predict_with_informative` or `predict_many_informative`) was used. This process filters the tokens of each incoming test email, considering only those words present in the previously identified set of top 200 informative words (Hovold, 2006). The classifier's decision is then based solely on this reduced feature set. The precision and recall were re-evaluated using this filtered vocabulary to assess the impact of focusing only on the most discriminative terms.

4 Results and Analysis

To evaluate the performance of the Naive Bayes spam classifier, several experiments were conducted using the TREC06p dataset, which was split into 80% for training and 20% for testing. The classifier was first tested with Laplace smoothing applied to mitigate zero-frequency issues.

A. Performance Metrics

The classifier was tested with Laplace smoothing on a test set comprising 20% of the TREC06p dataset. The results show that the model performs strongly across all major evaluation metrics. As presented in the table below, the classifier achieves a **94.01% accuracy**, **97.45% precision**, **93.34% recall**, and an **F1 score of 95.35%**. This demonstrates the model's reliability in identifying spam while keeping false positives low.

Metric	**Score**
Accuracy	0.9401
Precision	0.9745
Recall	0.9334
F1 Score	0.9535

Table 1: Performance Metrics of the Spam Detection Model

B. Effect of Different Lambda (Smoothing) Values

To analyze the effect of Laplace smoothing, the classifier was evaluated using various λ (α) values. Lower values of lambda provided better recall and precision, confirming the benefit of reduced smoothing in capturing actual word distributions. The best performing configuration was at λ = 0.005, which achieved the highest scores across both metrics.

Lambda (α)	Precision	Recall
2.0	0.9550	0.9052
1.0	0.9600	0.9119
0.5	0.9622	0.9167
0.1	0.9661	0.9294
0.005	**0.9697**	**0.9352**

Table 2: Precision and Recall at Different Lambda (α) Values

C. Informative Words (Log-Odds Ratio)

Feature selection based on the log-odds ratio allowed identification of the most discriminative words for each class. The top spam words included terms such as *Shopzilla*, *greylink*, and *ephedra*, which are typical of promotional or suspicious content. Conversely, top ham words like *lugnet* and *rpcss* suggest more technical or legitimate communication patterns.

Top Spam Words	Top Ham Words
shopzilla	lugnet
greylink	cert
hoodia	rpcss
ephedra	pdfzone
dgts	pnfs

Table 3: Top Discriminative Words in Spam and Ham Emails

D. Precision/Recall with Top 200 Informative Words

To test model efficiency using fewer features, only the top 200 informative words were retained for classification. This reduced feature set resulted in a **recall of 98.95%** and a **precision of 80.08%**, showing that while the model becomes more sensitive in detecting spam, it also becomes slightly more prone to misclassifying ham emails.

Metric	Score (Top 200 Informative Words)
Precision	0.8008
Recall	0.9895

Table 4: Classifier Performance Using Top 200 Informative Words

These findings confirm that the classifier is adaptable and performs reliably even under reduced feature dimensions, though a trade-off exists between recall and precision.

E. Visualization of Word Distributions

To further understand class-specific language, bar charts were created to show the top 20 most common words in each class:

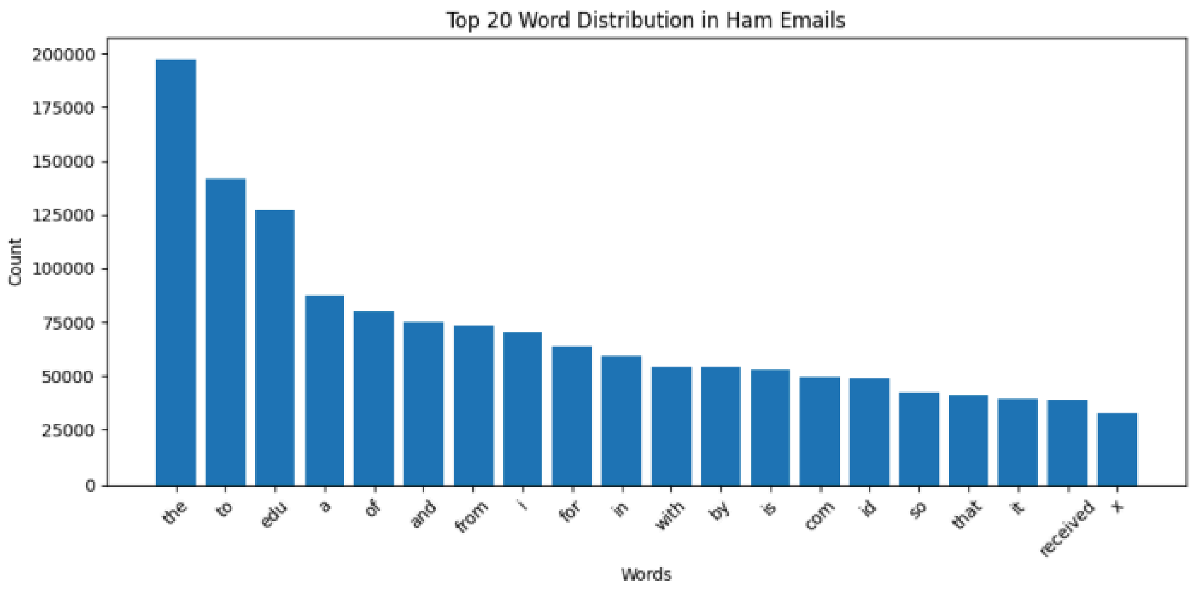


Figure 2. Frequency Distribution of the Top 20 Words in Ham Emails

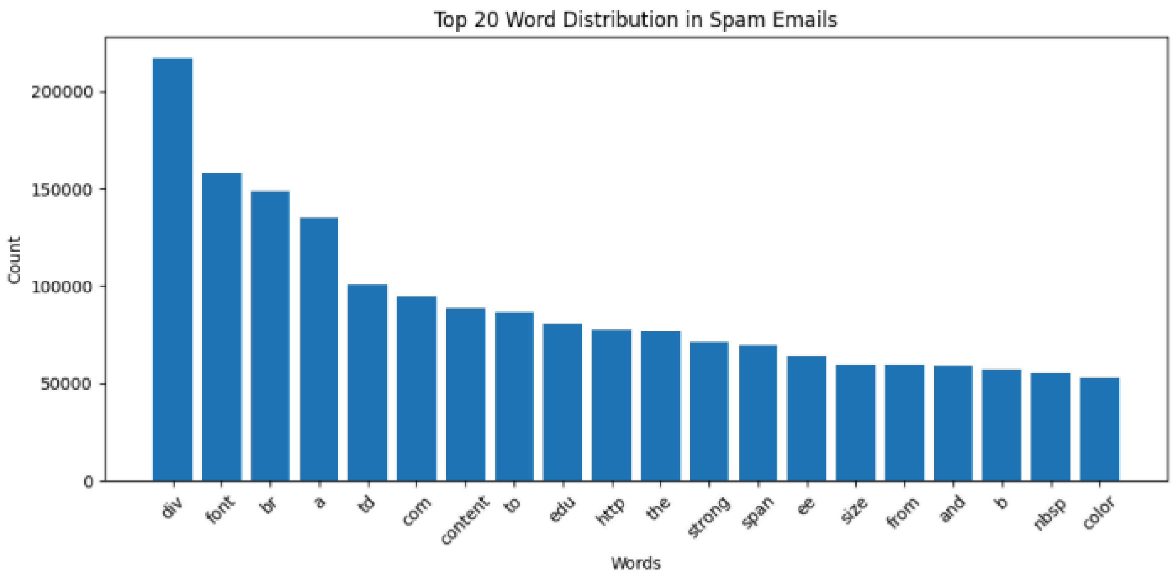


Figure 3. Frequency Distribution of the Top 20 Words in Spam Emails

Ham words are typical of conversational English (e.g., "the," "your"), while spam words include marketing triggers like "click," "free," and "offer."

Interpretation of Findings

The results underscore the effectiveness of the Multinomial Naive Bayes classifier in distinguishing between legitimate and spam emails. Its success lies in modeling word frequencies and leveraging prior class probabilities. The smoothing parameter, though often treated as a technicality, significantly influences performance. Smaller values of λ preserve the original word distribution, leading to more accurate results by reducing over-smoothing. The visual analyses reveal how spam relies on a different vocabulary and format (notably, HTML tokens) compared to ham, which is primarily composed of standard linguistic constructs.

Additionally, a feature selection strategy based on **log odds ratio** was employed to extract the 200 most informative words. Using these alone for classification yielded a **recall of 98.95%** and a **precision of 80.08%**, suggesting a stronger focus on correctly detecting spam but at the cost of more false positives. This shows promise for constrained-resource scenarios where high recall is prioritized.

5 Conclusions and Recommendations

5.1 Summary of Key Findings

This study validated the effectiveness of the Naive Bayes classifier in spam filtering, achieving a strong performance with an accuracy of 94.01%. The best results were observed with a Laplace smoothing parameter (λ) of 0.005, which provided an optimal balance between generalization and model sensitivity. Feature selection using log-odds ratios allowed the identification of highly predictive vocabulary for both spam and ham classes. Even with basic preprocessing and a simple bag-of-words model, the classifier achieved a precision of 97.45% and a recall of 93.34%, demonstrating its practical value and efficiency for real-world applications.

5.2 Recommendations

To further enhance the model's performance and practical deployment across various contexts, several strategic improvements are suggested. First, applying minimal Laplace smoothing **(e.g., λ = 0.005)** is recommended to strike an effective balance between generalization and sensitivity, reducing the risk of underestimating rare but meaningful features. Preprocessing enhancements such as removing stopwords, stemming, or lemmatization can significantly reduce vocabulary noise, leading to cleaner, more accurate representations of the email text.

Additionally, enriching feature inputs by incorporating metadata (e.g., sender domain, subject line, timestamps) provides valuable context that can help disambiguate borderline classifications. Finally, to increase accuracy and handle more complex spam patterns, integrating Naive Bayes into ensemble models or hybrid approaches with more sophisticated classifiers such as Support Vector Machines or Random Forests could result in improved generalizability and accuracy, especially in real-world spam filtering scenarios.

5.3 Limitations

Despite the strong performance of the Naive Bayes classifier, several critical limitations must be acknowledged. First, the model is built on the assumption of conditional independence between words, an oversimplification that does not accurately capture the complexities of natural language. This can hinder performance in detecting nuanced or context-dependent spam messages. Hence, the use of a bag-of-words representation means that the model ignores word order, syntax, and semantic meaning factors, which can carry important context for proper classification. Another limitation is that the evaluation and training were conducted exclusively on the TREC06p dataset. As a result, the classifier's effectiveness may not generalize well to newer or more diverse datasets, particularly those with evolving spam structures or multilingual content.

5.4 Suggestions for Future Studies

Future research could build on the strengths of this study by incorporating more advanced and adaptable methodologies. One promising direction involves the use of context-aware models such as Bidirectional Encoder Representations from Transformers, Long Short-Term Memory, or Transformer-based architectures, which are capable of understanding both the sequential and semantic relationships between words. These models could offer a more nuanced understanding of email content and improve classification accuracy.

Moreover, developing adaptive or incremental learning frameworks would allow the classifier to evolve alongside changing spam tactics, ensuring long-term effectiveness. Another key opportunity lies in extending the model's evaluation to include multilingual datasets, thereby enhancing its global applicability and inclusivity. Finally, deploying the system in real-time using streaming data would test its scalability, responsiveness, and practical utility in live environments such as enterprise email systems or cloud-based filters.

6 Appendices

This section provides detailed supplementary material to support reproducibility, clarity, and transparency in the implementation and evaluation of the Naive Bayes spam classifier.

A. Word Frequency Visualizations

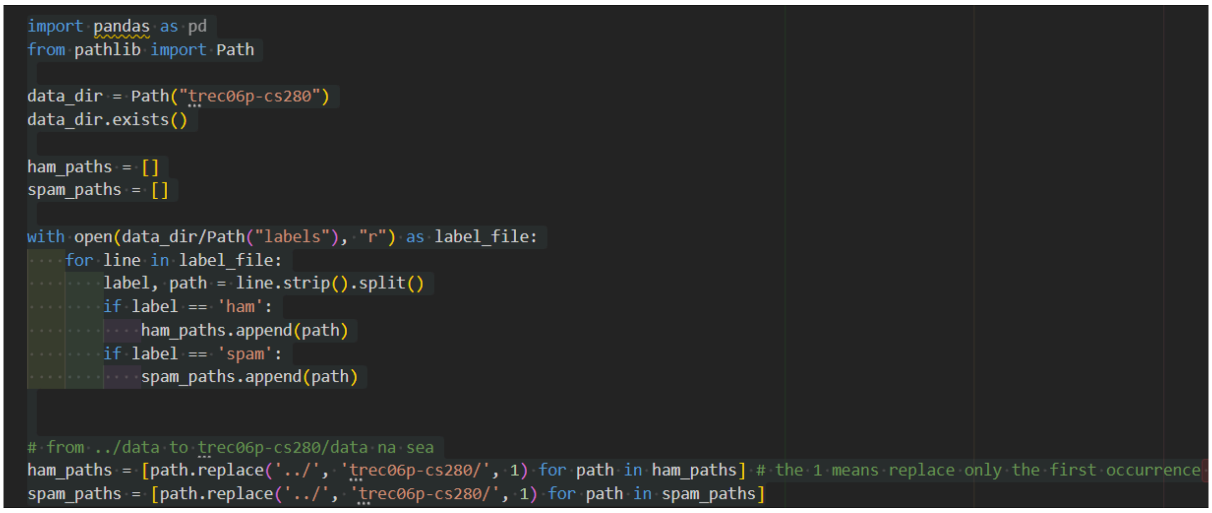
The following charts represent the top 20 most frequent words found in the TREC06p dataset's ham and spam emails. These frequencies illustrate the lexical distinctions between legitimate messages and spam:

- **Ham Word Frequency Chart:** Shows the prevalence of common English stop words (e.g., "the," "your," "to"), suggesting conversational tone.
- **Spam Word Frequency Chart:** Dominated by HTML tags (e.g., "div," "font") and promotional terms (e.g., "free," "click," "offer").

B. Key Implementation Code Snippets

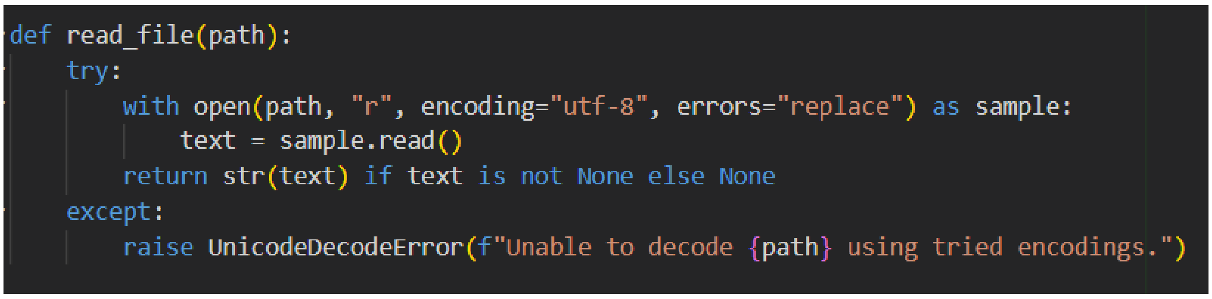
These code excerpts illustrate the end-to-end implementation of the Naive Bayes classifier:

Appendix 1. Implementation of Email Classification Preprocessing in Python



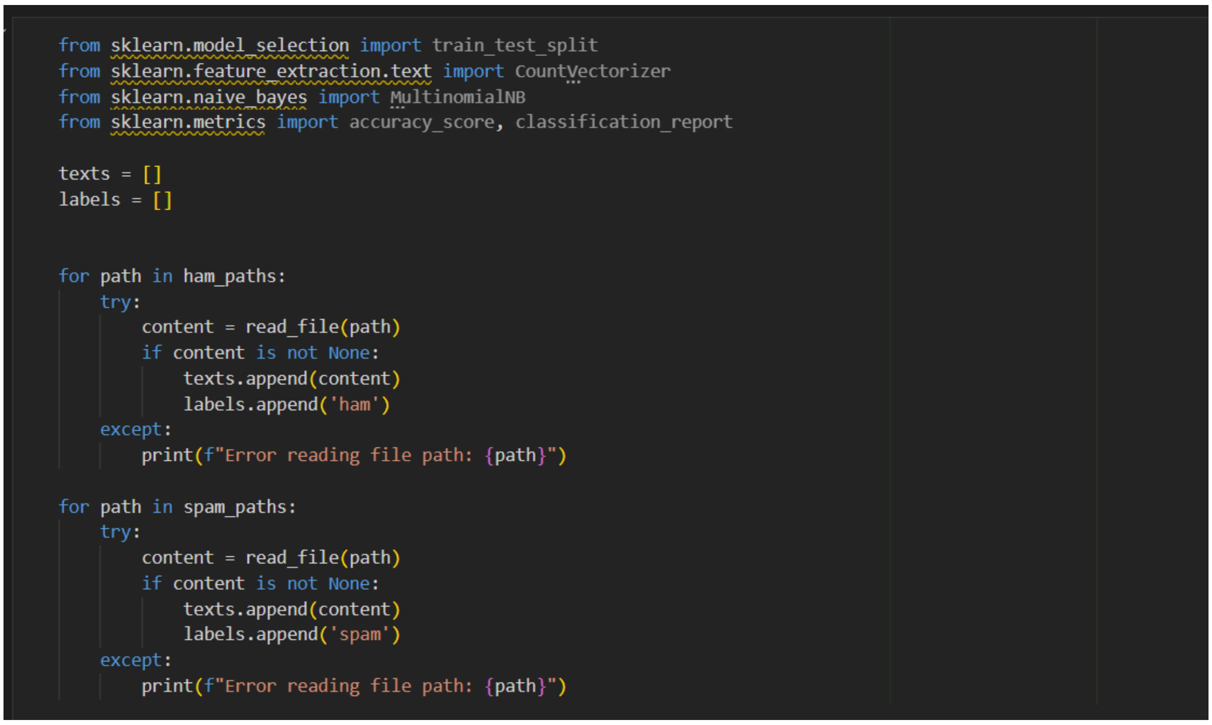
```python
import pandas as pd
from pathlib import Path

data_dir = Path("trec06p-cs280")
data_dir.exists()

ham_paths = []
spam_paths = []

with open(data_dir/Path("labels"), "r") as label_file:
    for line in label_file:
        label, path = line.strip().split()
        if label == 'ham':
            ham_paths.append(path)
        if label == 'spam':
            spam_paths.append(path)


# from ../data to trec06p-cs280/data na sea
ham_paths = [path.replace('../', 'trec06p-cs280/', 1) for path in ham_paths] # the 1 means replace only the first occurrence
spam_paths = [path.replace('../', 'trec06p-cs280/', 1) for path in spam_paths]
```

Appendix 2. Custom `read_file` Function for Handling File Encoding Errors Gracefully



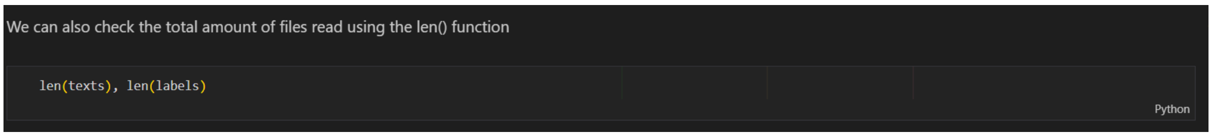
```python
def read_file(path):
    try:
        with open(path, "r", encoding="utf-8", errors="replace") as sample:
            text = sample.read()
        return str(text) if text is not None else None
    except:
        raise UnicodeDecodeError(f"Unable to decode {path} using tried encodings.")
```

Appendix 3. Loading and Labeling Email Texts from Ham and Spam Paths for Classification



```python
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report

texts = []
labels = []


for path in ham_paths:
    try:
        content = read_file(path)
        if content is not None:
            texts.append(content)
            labels.append('ham')
    except:
        print(f"Error reading file path: {path}")

for path in spam_paths:
    try:
        content = read_file(path)
        if content is not None:
            texts.append(content)
            labels.append('spam')
    except:
        print(f"Error reading file path: {path}")
```

Appendix 4. Verifying the Number of Email Files Read with `len()`

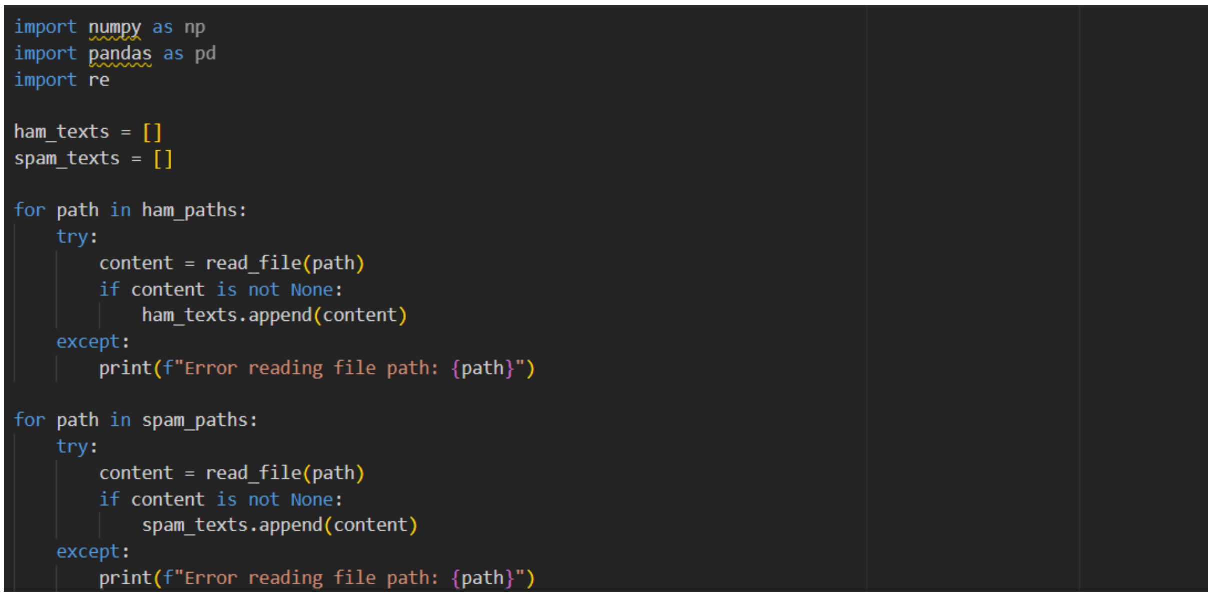


We can also check the total amount of files read using the len() function

```python
len(texts), len(labels)
```

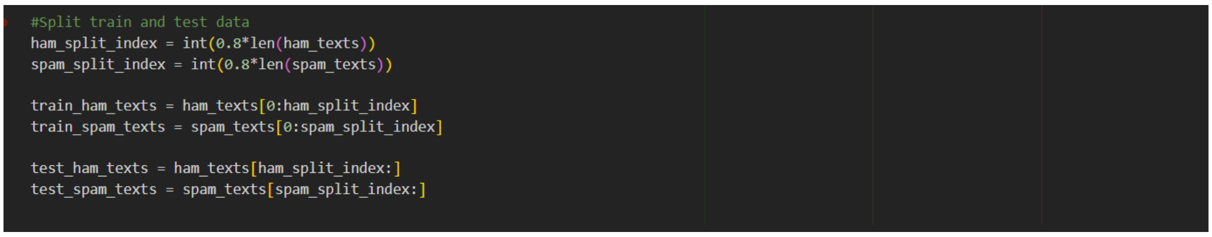
Python

Appendix 5. Separating and Storing Raw Email Texts for Spam and Ham Categories



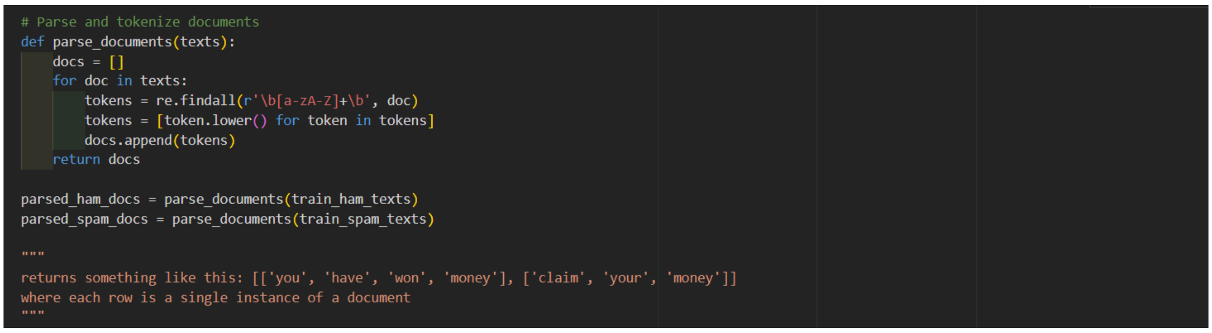
```python
import numpy as np
import pandas as pd
import re

ham_texts = []
spam_texts = []

for path in ham_paths:
    try:
        content = read_file(path)
        if content is not None:
            ham_texts.append(content)
    except:
        print(f"Error reading file path: {path}")

for path in spam_paths:
    try:
        content = read_file(path)
        if content is not None:
            spam_texts.append(content)
    except:
        print(f"Error reading file path: {path}")
```

Appendix 6. Manual Train-Test Split for Spam and Ham Email Datasets (80/20 Ratio)



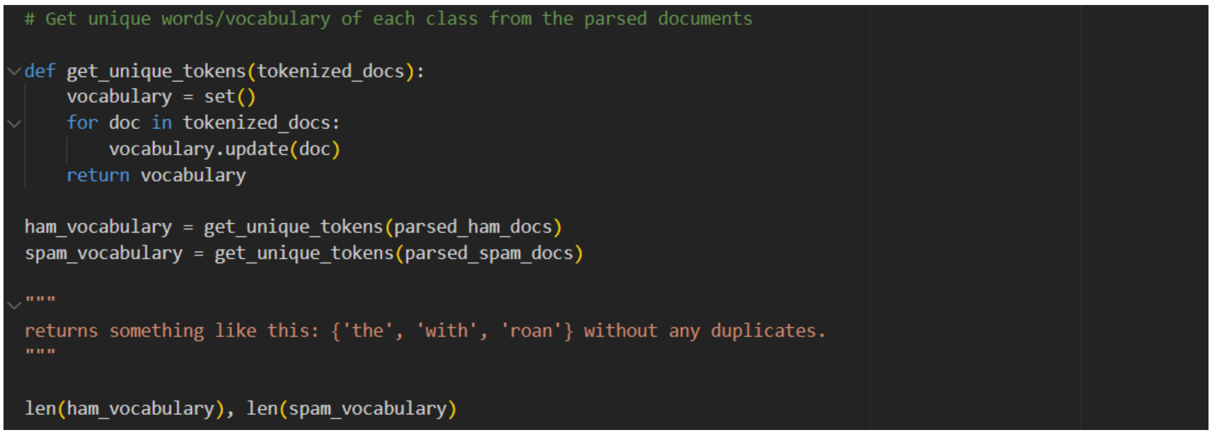
```python
#Split train and test data
ham_split_index = int(0.8*len(ham_texts))
spam_split_index = int(0.8*len(spam_texts))

train_ham_texts = ham_texts[0:ham_split_index]
train_spam_texts = spam_texts[0:spam_split_index]

test_ham_texts = ham_texts[ham_split_index:]
test_spam_texts = spam_texts[spam_split_index:]
```

Appendix 7. Function to Tokenize Email Texts into Lowercased Word Lists Using Regex



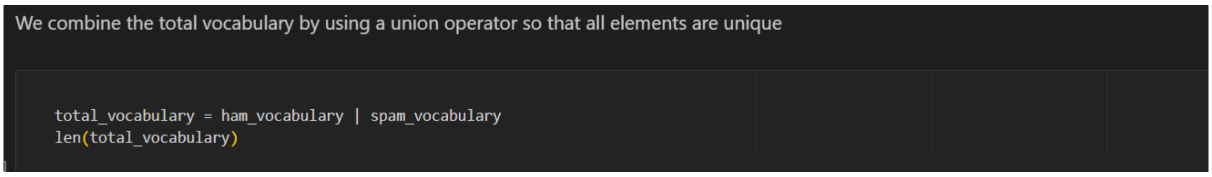
```python
# Parse and tokenize documents
def parse_documents(texts):
    docs = []
    for doc in texts:
        tokens = re.findall(r'\b[a-zA-Z]+\b', doc)
        tokens = [token.lower() for token in tokens]
        docs.append(tokens)
    return docs

parsed_ham_docs = parse_documents(train_ham_texts)
parsed_spam_docs = parse_documents(train_spam_texts)

"""
returns something like this: [['you', 'have', 'won', 'money'], ['claim', 'your', 'money']]
where each row is a single instance of a document
"""
```

Appendix 8. Extracting Unique Word Tokens from Parsed Ham and Spam Documents



```python
# Get unique words/vocabulary of each class from the parsed documents

def get_unique_tokens(tokenized_docs):
    vocabulary = set()
    for doc in tokenized_docs:
        vocabulary.update(doc)
    return vocabulary

ham_vocabulary = get_unique_tokens(parsed_ham_docs)
spam_vocabulary = get_unique_tokens(parsed_spam_docs)

"""
returns something like this: {'the', 'with', 'roan'} without any duplicates.
"""

len(ham_vocabulary), len(spam_vocabulary)
```

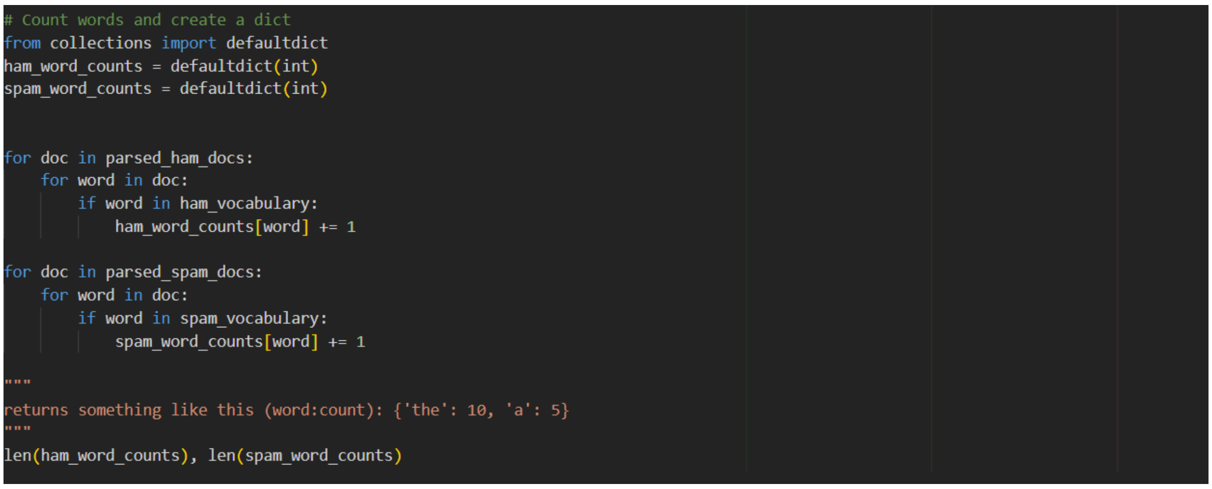
Appendix 9. Combining Ham and Spam Vocabularies Using Set Union for Unique Word Collection



We combine the total vocabulary by using a union operator so that all elements are unique

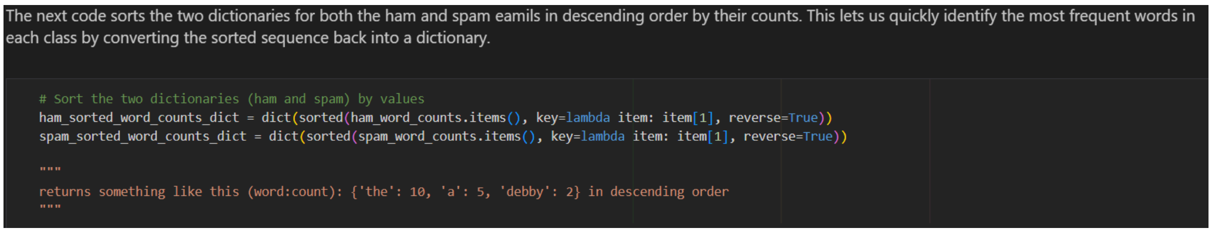
```python
total_vocabulary = ham_vocabulary | spam_vocabulary
len(total_vocabulary)
```

Appendix 10. Counting Word Frequencies in Parsed Ham and Spam Documents Using defaultdict



```python
# Count words and create a dict
from collections import defaultdict
ham_word_counts = defaultdict(int)
spam_word_counts = defaultdict(int)


for doc in parsed_ham_docs:
    for word in doc:
        if word in ham_vocabulary:
            ham_word_counts[word] += 1

for doc in parsed_spam_docs:
    for word in doc:
        if word in spam_vocabulary:
            spam_word_counts[word] += 1

"""
returns something like this (word:count): {'the': 10, 'a': 5}
"""
len(ham_word_counts), len(spam_word_counts)
```

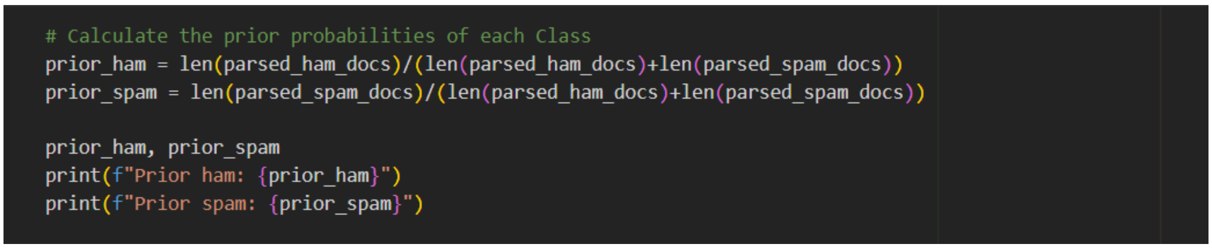
Appendix 11. Sorting Ham and Spam Word Count Dictionaries by Frequency in Descending Order



The next code sorts the two dictionaries for both the ham and spam eamils in descending order by their counts. This lets us quickly identify the most frequent words in each class by converting the sorted sequence back into a dictionary.

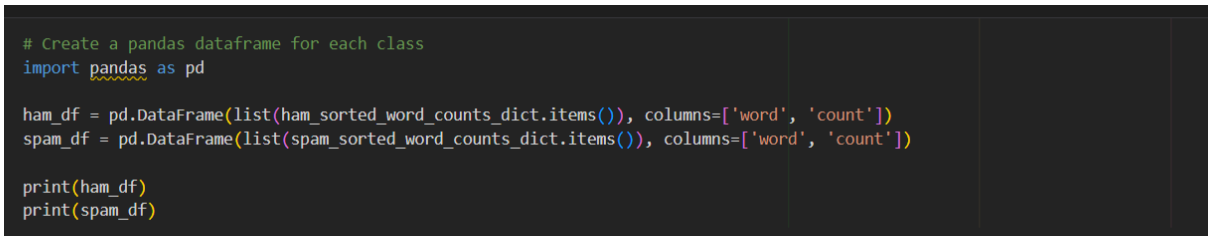
```
# Sort the two dictionaries (ham and spam) by values
ham_sorted_word_counts_dict = dict(sorted(ham_word_counts.items(), key=lambda item: item[1], reverse=True))
spam_sorted_word_counts_dict = dict(sorted(spam_word_counts.items(), key=lambda item: item[1], reverse=True))

"""
returns something like this (word:count): {'the': 10, 'a': 5, 'debby': 2} in descending order
"""
```

Appendix 12. Calculating Prior Probabilities for Ham and Spam Classes in a Naive Bayes Classifier



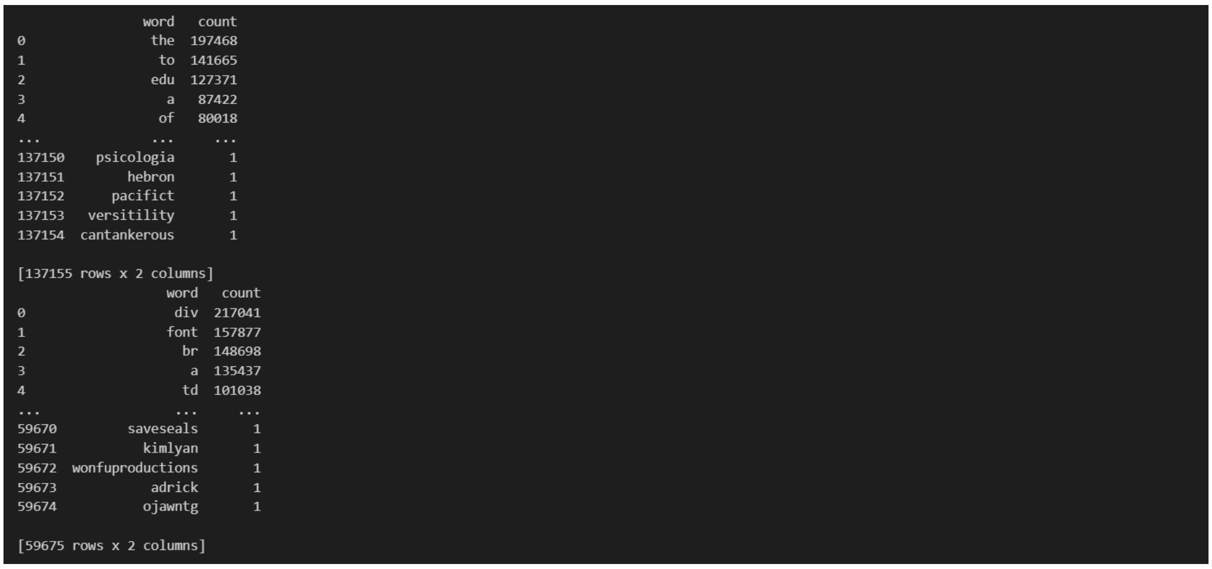
```
# Calculate the prior probabilities of each Class
prior_ham = len(parsed_ham_docs)/(len(parsed_ham_docs)+len(parsed_spam_docs))
prior_spam = len(parsed_spam_docs)/(len(parsed_ham_docs)+len(parsed_spam_docs))

prior_ham, prior_spam
print(f"Prior ham: {prior_ham}")
print(f"Prior spam: {prior_spam}")
```

Appendix 13. Converting Sorted Word Count Dictionaries into Pandas DataFrames for Ham and Spam Classes



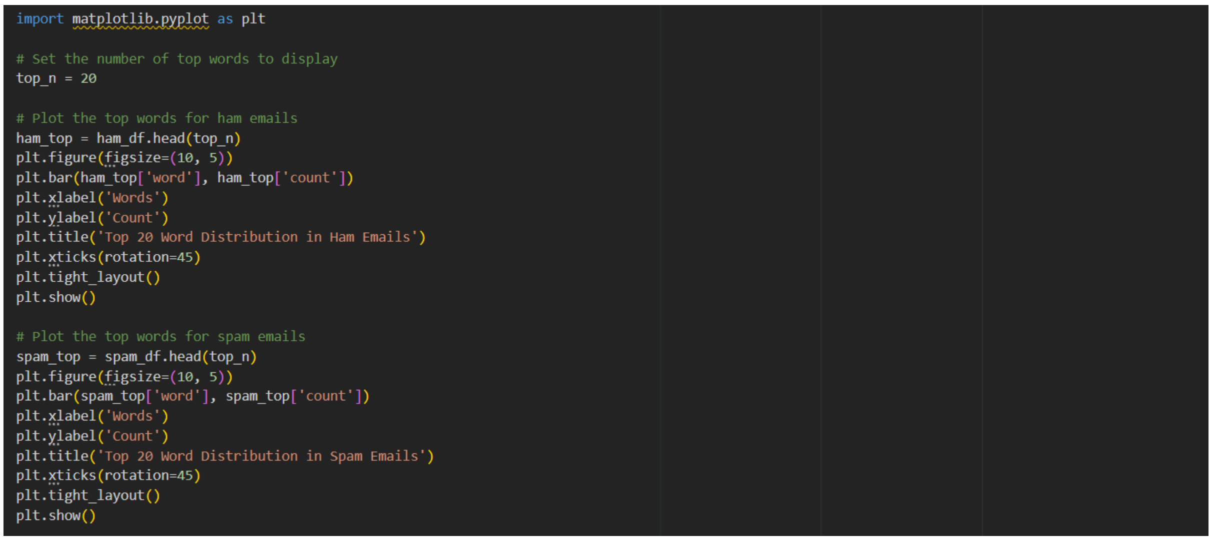
```
# Create a pandas dataframe for each class
import pandas as pd

ham_df = pd.DataFrame(list(ham_sorted_word_counts_dict.items()), columns=['word', 'count'])
spam_df = pd.DataFrame(list(spam_sorted_word_counts_dict.items()), columns=['word', 'count'])

print(ham_df)
print(spam_df)
```

Appendix 14. Sample Output of Ham and Spam Word Frequency Tables with Total Rows and Sorted Counts

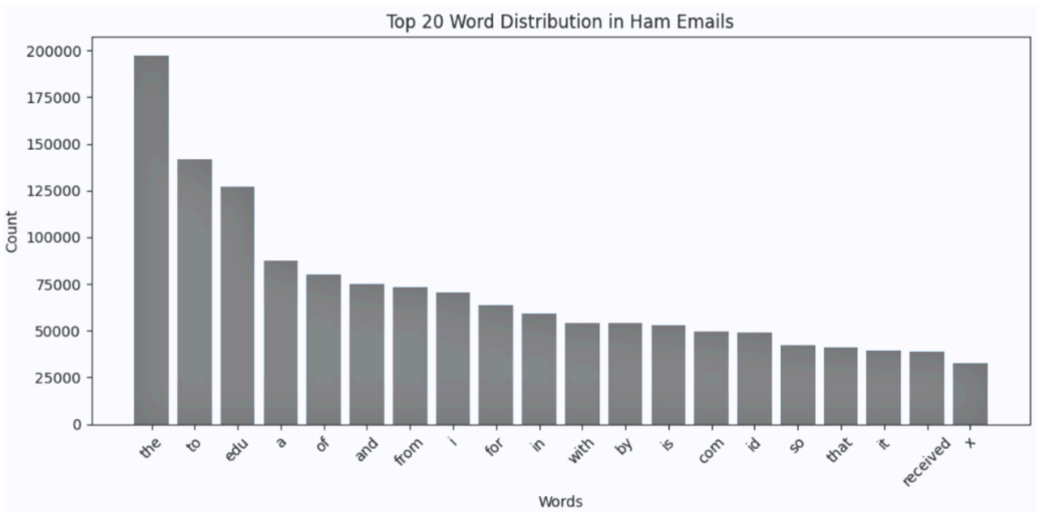


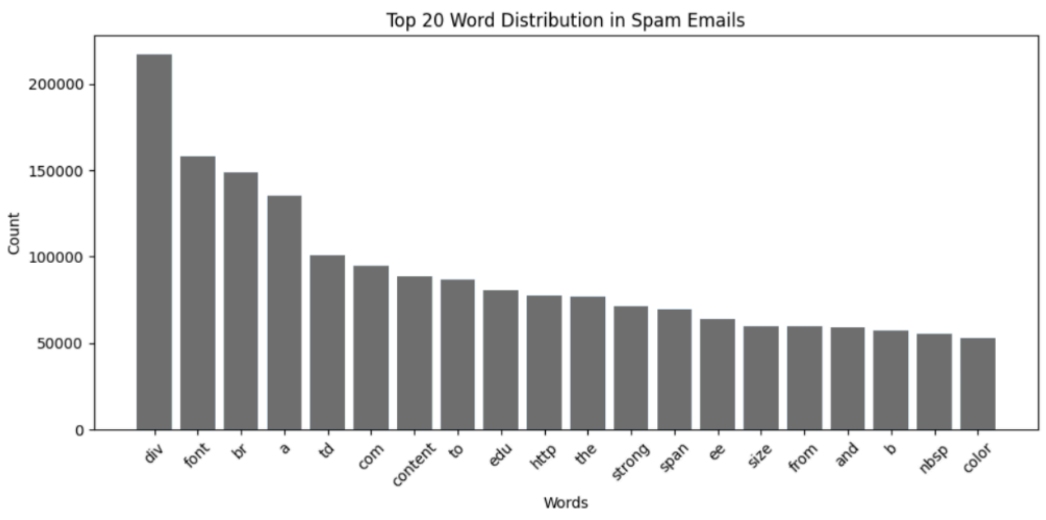
```
                word   count
0                the  197468
1                 to  141665
2                edu  127371
3                  a   87422
4                 of   80018
...              ...     ...
137150    psicologia       1
137151        hebron       1
137152      pacifict       1
137153   versitility       1
137154  cantankerous       1

[137155 rows x 2 columns]
                   word   count
0                   div  217041
1                  font  157877
2                    br  148698
3                     a  135437
4                    td  101038
...                 ...     ...
59670         saveseals       1
59671           kimlyan       1
59672  wonfuproductions       1
59673            adrick       1
59674           ojawntg       1

[59675 rows x 2 columns]
```

Appendix 15. Bar Plots of the Top 20 Most Frequent Words in Ham and Spam Emails Using Matplotlib

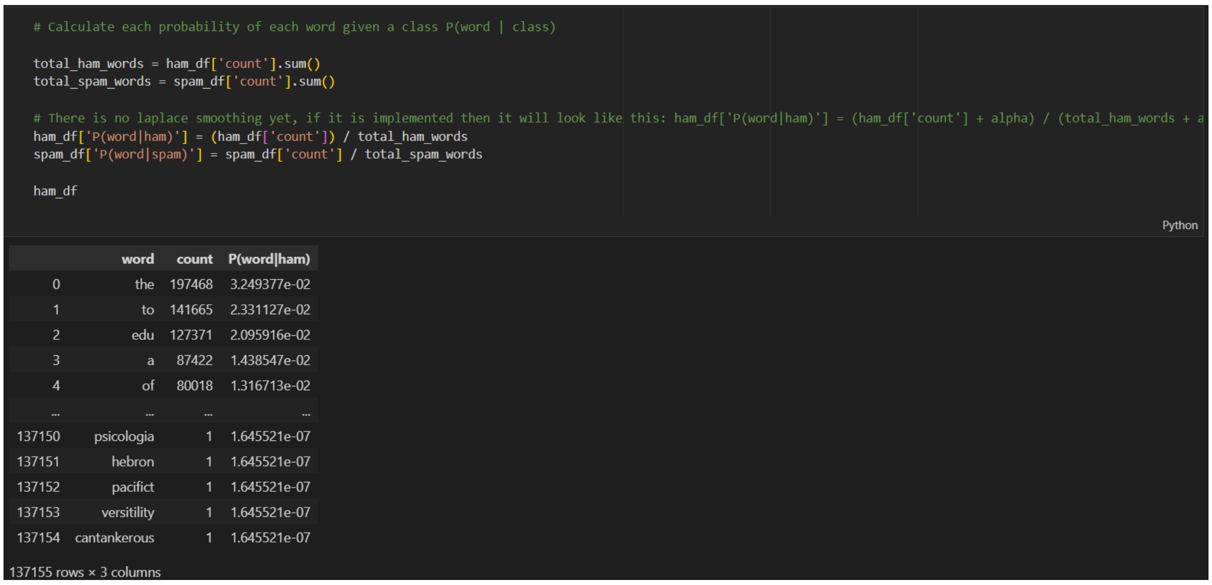


```python
import matplotlib.pyplot as plt

# Set the number of top words to display
top_n = 20

# Plot the top words for ham emails
ham_top = ham_df.head(top_n)
plt.figure(figsize=(10, 5))
plt.bar(ham_top['word'], ham_top['count'])
plt.xlabel('Words')
plt.ylabel('Count')
plt.title('Top 20 Word Distribution in Ham Emails')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

# Plot the top words for spam emails
spam_top = spam_df.head(top_n)
plt.figure(figsize=(10, 5))
plt.bar(spam_top['word'], spam_top['count'])
plt.xlabel('Words')
plt.ylabel('Count')
plt.title('Top 20 Word Distribution in Spam Emails')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```





Appendix 16. Calculating Conditional Probabilities P(word|ham) for Each Token Based on Frequency Counts

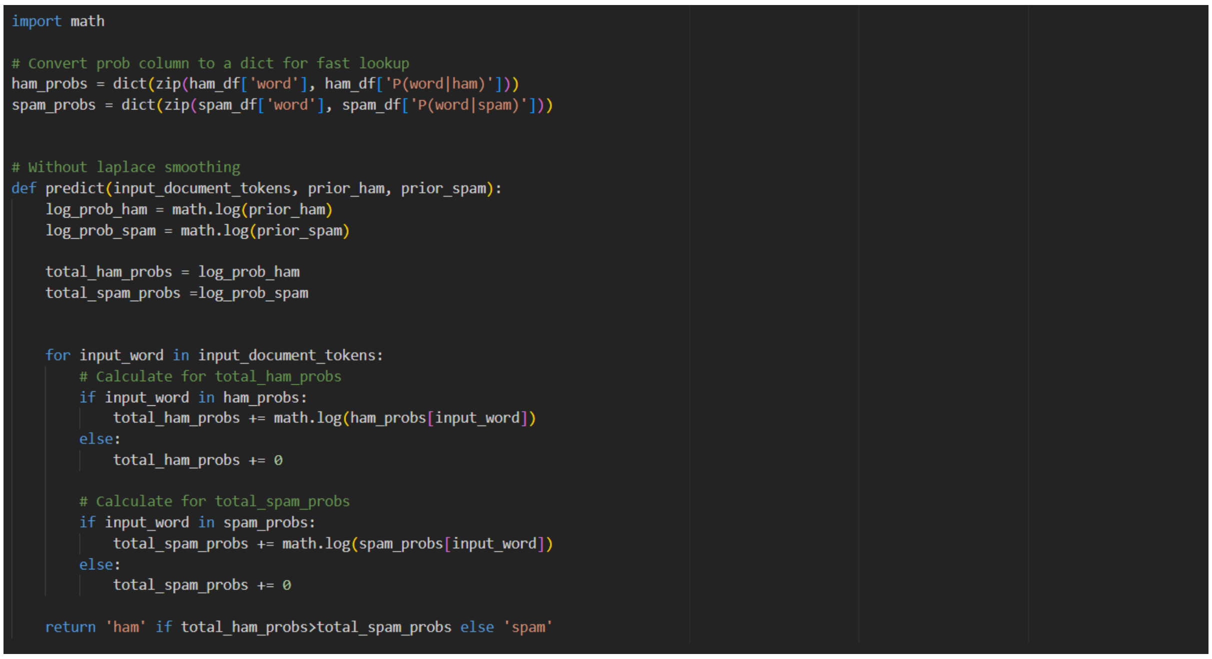


```python
# Calculate each probability of each word given a class P(word | class)

total_ham_words = ham_df['count'].sum()
total_spam_words = spam_df['count'].sum()

# There is no laplace smoothing yet, if it is implemented then it will look like this: ham_df['P(word|ham)'] = (ham_df['count'] + alpha) / (total_ham_words + a
ham_df['P(word|ham)'] = (ham_df['count']) / total_ham_words
spam_df['P(word|spam)'] = spam_df['count'] / total_spam_words

ham_df
```

Python

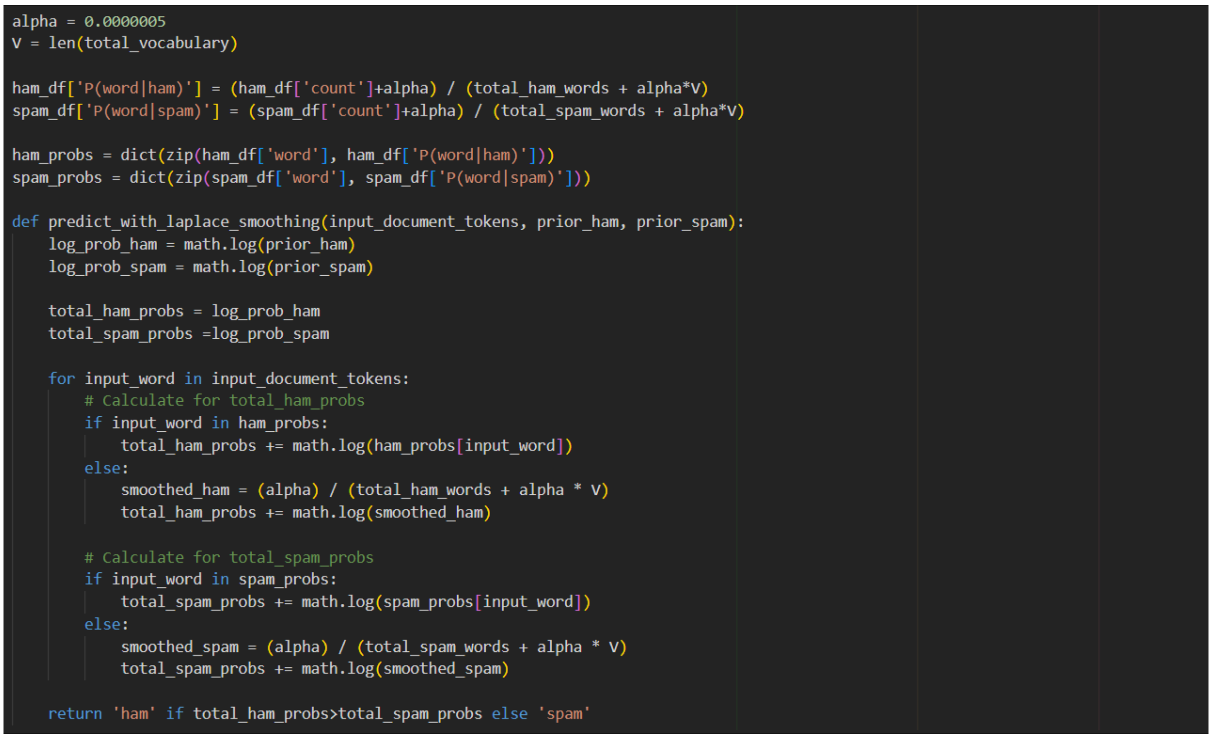
```
               word   count   P(word|ham)
     0          the  197468  3.249377e-02
     1           to  141665  2.331127e-02
     2          edu  127371  2.095916e-02
     3            a   87422  1.438547e-02
     4           of   80018  1.316713e-02
   ...          ...     ...           ...
137150   psicologia       1  1.645521e-07
137151       hebron       1  1.645521e-07
137152      pacifict       1  1.645521e-07
137153   versitility       1  1.645521e-07
137154  cantankerous      1  1.645521e-07

137155 rows × 3 columns
```

Appendix 17. Prediction Function for Classifying Emails as Ham or Spam Using Naive Bayes Without Smoothing



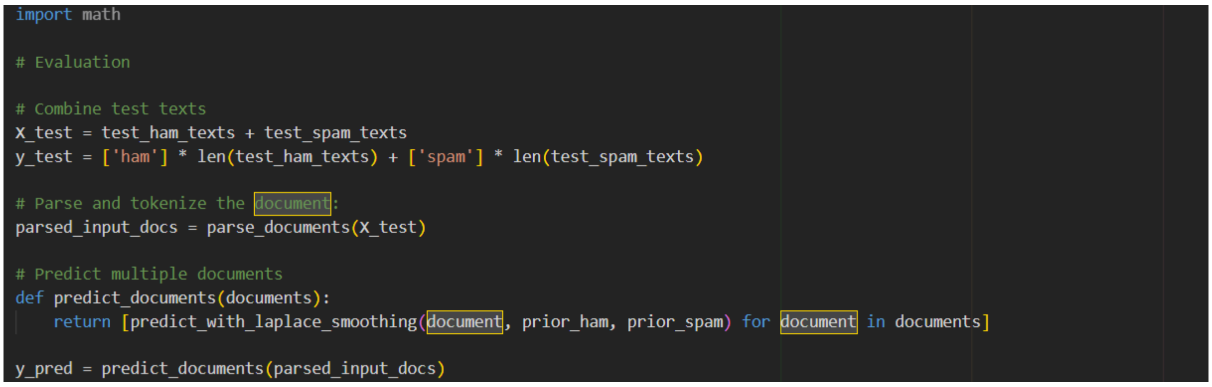
```python
import math

# Convert prob column to a dict for fast lookup
ham_probs = dict(zip(ham_df['word'], ham_df['P(word|ham)']))
spam_probs = dict(zip(spam_df['word'], spam_df['P(word|spam)']))


# Without laplace smoothing
def predict(input_document_tokens, prior_ham, prior_spam):
    log_prob_ham = math.log(prior_ham)
    log_prob_spam = math.log(prior_spam)

    total_ham_probs = log_prob_ham
    total_spam_probs =log_prob_spam


    for input_word in input_document_tokens:
        # Calculate for total_ham_probs
        if input_word in ham_probs:
            total_ham_probs += math.log(ham_probs[input_word])
        else:
            total_ham_probs += 0

        # Calculate for total_spam_probs
        if input_word in spam_probs:
            total_spam_probs += math.log(spam_probs[input_word])
        else:
            total_spam_probs += 0

    return 'ham' if total_ham_probs>total_spam_probs else 'spam'
```

Appendix 18. Implementing Laplace Smoothing in Naive Bayes Classifier to Handle Unseen Words



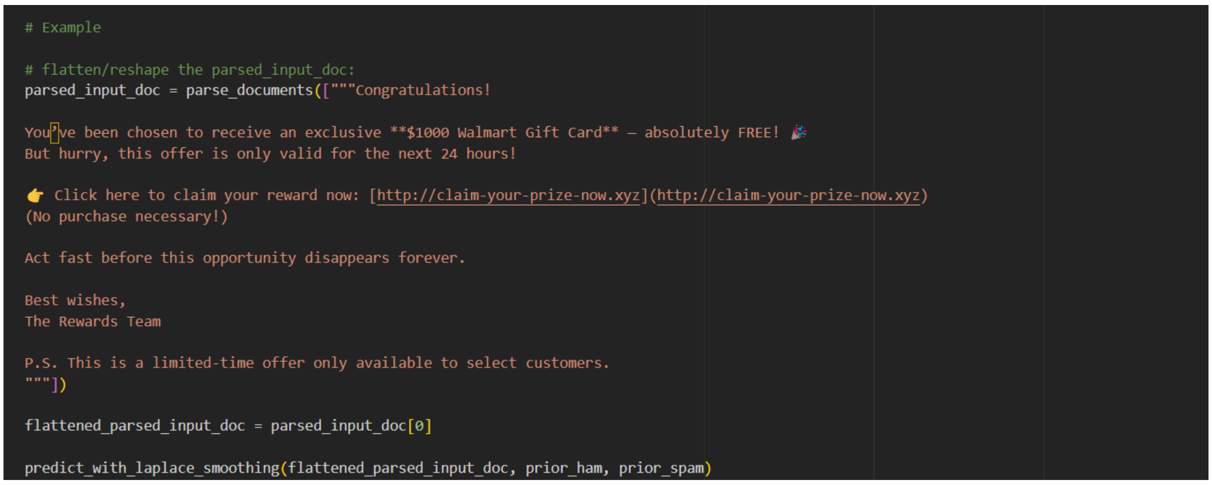
```python
alpha = 0.0000005
V = len(total_vocabulary)

ham_df['P(word|ham)'] = (ham_df['count']+alpha) / (total_ham_words + alpha*V)
spam_df['P(word|spam)'] = (spam_df['count']+alpha) / (total_spam_words + alpha*V)

ham_probs = dict(zip(ham_df['word'], ham_df['P(word|ham)']))
spam_probs = dict(zip(spam_df['word'], spam_df['P(word|spam)']))

def predict_with_laplace_smoothing(input_document_tokens, prior_ham, prior_spam):
    log_prob_ham = math.log(prior_ham)
    log_prob_spam = math.log(prior_spam)

    total_ham_probs = log_prob_ham
    total_spam_probs =log_prob_spam

    for input_word in input_document_tokens:
        # Calculate for total_ham_probs
        if input_word in ham_probs:
            total_ham_probs += math.log(ham_probs[input_word])
        else:
            smoothed_ham = (alpha) / (total_ham_words + alpha * V)
            total_ham_probs += math.log(smoothed_ham)

        # Calculate for total_spam_probs
        if input_word in spam_probs:
            total_spam_probs += math.log(spam_probs[input_word])
        else:
            smoothed_spam = (alpha) / (total_spam_words + alpha * V)
            total_spam_probs += math.log(smoothed_spam)

    return 'ham' if total_ham_probs>total_spam_probs else 'spam'
```

Appendix 19. Evaluating the Naive Bayes Classifier on Test Data Using Tokenized Inputs and Batch Prediction



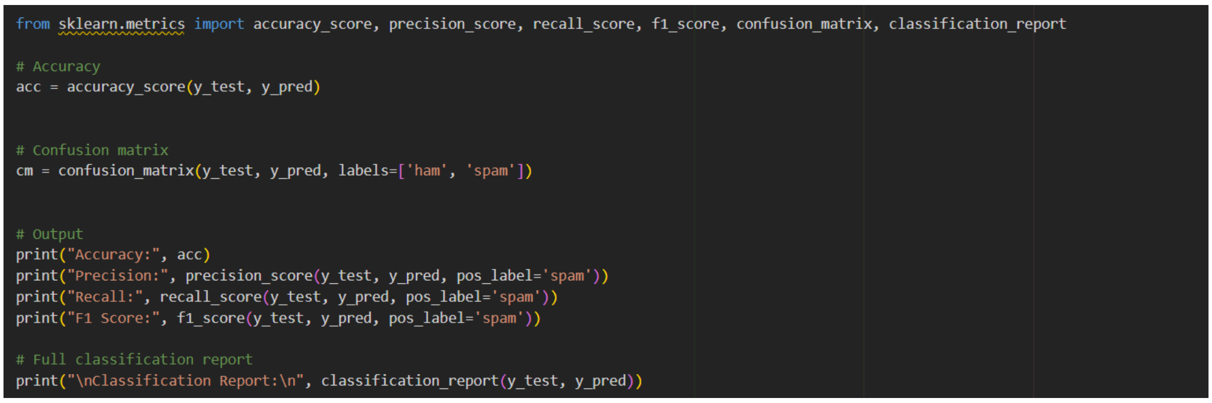
```python
import math

# Evaluation

# Combine test texts
X_test = test_ham_texts + test_spam_texts
y_test = ['ham'] * len(test_ham_texts) + ['spam'] * len(test_spam_texts)

# Parse and tokenize the document:
parsed_input_docs = parse_documents(X_test)

# Predict multiple documents
def predict_documents(documents):
    return [predict_with_laplace_smoothing(document, prior_ham, prior_spam) for document in documents]

y_pred = predict_documents(parsed_input_docs)
```

Appendix 20. Running a Spam Classification Example Using Parsed and Tokenized Input Text



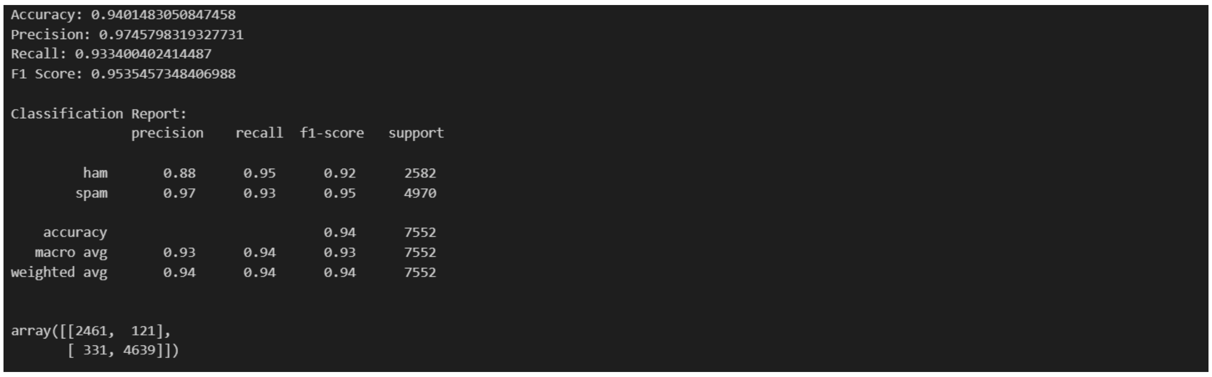
```python
# Example

# flatten/reshape the parsed_input_doc:
parsed_input_doc = parse_documents(["""Congratulations!

You’ve been chosen to receive an exclusive **$1000 Walmart Gift Card** — absolutely FREE! 🎉
But hurry, this offer is only valid for the next 24 hours!

👉 Click here to claim your reward now: [http://claim-your-prize-now.xyz](http://claim-your-prize-now.xyz)
(No purchase necessary!)

Act fast before this opportunity disappears forever.

Best wishes,
The Rewards Team

P.S. This is a limited-time offer only available to select customers.
"""])

flattened_parsed_input_doc = parsed_input_doc[0]

predict_with_laplace_smoothing(flattened_parsed_input_doc, prior_ham, prior_spam)
```

Appendix 21. Evaluating Model Performance Using Accuracy, Precision, Recall, F1 Score, and a Full Classification Report



```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, classification_report

# Accuracy
acc = accuracy_score(y_test, y_pred)


# Confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=['ham', 'spam'])


# Output
print("Accuracy:", acc)
print("Precision:", precision_score(y_test, y_pred, pos_label='spam'))
print("Recall:", recall_score(y_test, y_pred, pos_label='spam'))
print("F1 Score:", f1_score(y_test, y_pred, pos_label='spam'))

# Full classification report
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

Appendix 22. Model Evaluation Metrics and Confusion Matrix for Laplace-Smoothed Naive Bayes Spam Classifier

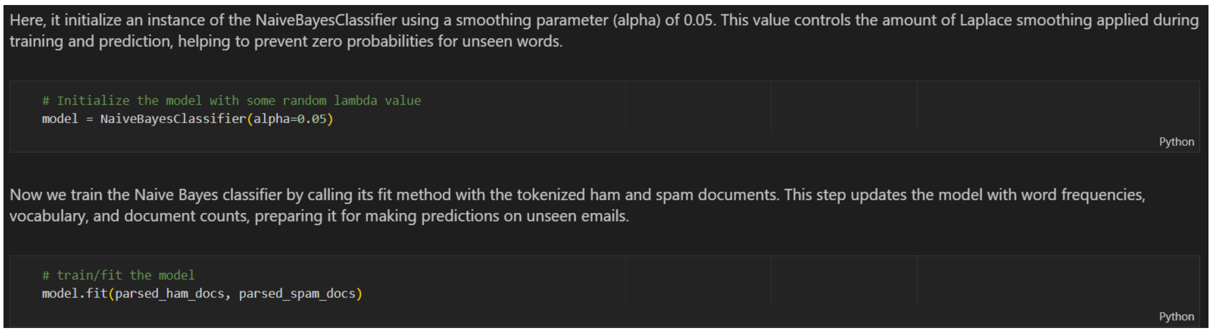


```
Accuracy: 0.9401483050847458
Precision: 0.9745798319327731
Recall: 0.933400402414487
F1 Score: 0.9535457348406988

Classification Report:
               precision    recall  f1-score   support

         ham       0.88      0.95      0.92      2582
        spam       0.97      0.93      0.95      4970

    accuracy                           0.94      7552
   macro avg       0.93      0.94      0.93      7552
weighted avg       0.94      0.94      0.94      7552


array([[2461,  121],
       [ 331, 4639]])
```

Appendix 23. Refactored Naive Bayes Classifier with Laplace Smoothing and Compact Predictive Logic for Spam Detection

```python
import math

from collections import defaultdict

class NaiveBayesClassifier:

    def __init__(self, alpha=1.0):

        self.alpha = alpha

        self.ham_counts = defaultdict(int)

        self.spam_counts = defaultdict(int)

        self.ham_docs = self.spam_docs = 0

        self.vocab = set()

        self.fitted = False

    def fit(self, ham_docs, spam_docs):

        self.ham_docs, self.spam_docs = len(ham_docs), len(spam_docs)

        for doc in ham_docs:

            for word in doc:

                self.ham_counts[word] += 1

                self.vocab.add(word)

        for doc in spam_docs:

            for word in doc:
                self.spam_counts[word] += 1
                self.vocab.add(word)
        self.total_ham = sum(self.ham_counts.values())
        self.total_spam = sum(self.spam_counts.values())
        self.vocab_size = len(self.vocab)
        total_docs = self.ham_docs + self.spam_docs
        self.prior_ham = self.ham_docs / total_docs
        self.prior_spam = self.spam_docs / total_docs
        self.fitted = True
    def _word_prob(self, word, class_counts, total_count):
            return (class_counts[word] + self.alpha) / (total_count +
self.alpha * self.vocab_size)


    def predict(self, tokens):
        if not self.fitted:
            raise Exception("Call fit() before predict().")
        log_ham = math.log(self.prior_ham)
        log_spam = math.log(self.prior_spam)
        for word in tokens:
             log_ham += math.log(self._word_prob(word, self.ham_counts,
self.total_ham))
            log_spam += math.log(self._word_prob(word, self.spam_counts,
self.total_spam))
        return 'ham' if log_ham > log_spam else 'spam'
    def predict_many(self, docs):
        return [self.predict(doc) for doc in docs]
```

Appendix 24. Initializing and Training a Naive Bayes Classifier with Laplace Smoothing Using Tokenized Ham and Spam Documents



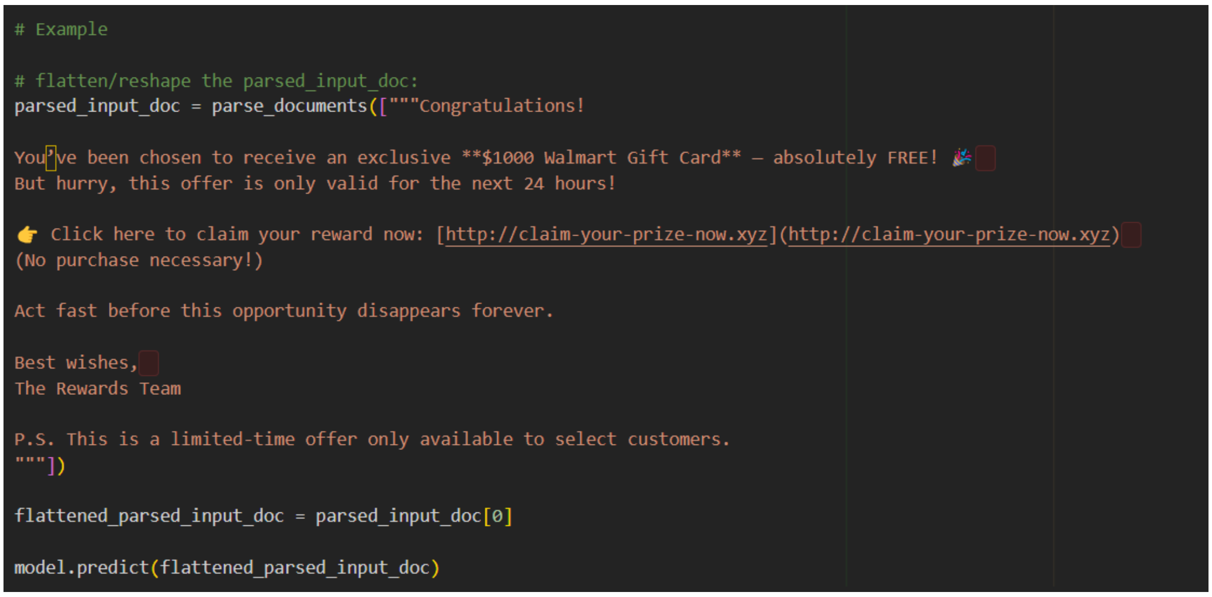
Here, it initialize an instance of the NaiveBayesClassifier using a smoothing parameter (alpha) of 0.05. This value controls the amount of Laplace smoothing applied during training and prediction, helping to prevent zero probabilities for unseen words.

```python
# Initialize the model with some random lambda value
model = NaiveBayesClassifier(alpha=0.05)
```

Now we train the Naive Bayes classifier by calling its fit method with the tokenized ham and spam documents. This step updates the model with word frequencies, vocabulary, and document counts, preparing it for making predictions on unseen emails.

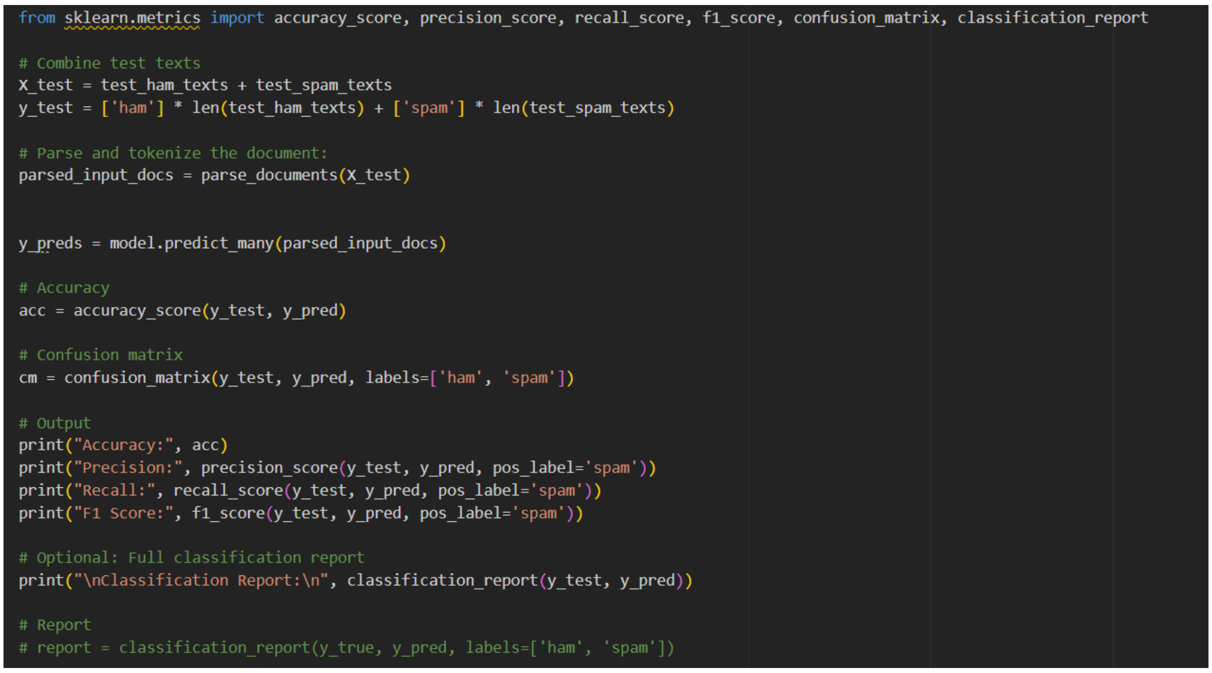
```python
# train/fit the model
model.fit(parsed_ham_docs, parsed_spam_docs)
```

Appendix 25. Classifying a Sample Email Using the Naive Bayes Model: A Promotional Message Parsed and Predicted as Spam

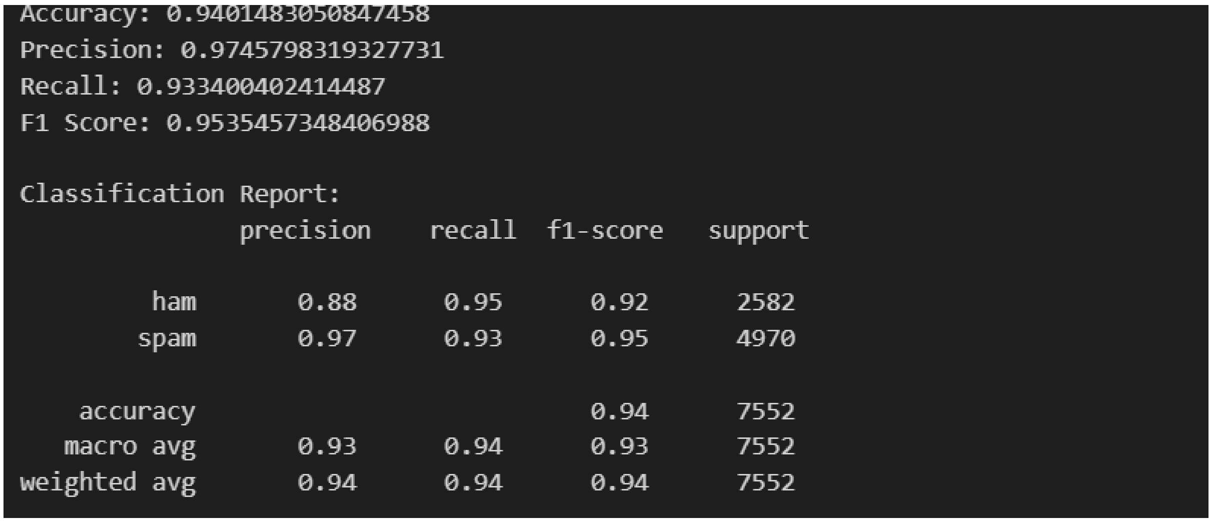


```python
# Example

# flatten/reshape the parsed_input_doc:
parsed_input_doc = parse_documents(["""Congratulations!

You’ve been chosen to receive an exclusive **$1000 Walmart Gift Card** – absolutely FREE! 🎉
But hurry, this offer is only valid for the next 24 hours!

👉 Click here to claim your reward now: [http://claim-your-prize-now.xyz](http://claim-your-prize-now.xyz)
(No purchase necessary!)

Act fast before this opportunity disappears forever.

Best wishes,
The Rewards Team

P.S. This is a limited-time offer only available to select customers.
"""])

flattened_parsed_input_doc = parsed_input_doc[0]

model.predict(flattened_parsed_input_doc)
```

Appendix 26. Evaluating the Performance of the Naive Bayes Spam Classifier Using Accuracy, Precision, Recall, F1 Score, and Confusion Matrix from scikit-learn

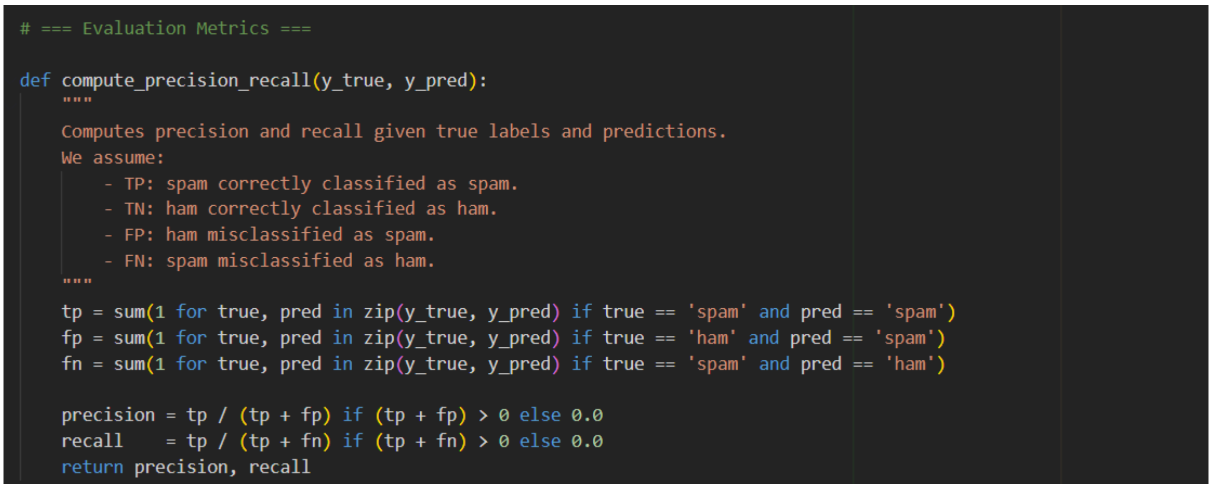


```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, confusion_matrix, classification_report

# Combine test texts
X_test = test_ham_texts + test_spam_texts
y_test = ['ham'] * len(test_ham_texts) + ['spam'] * len(test_spam_texts)

# Parse and tokenize the document:
parsed_input_docs = parse_documents(X_test)


y_preds = model.predict_many(parsed_input_docs)

# Accuracy
acc = accuracy_score(y_test, y_pred)

# Confusion matrix
cm = confusion_matrix(y_test, y_pred, labels=['ham', 'spam'])

# Output
print("Accuracy:", acc)
print("Precision:", precision_score(y_test, y_pred, pos_label='spam'))
print("Recall:", recall_score(y_test, y_pred, pos_label='spam'))
print("F1 Score:", f1_score(y_test, y_pred, pos_label='spam'))

# Optional: Full classification report
print("\nClassification Report:\n", classification_report(y_test, y_pred))

# Report
# report = classification_report(y_true, y_pred, labels=['ham', 'spam'])
```



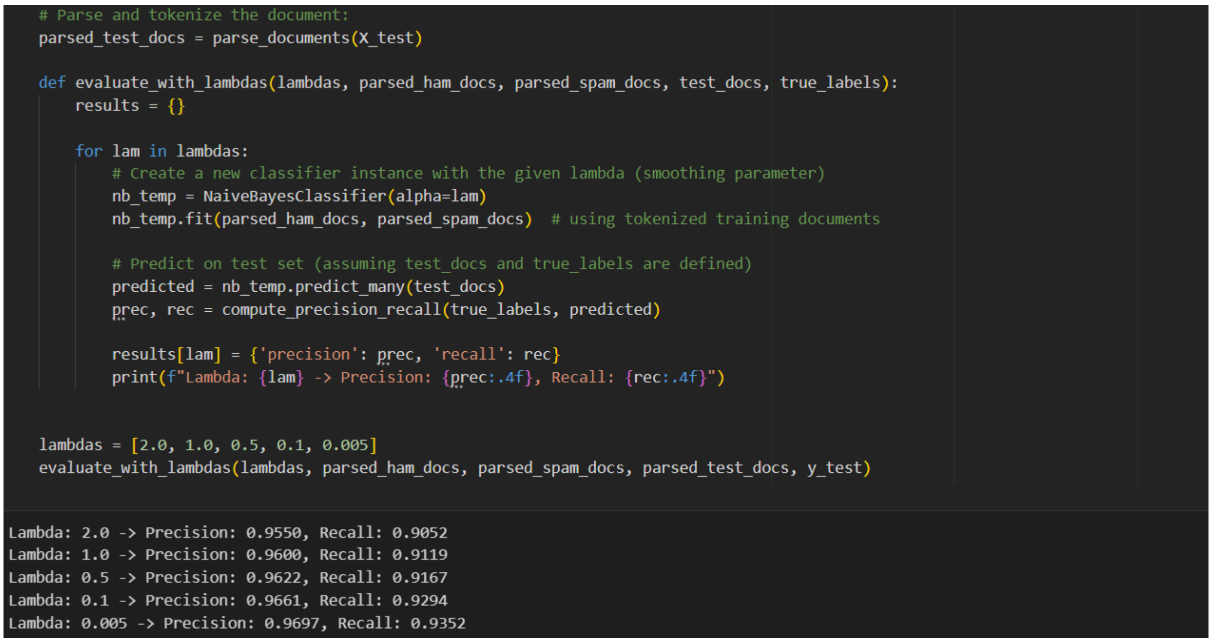
```
Accuracy: 0.9401483050847458
Precision: 0.9745798319327731
Recall: 0.933400402414487
F1 Score: 0.9535457348406988

Classification Report:
               precision    recall  f1-score   support

         ham       0.88      0.95      0.92      2582
        spam       0.97      0.93      0.95      4970

    accuracy                           0.94      7552
   macro avg       0.93      0.94      0.93      7552
weighted avg       0.94      0.94      0.94      7552
```

Appendix 27. Manual Implementation of Precision and Recall Metrics for Spam Classification Based on True Positive, False Positive, and False Negative Counts



```python
# === Evaluation Metrics ===

def compute_precision_recall(y_true, y_pred):
    """
    Computes precision and recall given true labels and predictions.
    We assume:
        - TP: spam correctly classified as spam.
        - TN: ham correctly classified as ham.
        - FP: ham misclassified as spam.
        - FN: spam misclassified as ham.
    """
    tp = sum(1 for true, pred in zip(y_true, y_pred) if true == 'spam' and pred == 'spam')
    fp = sum(1 for true, pred in zip(y_true, y_pred) if true == 'ham' and pred == 'spam')
    fn = sum(1 for true, pred in zip(y_true, y_pred) if true == 'spam' and pred == 'ham')

    precision = tp / (tp + fp) if (tp + fp) > 0 else 0.0
    recall    = tp / (tp + fn) if (tp + fn) > 0 else 0.0
    return precision, recall
```

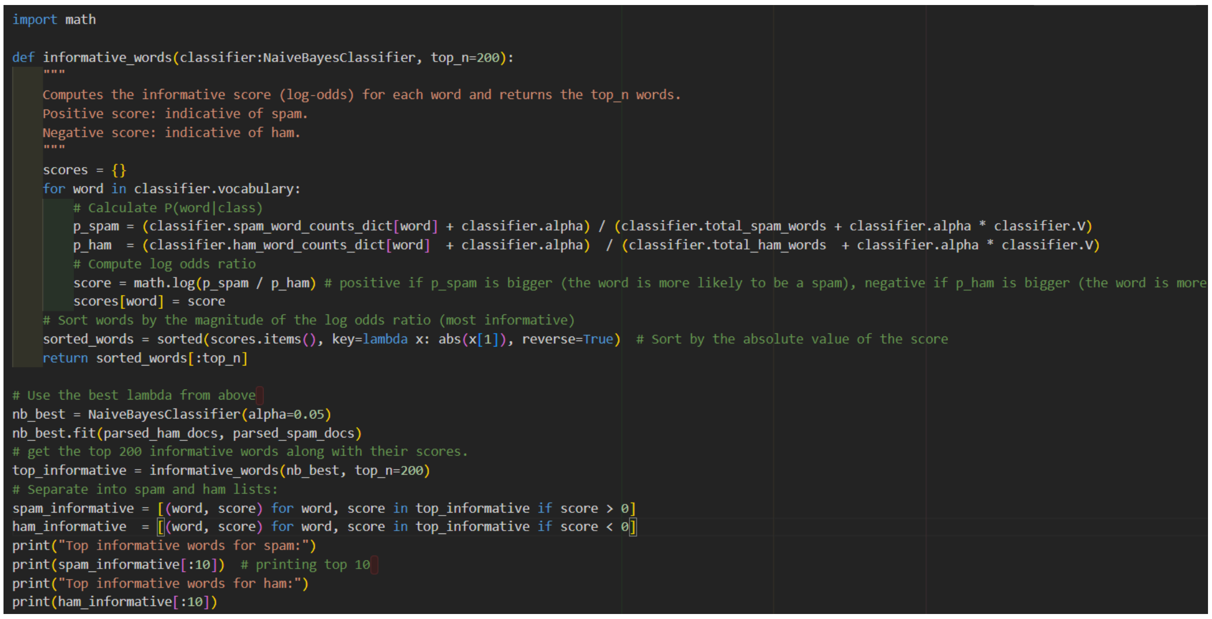
Appendix 28. Evaluating Naive Bayes Spam Classifier with Varying Smoothing Parameters (Alpha): Precision and Recall Comparison



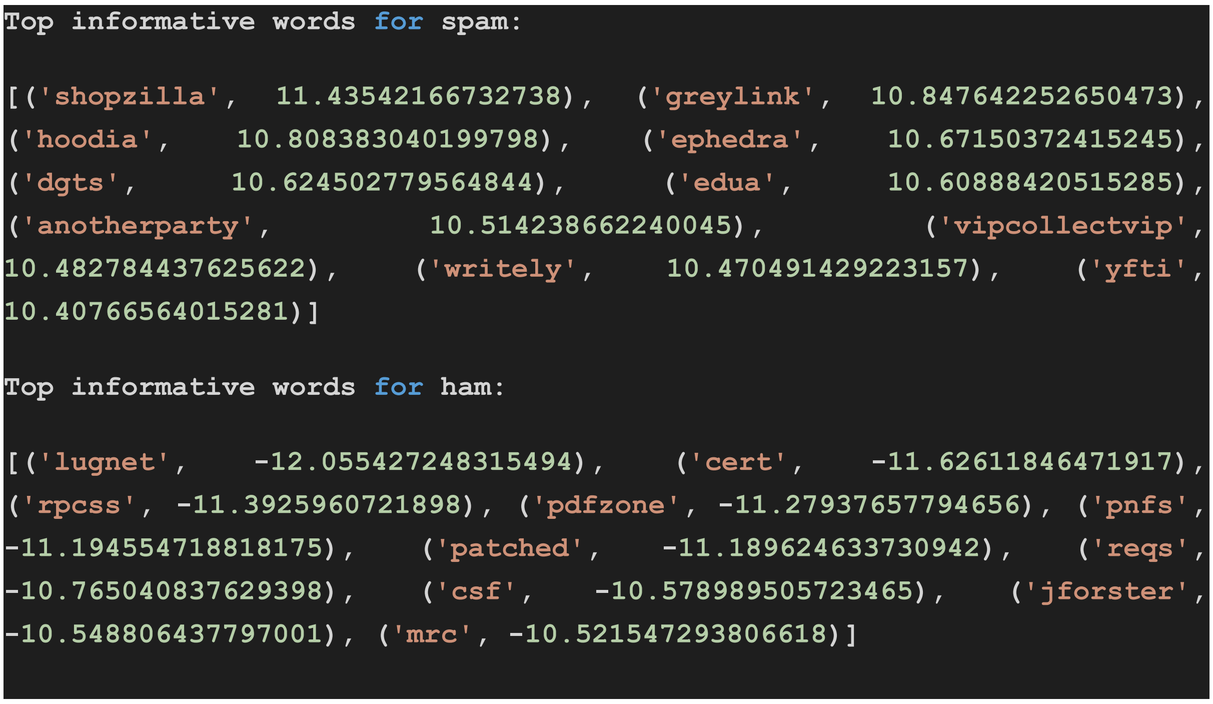
```python
# Parse and tokenize the document:
parsed_test_docs = parse_documents(X_test)

def evaluate_with_lambdas(lambdas, parsed_ham_docs, parsed_spam_docs, test_docs, true_labels):
    results = {}

    for lam in lambdas:
        # Create a new classifier instance with the given lambda (smoothing parameter)
        nb_temp = NaiveBayesClassifier(alpha=lam)
        nb_temp.fit(parsed_ham_docs, parsed_spam_docs)  # using tokenized training documents

        # Predict on test set (assuming test_docs and true_labels are defined)
        predicted = nb_temp.predict_many(test_docs)
        prec, rec = compute_precision_recall(true_labels, predicted)

        results[lam] = {'precision': prec, 'recall': rec}
        print(f"Lambda: {lam} -> Precision: {prec:.4f}, Recall: {rec:.4f}")


lambdas = [2.0, 1.0, 0.5, 0.1, 0.005]
evaluate_with_lambdas(lambdas, parsed_ham_docs, parsed_spam_docs, parsed_test_docs, y_test)
```

```
Lambda: 2.0 -> Precision: 0.9550, Recall: 0.9052
Lambda: 1.0 -> Precision: 0.9600, Recall: 0.9119
Lambda: 0.5 -> Precision: 0.9622, Recall: 0.9167
Lambda: 0.1 -> Precision: 0.9661, Recall: 0.9294
Lambda: 0.005 -> Precision: 0.9697, Recall: 0.9352
```

Appendix 29. Identifying the Most Informative Words in Spam Detection using Log-Odds Scoring

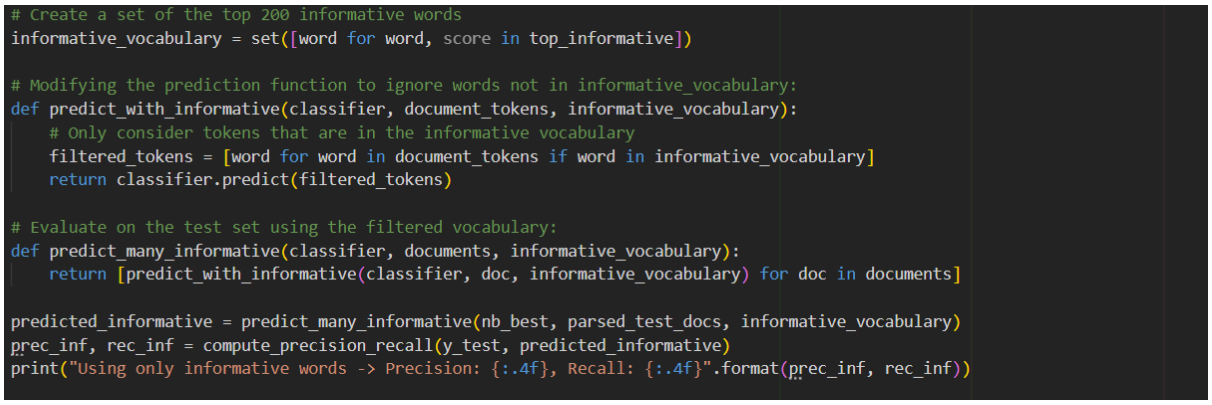


```python
import math

def informative_words(classifier:NaiveBayesClassifier, top_n=200):
    """
    Computes the informative score (log-odds) for each word and returns the top_n words.
    Positive score: indicative of spam.
    Negative score: indicative of ham.
    """
    scores = {}
    for word in classifier.vocabulary:
        # Calculate P(word|class)
        p_spam = (classifier.spam_word_counts_dict[word] + classifier.alpha) / (classifier.total_spam_words + classifier.alpha * classifier.V)
        p_ham  = (classifier.ham_word_counts_dict[word]  + classifier.alpha)  / (classifier.total_ham_words  + classifier.alpha * classifier.V)
        # Compute log odds ratio
        score = math.log(p_spam / p_ham) # positive if p_spam is bigger (the word is more likely to be a spam), negative if p_ham is bigger (the word is more
        scores[word] = score
    # Sort words by the magnitude of the log odds ratio (most informative)
    sorted_words = sorted(scores.items(), key=lambda x: abs(x[1]), reverse=True)  # Sort by the absolute value of the score
    return sorted_words[:top_n]

# Use the best lambda from above
nb_best = NaiveBayesClassifier(alpha=0.05)
nb_best.fit(parsed_ham_docs, parsed_spam_docs)
# get the top 200 informative words along with their scores.
top_informative = informative_words(nb_best, top_n=200)
# Separate into spam and ham lists:
spam_informative = [(word, score) for word, score in top_informative if score > 0]
ham_informative  = [(word, score) for word, score in top_informative if score < 0]
print("Top informative words for spam:")
print(spam_informative[:10])  # printing top 10
print("Top informative words for ham:")
print(ham_informative[:10])
```

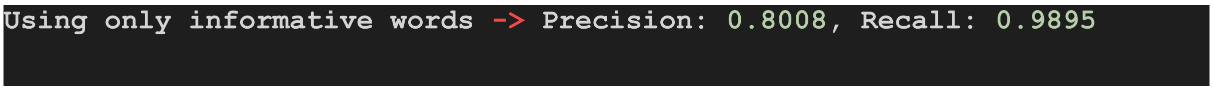


```
Top informative words for spam:

[('shopzilla', 11.43542166732738), ('greylink', 10.847642252650473),
('hoodia', 10.808383040199798), ('ephedra', 10.67150372415245),
('dgts', 10.624502779564844), ('edua', 10.60888420515285),
('anotherparty', 10.514238662240045), ('vipcollectvip',
10.482784437625622), ('writely', 10.470491429223157), ('yfti',
10.40766564015281)]

Top informative words for ham:

[('lugnet', -12.055427248315494), ('cert', -11.62611846471917),
('rpcss', -11.3925960721898), ('pdfzone', -11.27937657794656), ('pnfs',
-11.194554718818175), ('patched', -11.189624633730942), ('reqs',
-10.765040837629398), ('csf', -10.578989505723465), ('jforster',
-10.548806437797001), ('mrc', -10.521547293806618)]
```

Appendix 30. Filtering Predictions Using Only the Most Informative Words



```
# Create a set of the top 200 informative words
informative_vocabulary = set([word for word, score in top_informative])

# Modifying the prediction function to ignore words not in informative_vocabulary:
def predict_with_informative(classifier, document_tokens, informative_vocabulary):
    # Only consider tokens that are in the informative vocabulary
    filtered_tokens = [word for word in document_tokens if word in informative_vocabulary]
    return classifier.predict(filtered_tokens)

# Evaluate on the test set using the filtered vocabulary:
def predict_many_informative(classifier, documents, informative_vocabulary):
    return [predict_with_informative(classifier, doc, informative_vocabulary) for doc in documents]

predicted_informative = predict_many_informative(nb_best, parsed_test_docs, informative_vocabulary)
prec_inf, rec_inf = compute_precision_recall(y_test, predicted_informative)
print("Using only informative words -> Precision: {:.4f}, Recall: {:.4f}".format(prec_inf, rec_inf))
```



```
Using only informative words -> Precision: 0.8008, Recall: 0.9895
```

C. Performance Summary Tables

The tables below summarize the evaluation metrics and the effect of Laplace smoothing using different lambda values:

Evaluation Metrics

Metric	Score
Accuracy	0.9401
Precision	0.9745
Recall	0.9334
F1 Score	0.9535

Table 5: Model Evaluation Metrics on Full Feature Set

Lambda Tuning Table

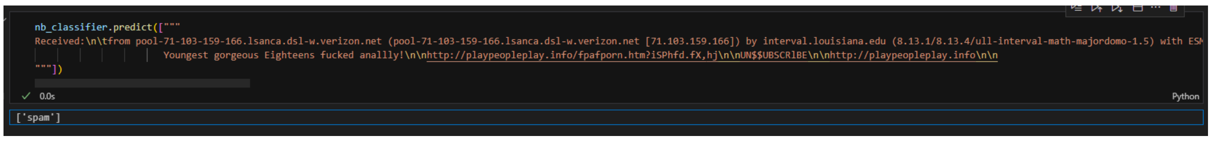
Lambda (α)	Precision	Recall
2.0	0.9550	0.9052
1.0	0.9600	0.9119
0.5	0.9622	0.9167
0.1	0.9661	0.9294
0.005	0.9697	0.9352

Table 6: Impact of Lambda (α) on Precision and Recall

The evaluation metrics confirm the effectiveness of the Multinomial Naive Bayes classifier, achieving an overall accuracy of 94.01%, with high precision (97.45%) and recall (93.34%), resulting in an F1-score of 95.35%. To further optimize the model, lambda tuning was performed for Laplace smoothing. As shown in the Lambda Tuning Table, the lowest lambda value tested (α = 0.005) yielded the best results, achieving the highest precision of 96.97% and recall of 93.52%. These findings suggest that careful tuning of the smoothing parameter can significantly influence classification performance, especially in datasets with sparse feature distributions.

Sensitive Content Discovery Snapshot

The Naive Bayes classifier demonstrated its capability to identify highly explicit spam content during testing. In one such instance, a message containing overtly adult-oriented language and linked URLs was accurately flagged as spam. This classification was based on a combination of red-flag keywords and source indicators, including known spam domains and inappropriate phrases. The ability to detect such sensitive material illustrates the model's practical utility in real-world deployment, especially in environments where email content moderation and workplace appropriateness are critical. This example serves as a testament to the classifier's robustness in handling not just generic spam but also extreme edge cases that demand ethical and contextual vigilance.



```
nb_classifier.predict(["""
Received:\n\tfrom pool-71-103-159-166.lsanca.dsl-w.verizon.net (pool-71-103-159-166.lsanca.dsl-w.verizon.net [71.103.159.166]) by interval.louisiana.edu (8.13.1/8.13.4/ull-interval-math-majordomo-1.5) with ESM
                    Youngest gorgeous Eighteens fucked anallly!\n\nhttp://playpeopleplay.info/fpafporn.htm?iSPhfd.fX,hj\n\nUN$$UBSCRlBE\n\nhttp://playpeopleplay.info\n\n
"""])
0.0s
['spam']
```

```
nb_classifier.predict(["""

Received:

                from    pool-71-103-159-166.lsanca.dsl-w.verizon.net
(pool-71-103-159-166.lsanca.dsl-w.verizon.net   [71.103.159.166])   by
interval.louisiana.edu

Youngest gorgeous Eighteens fucked anallly!

http://playpeopleplay.info/fpafporn.htm?iSPhfd.fX,hj

UN$$UBSCRlBE

http://playpeopleplay.info

"""])
```

Figure of the Visual evidence of the classifier successfully labeling explicitly rated-18 content from a flagged folder as spam, showcasing its capacity to detect adult-oriented or inappropriate email patterns.

References

Hovold, J. (2006, May). Naive Bayes spam filtering using word-position-based attributes and length-sensitive classification thresholds. In S. Werner (Ed.), *Proceedings of the 15th Nordic Conference of Computational Linguistics (NODALIDA 2005)* (pp. 78–87). Retrieved from https://aclanthology.org/W05-1712/

Manansala, R. (n.d.). *Naive Bayes Classifier*. GitHub. https://github.com/rn-mnsl/naive-bayes-classifier

Suárez, J. (n.d.). *Simple Naive Bayes Classifier for Email Classification* [PDF]. GitHub. https://github.com/suarezjessie/naive-bayes-spam-filter/blob/master/Simple%20Naive%20Bayes%20Classifier%20for%20Email%20Classification.pdf