Natural Language Processing

FINAL PROJECT

-Pavan Katasani

-Sai Sisira Pathakamuri

Task Dataset:

Email Spam Corpora:

Email spam corpora are collections of emails that have been labeled as spam or non-spam (ham). They are used for training and evaluating machine learning models and algorithms to detect and filter out spam emails.

These corpora typically contain a large number of email messages, with each message labeled as either spam or ham. The labels are assigned by human annotators who review the content of the emails and determine whether they are unsolicited spam or legitimate non-spam messages.

Email spam corpora are valuable resources for developing and testing spam filters and other email classification systems. They help researchers and developers train machine learning models to identify patterns and characteristics of spam emails, such as specific keywords, suspicious attachments, or deceptive subject lines.

The dataset we used is from, Enron Email Corpus: Although not specifically focused on spam, the Enron Email Corpus is a widely used dataset for email-related research. It includes a large collection of emails from the Enron Corporation, which collapsed due to a financial scandal. The corpus contains a mix of spam and non-spam emails.

These corpora provide valuable resources for researchers and developers to train machine learning models, assess the effectiveness of spam filters, and innovate in the field of email classification and spam detection. However, it is crucial to adhere to data usage and privacy regulations while working with email data, ensuring the secure and responsible handling of personal information.

Although there are 3 large directories of both Spam and Ham emails, only the first one is used here with 3,672 regularemails in the “ham” folder, and 1,500 emails in the “spam” folder.

Text Pre-processing and Tokenization:

In this step, the code imports necessary libraries and defines functions to read emails from the provided folder paths. The emails are read and stored in separate lists for ham and spam emails. The read_emails function iterates over the files in the folder and reads their content. The emails are then returned as a list.

In [1]:
```
import os
import email
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.probability import FreqDist
from nltk.classify import NaiveBayesClassifier
from nltk.metrics import precision, recall, f_measure, ConfusionMatrix
from sklearn.model_selection import KFold
from nltk.collocations import BigramCollocationFinder
from nltk.metrics import BigramAssocMeasures
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report
# Step 1: Data Processing and Tokenization

def read_emails(folder_path):
    emails = []
    for filename in os.listdir(folder_path):
        with open(os.path.join(folder_path, filename), "r", encoding="latin1") as file:
            emails.append(file.read())
    return emails

ham_folder = "/Users/pavan/Downloads/FinalProjectData/EmailSpamCorpora/corpus/ham"
spam_folder = "/Users/pavan/Downloads/FinalProjectData/EmailSpamCorpora/corpus/spam"

ham_emails = read_emails(ham_folder)
spam_emails = read_emails(spam_folder)
```

Feature Extraction:

The code defines a function extract_features to extract features from the emails. It tokenizes each email using the word_tokenize function from NLTK. It then filters the tokens, converting them to lowercase and removing stopwords using the stopwords.words('english') function. The filtered tokens are added to the features list. The function returns the extracted features.

The code selects the top N most frequent words as **unigram features**. Unigrams refer to single words in the text. By using unigram features, the classifier can learn patterns and associations between individual words, which can be helpful in tasks such as sentiment analysis or text categorization.It combines the ham and spam features and creates a frequency distribution using FreqDist from NLTK. The most_common(N) method is used to retrieve the N most frequent words, which are stored in the word_features list.

The code defines a function email_features to create a dictionary of features for a given email text. It tokenizes the email, creates a set of unique words, and iterates over the word_features list. For each word feature, it checks if the word is present in the email words and assigns a boolean value accordingly. The function returns the features dictionary.

The code also extracts **bigram features** using collocations. Bigrams are pairs of consecutive words in the text. Including bigram features allows the classifier to capture more context and sequential information. This can be particularly useful in tasks where the order of words is important, such as language modeling or named entity recognition. It uses the BigramCollocationFinder class from NLTK to find bigrams in the ham and spam features. The bigram scores are calculated using score_ngrams and stored in ham_bigram_scores and spam_bigram_scores. The bigrams with the highest scores (up to N) are stored in the bigram_features list.

Similar to the email_features function, the email_features_with_bigrams function is written for the bigram features. We also define a new feature function using POS (Part-of-speech) tag counts called **pos_features** which uses the nltk.pos_tag(tokens). It counts the occurrences of each POS tag and includes them as features. These features capture additional linguistic information to the classifier, enabling it to consider syntactic patterns and grammatical structures.

Another feature set is created using **all features** combined. This approach considers a broader range of information, including word-level associations, contextual information, and linguistic features. It can potentially improve the overall performance of the classifier by leveraging multiple sources of information.

In [2]:

```
# Step 2: Feature Extraction
def extract_features(emails):
    features = []
    for email_text in emails:
        tokens = word_tokenize(email_text)
        filtered_tokens = [token.lower() for token in tokens if token.isalpha() and token.lower() not
        features.extend(filtered_tokens)
    return features

# Extract features from ham and spam emails
ham_features = extract_features(ham_emails)
spam_features = extract_features(spam_emails)

# Select top N most frequent words as unigram features
N = 1000
all_features = FreqDist(ham_features + spam_features)
word_features = [feature for feature, _ in all_features.most_common(N)]

def email_features(email_text):
    email_words = set(word_tokenize(email_text))
    features = {}
    for word in word_features:
        features[word] = (word in email_words)
    return features

# Extract bigram features using collocations
bigram_measures = BigramAssocMeasures()
ham_bigram_finder = BigramCollocationFinder.from_words(ham_features)
ham_bigram_scores = ham_bigram_finder.score_ngrams(bigram_measures.raw_freq)
spam_bigram_finder = BigramCollocationFinder.from_words(spam_features)
spam_bigram_scores = spam_bigram_finder.score_ngrams(bigram_measures.raw_freq)
bigram_features = [bigram for bigram, _ in (ham_bigram_scores + spam_bigram_scores)[:N]]
```

```python
def email_features_with_bigrams(email_text):
    email_words = set(word_tokenize(email_text))
    features = {}
    for word in word_features:
        features[word] = (word in email_words)
    for bigram in bigram_features:
        features[bigram] = (bigram in email_words)
    return features

# Define new feature function using POS tag counts
def pos_features(email_text):
    tokens = word_tokenize(email_text)
    tagged_tokens = nltk.pos_tag(tokens)
    features = {}
    for _, tag in tagged_tokens:
        features[tag] = features.get(tag, 0) + 1
    return features

# Convert emails into feature sets
ham_feature_sets = [(email_features(email), 'ham') for email in ham_emails]
spam_feature_sets = [(email_features(email), 'spam') for email in spam_emails]
unigram_feature_sets = [(email_features(email), 'ham') for email in ham_emails] + [(email_featu
bigram_feature_sets = [(email_features_with_bigrams(email), 'ham') for email in ham_emails] + [
pos_feature_sets = [(pos_features(email), 'ham') for email in ham_emails] + [(pos_features(emai
all_feature_sets = [(email_features(email), 'ham') for email in ham_emails] + [(email_features(
```

Classification, Cross-Validation, and Evaluation:

The code sets the number of folds for cross-validation (k) and initializes a KFold object from sklearn.model_selection.
The code establishes the number of folds for cross-validation, denoted as 'k', and initializes a KFold object from the sklearn.model_selection module.

A function called 'train_and_evaluate' is defined to handle the training and evaluation of a classifier. It requires a classifier object, a training set, and a test set as inputs. The classifier is trained using the 'fit' method, and predictions are made on the test set using the 'predict' method. Various evaluation metrics such as accuracy, precision, recall, F1 score, and the confusion matrix are computed using functions from the NLTK library. The evaluation metrics and confusion matrix are then displayed.

The code executes cross-validation and evaluation for Naive Bayes classifier. Different sets of features, namely 'unigram_feature_sets', ‘bigram_feature_sets’, ‘pos_feature_sets’ and 'all_feature_sets', are utilized. The code iterates through the folds and invokes the 'train_and_evaluate' function with the corresponding train and test sets.

Cross-validation is used to assess the performance and generalization ability of the classifiers. In the code, k-fold cross-validation is applied, where the data is divided into k subsets (folds). The choice of k (in this case, k=5) is a trade-off between computational cost and obtaining reliable estimates of performance.

Justification for k-fold Cross-validation: Cross-validation helps to estimate how well the classifier will perform on unseen data. By splitting the data into multiple folds and iteratively training and testing the classifier, we can obtain more robust performance measures by averaging the results across different data partitions. A higher value of k (e.g., 5 or 10) is generally preferred as it reduces bias and provides a more accurate estimation of the classifier's performance. Reliability of Cross-validation: By using k-fold cross-validation, the evaluation results are less likely to be biased by a specific data split. The averaging of performance measures across multiple folds helps to reduce the impact of data variability and provides a more reliable estimate of the classifier's performance. This approach ensures that the evaluation results are not overly dependent on a single train-test split and are more generalizable to unseen data. Overall, the combination of appropriate feature selection and k-fold cross-validation allows for a comprehensive evaluation of the classifiers' performance and provides reliable estimates of their effectiveness on unseen data.

The code provided implements the evaluation process using the following steps:

a. Import necessary libraries and modules: The code imports the required modules, including classification_report from sklearn.metrics, ConfusionMatrix from nltk.metrics, and KFold from sklearn.model_selection.

b. Define the number of folds: The variable 'k' is set to 5, indicating that the data will be divided into 5 folds for cross-validation.

c. Initialize KFold object: The KFold object 'kf' is initialized with the specified number of folds.

d. Define the 'train_and_evaluate' function: This function takes a classifier, train set, and test set as input. It trains the classifier on the train set and evaluates its performance on the test set. The function calculates accuracy, generates a confusion matrix, and computes a classification report containing precision, recall, and F1-score.

e. Evaluate Naive Bayes classifier with unigram features: The code performs cross-validation and evaluation for the Naive Bayes classifier using unigram features. It prints the results for each fold and calculates the average accuracy and average metrics for precision, recall, and F1-score across all folds.

f. Evaluate Naive Bayes classifier with bigram features: Similar to the previous step, the code performs cross-validation and evaluation for the Naive Bayes classifier using bigram features.

g. Evaluate Naive Bayes classifier with POS features: Again, the code performs cross-validation and evaluation for the Naive Bayes classifier using part-of-speech (POS) features. h. Evaluate Naive Bayes classifier with all features: Finally, the code performs cross-validation and evaluation for the Naive Bayes classifier using all available features.

h. Evaluate Naive Bayes classifier with all features: Similar to the previous step, the code performs cross-validation and evaluation for the Naive Bayes classifier using all features.

```python
from sklearn.metrics import classification_report
from nltk.metrics import ConfusionMatrix
from sklearn.model_selection import KFold

k = 5  # Number of folds for cross-validation
kf = KFold(n_splits=k)

# Function to train and evaluate the classifier
def train_and_evaluate(classifier, train_set, test_set):
    classifier = classifier.train(train_set)

    # Evaluate the classifier on the test set
    true_labels = [label for _, label in test_set]
    predicted_labels = [classifier.classify(features) for features, _ in test_set]

    # Calculate evaluation measures
    accuracy = nltk.classify.accuracy(classifier, test_set)
    confusion_matrix = ConfusionMatrix(true_labels, predicted_labels)
    report = classification_report(true_labels, predicted_labels, output_dict=True)

    return accuracy, report['accuracy'], report

# Perform cross-validation and evaluation for Naive Bayes classifier with unigram features
print("Results for Naive Bayes Classifier with unigram features:")
nb_classifier_unigram = NaiveBayesClassifier.train(unigram_feature_sets)

total_accuracy_unigram = 0
total_report_unigram = None

for train_index, test_index in kf.split(unigram_feature_sets):
    train_set = [unigram_feature_sets[i] for i in train_index]
    test_set = [unigram_feature_sets[i] for i in test_index]
    accuracy, _, report = train_and_evaluate(nb_classifier_unigram, train_set, test_set)
    total_accuracy_unigram += accuracy

    if total_report_unigram is None:
        total_report_unigram = report
```

Results and Discussion:

These results provide insights into the effectiveness of the classifiers with different feature sets and can be used to compare their performance. The results for each classifier are presented below:

a. Naive Bayes Classifier with unigram features:

```
Average Accuracy (Unigram): 0.9346587054635158

Average metrics for 'ham':
Precision: 0.7981167608286253
Recall: 0.7308951181368479
F1-score: 0.7629977183081735

Average metrics for 'spam':
Precision: 0.3833001988071570 4
Recall: 0.3959198412764297
F1-score: 0.3893274603074312 6

Average metrics for 'macro avg':
Precision: 0.5907084798178912
Recall: 0.5634074797066388
F1-score: 0.5761625893078024

Average metrics for 'weighted avg':
Precision: 0.9914392773644046
Recall: 0.9346587054635158
F1-score: 0.9615087526943693
```

- The Naive Bayes classifier using unigram features achieved an average accuracy of 0.9347. It performed well in classifying "ham" messages, but struggled with "spam" messages, resulting in lower precision, recall, and F1-score. Further improvements are needed to enhance its ability to handle "spam" effectively.
- The classifier had lower precision, recall, and F1-score for classifying "spam" messages, indicating challenges in accurately identifying and distinguishing spam content.

b. Naive Bayes Classifier with bigram features:

```
Results for Naive Bayes Classifier with bigram features:
Average Accuracy (Bigram): 0.9346587054635158

Average metrics for 'ham':
Precision: 0.7981167608286253
Recall: 0.7308951181368479
F1-score: 0.7629977183081735

Average metrics for 'spam':
Precision: 0.38330019880715704
Recall: 0.3959198412764297
F1-score: 0.38932746030743126

Average metrics for 'macro avg':
Precision: 0.5907084798178912
Recall: 0.5634074797066388
F1-score: 0.5761625893078024

Average metrics for 'weighted avg':
Precision: 0.9914392773644046
Recall: 0.9346587054635158
F1-score: 0.9615087526943693
```

- "spam" classes are generally lower with bigram features. This indicates that the bigram representation may not be as effective in capturing the discriminative patterns between spam and non-spam messages as compared to unigram features.

c. Naive Bayes Classifier with POS features:

```
Results for Naive Bayes Classifier with POS features:
Average Accuracy (POS): 0.745887739560265

Average metrics for 'ham':
Precision: 0.7427777777777778
Recall: 0.7088852604690835
F1-score: 0.7212652563332307

Average metrics for 'spam':
Precision: 0.3656050955414013
Recall: 0.17986651281327565
F1-score: 0.2351357846671473

Average metrics for 'macro avg':
Precision: 0.5541914366595895
Recall: 0.44437588664117955
F1-score: 0.4782005205001891

Average metrics for 'weighted avg':
Precision: 0.9530655244681536
Recall: 0.745887739560265
F1-score: 0.8112176089529312
```

The Naive Bayes classifier with POS (Part of Speech) features achieved an average accuracy of 74.59%. However, the precision, recall, and F1-score for both the "ham" and "spam" classes are relatively low compared to previous feature representations. The precision, recall, and F1-score for the "ham" class are moderate, indicating reasonable accuracy in identifying non-spam messages.

d. Naive Bayes Classifier with all features:

```
Results for Naive Bayes Classifier with all features:
Average Accuracy (All Features): 0.9346587054635158

Average metrics for 'ham':
Precision: 0.7981167608286253
Recall: 0.7308951181368479
F1-score: 0.7629977183081735

Average metrics for 'spam':
Precision: 0.38330019880715704
Recall: 0.3959198412764297
F1-score: 0.38932746030743126

Average metrics for 'macro avg':
Precision: 0.5907084798178912
Recall: 0.5634074797066388
F1-score: 0.5761625893078024

Average metrics for 'weighted avg':
Precision: 0.9914392773644046
Recall: 0.9346587054635158
F1-score: 0.9615087526943693
```

The Naive Bayes classifier with all features achieves an average accuracy of 93.46%. However, its performance in classifying both "ham" and "spam" messages is suboptimal, with low precision, recall, and F1-score.

Experiment 1: Stopword Filtering

Stopwords are commonly used words (such as "the", "is", "and", etc.) that are often irrelevant for text analysis. In this experiment, the NLTK library was used to obtain a set of English stopwords. The email texts were tokenized using word_tokenize and filtered to remove stopwords. Feature extraction: Unigram features were extracted from the filtered email texts. The features represent individual words that appear in the emails.
In this experiment, the code modifies the feature extraction process to include stopword filtering. It defines a new function extract_features_with_stopwords that filters tokens using the stop_words set. The code creates new feature sets (unigram_feature_sets_with_stopwords) using the modified feature extraction function. It then trains and evaluates a Naive Bayes classifier (nb_classifier_with_stopwords) using cross-validation.

Experimental Design:

```
# Step 1: Extract features with stopwords
stop_words = set(stopwords.words('english'))

def extract_features_with_stopwords(emails):
    features = []
    for email_text in emails:
        tokens = word_tokenize(email_text)
        filtered_tokens = [token.lower() for token in tokens if token.isalpha() and token.lower()
        features.extend(filtered_tokens)
    return features

ham_features_with_stopwords = extract_features_with_stopwords(ham_emails)
spam_features_with_stopwords = extract_features_with_stopwords(spam_emails)
all_features_with_stopwords = FreqDist(ham_features_with_stopwords + spam_features_with_stopwords)
word_features_with_stopwords = [feature for feature, _ in all_features_with_stopwords.most_common(

def email_features_with_stopwords(email_text):
    email_words = set(word_tokenize(email_text))
    features = {}
    for word in word_features_with_stopwords:
        features[word] = (word in email_words)
    return features

unigram_feature_sets_with_stopwords = [(email_features_with_stopwords(email), 'ham') for email in

print("Results for Unigram Features with Stopword Filtering:")
kf = KFold(n_splits=3, shuffle=True)

for train_index, test_index in kf.split(unigram_feature_sets_with_stopwords):
    train_set = [unigram_feature_sets_with_stopwords[i] for i in train_index]
    test_set = [unigram_feature_sets_with_stopwords[i] for i in test_index]
    nb_classifier_with_stopwords = nltk.NaiveBayesClassifier.train(train_set)
    true_labels = [label for _, label in test_set]
    predicted_labels = [nb_classifier_with_stopwords.classify(features) for features, _ in test_se
    report = classification_report(true_labels, predicted_labels, output_dict=True)
    accuracy = report['accuracy']
    precision = report['macro avg']['precision']
    recall = report['macro avg']['recall']
    f1_score = report['macro avg']['f1-score']
```

Results and Analysis:

```
Results for Unigram Features with Stopword Filtering:
Accuracy: 0.9385150812064965
Precision: 0.9097772642196424
Recall: 0.9561160235798499
F1 Score: 0.9278812533248721
Classification Report:
              precision    recall  f1-score   support

         ham       1.00      0.92      0.96      1244
        spam       0.82      1.00      0.90       480

    accuracy                           0.94      1724
   macro avg       0.91      0.96      0.93      1724
weighted avg       0.95      0.94      0.94      1724

Accuracy: 0.9373549883990719
Precision: 0.9148902258079473
Recall: 0.9543916667195643
F1 Score: 0.9297870896214675
Classification Report:
              precision    recall  f1-score   support

         ham       1.00      0.91      0.95      1198
        spam       0.83      1.00      0.91       526

    accuracy                           0.94      1724
   macro avg       0.91      0.95      0.93      1724
weighted avg       0.95      0.94      0.94      1724

Accuracy: 0.9373549883990719
Precision: 0.9106880009534877
Recall: 0.9536749942398209
F1 Score: 0.9274862911266202
Classification Report:
              precision    recall  f1-score   support

         ham       1.00      0.92      0.95      1230
        spam       0.82      0.99      0.90       494

    accuracy                           0.94      1724
   macro avg       0.91      0.95      0.93      1724
weighted avg       0.95      0.94      0.94      1724
```

The accuracy values for the three iterations range from 0.9374 to 0.9385, indicating a consistently high level of accuracy.he precision values for the "ham" class (non-spam) range from 0.9098 to 0.9149.The recall values for the "ham" class range from 0.9116 to 0.9561.The F1 scores for the "ham" class range from 0.9275 to 0.9298.
Comparing the results with the previous unigram features with negation representation, we can observe that the performance is similar in terms of accuracy, precision, recall, and F1 score. However, the stopword filtering approach slightly improves precision and F1 score for the "ham" class. This suggests that removing stopwords from the text may help the classifier in identifying non-spam messages more accurately.

Overall, the results indicate that the classifier using unigram features with stopword filtering performs well in classifying messages as "ham" or "spam."

Interpretation:

Based on the experiment results, it can be concluded that stopword filtering has an impact on the performance of the Naive Bayes classifier for email classification. The removal of stopwords helped in improving the accuracy, precision, recall, and F1 score of the classifier. By eliminating irrelevant words from the analysis, the classifier could focus on more meaningful features, leading to better classification performance.

Experiment 2: Negation Representation

In this experiment, the objective was to investigate the effectiveness of incorporating negation representation in the feature extraction process for email classification. It modifies the email_features function to include negation words. For each word feature, it checks if the word is present in the email words and also if the word "not" is present in the email words. The features dictionary includes both the word feature and the negated feature. The code creates new feature sets (unigram_feature_sets_with_negation) using the modified feature extraction function. It then trains and evaluates a Naive Bayes classifier (nb_classifier_with_negation).

Code Screenshot:

```
# Step 2: Experiment with Negation Representation
negation_words = set(["not", "no", "n't"])

def email_features_with_negation(email_text):
    email_words = set(word_tokenize(email_text))
    features = {}
    for word in word_features_with_stopwords:
        features[word] = (word in email_words)
        features["not_" + word] = ("not" in email_words and word in email_words)
    return features

unigram_feature_sets_with_negation = [(email_features_with_negation(email), 'ham') for email in ham_

print("Results for Unigram Features with Negation Representation:")
for train_index, test_index in kf.split(unigram_feature_sets_with_negation):
    train_set = [unigram_feature_sets_with_negation[i] for i in train_index]
    test_set = [unigram_feature_sets_with_negation[i] for i in test_index]

    nb_classifier_with_negation = nltk.NaiveBayesClassifier.train(train_set)

    true_labels = [label for _, label in test_set]
    predicted_labels = [nb_classifier_with_negation.classify(features) for features, _ in test_set]

    report = classification_report(true_labels, predicted_labels, output_dict=True)
    accuracy = report['accuracy']
    precision = report['macro avg']['precision']
    recall = report['macro avg']['recall']
    f1_score = report['macro avg']['f1-score']

    print("Accuracy:", accuracy)
    print("Precision:", precision)
    print("Recall:", recall)
    print("F1 Score:", f1_score)
    print("Classification Report:")
    print(classification_report(true_labels, predicted_labels))
```

Experimental Design:

Stopword filtering: Stopwords are commonly used words (such as "the", "is", "and", etc.) that are often irrelevant for text analysis. In this experiment, the NLTK library was used to obtain a set of English stopwords. The email texts were tokenized using word_tokenize and filtered to remove stopwords. Feature extraction: Unigram features were extracted from the filtered email texts. The features represent individual words that appear in the emails. Naive Bayes classifier: The NLTK NaiveBayesClassifier was trained on the extracted features using the training set. Evaluation: The trained classifier was evaluated on the test set using accuracy, precision, recall, and the F1 score. The classification report was generated to provide a detailed analysis of the performance.

Results and Analysis:

```
Results for Unigram Features with Negation Representation:
Accuracy: 0.9425754060324826
Precision: 0.9198262330926397
Recall: 0.9562932115638165
F1 Score: 0.9344493489524823
Classification Report:
              precision    recall  f1-score   support

         ham       1.00      0.92      0.96      1210
        spam       0.84      0.99      0.91       514

    accuracy                           0.94      1724
   macro avg       0.92      0.96      0.93      1724
weighted avg       0.95      0.94      0.94      1724

Accuracy: 0.9431554524361949
Precision: 0.9184416206068675
Recall: 0.9576246370291603
F1 Score: 0.9342005234297108
Classification Report:
              precision    recall  f1-score   support

         ham       1.00      0.92      0.96      1225
        spam       0.84      0.99      0.91       499

    accuracy                           0.94      1724
   macro avg       0.92      0.96      0.93      1724
weighted avg       0.95      0.94      0.94      1724

Accuracy: 0.9501160092807425
Precision: 0.9257080027793261
Recall: 0.9627485188880165
F1 Score: 0.9411788976557888
Classification Report:
              precision    recall  f1-score   support

         ham       1.00      0.93      0.96      1237
        spam       0.85      0.99      0.92       487

    accuracy                           0.95      1724
   macro avg       0.93      0.96      0.94      1724
weighted avg       0.96      0.95      0.95      1724
```

The accuracy values for the three iterations are consistently high, ranging from 0.9426 to 0.9501.The precision values for the "ham" class (non-spam) are consistently high, ranging from 0.9184 to 0.9257. This suggests that when the classifier predicts a message as "ham," it is correct with a high degree of confidence.

The recall values for the "ham" class are also consistently high, ranging from 0.9200 to 0.9627. This indicates that the classifier is able to correctly identify a large proportion of the "ham" messages in the dataset.The F1 scores for the "ham" class are consistently high, ranging from 0.9342 to 0.9412. The F1 score is a harmonic mean of precision and recall, and it provides a balanced measure of the classifier's performance.Overall, the results suggest that the classifier using unigram features with negation representation performs well in classifying messages as "ham" or "spam."

Conclusion:

Initially, from the Naive Bayes classification results with cross fold validation above for unigram, bigram, pos, all features, we can see based on average metrics for 'ham' and 'spam' , the classifier performs moderately and there is room for improvement either with more data preprocessing, choosing better models, hyper parameter tuning.

But, based on the experiment results, it can be concluded that stopword filtering has an impact on the performance of the Naive Bayes classifier for email classification. The removal of stopwords helped in improving the accuracy, precision, recall, and F1 score of the classifier. By eliminating irrelevant words from the analysis, the classifier could focus on more meaningful features, leading to better classification performance.

Work division among teammates:

Pavan: Text Pre-processing and Tokenization,Feature Extraction, Documentation.
Sisira: Classification, Cross-Validation, and Evaluation, Experiments**,** Documentation.