SRH University Heidelberg

Task 2: SMS Spam Detection Based on Character N-gram models in Machine Learning

Author: Nikhil Kumar
Matriculation Number: 11037894
Email Address: nikhil.kumar@stud.hochschule-heidelberg.de
Date of Submission: 6th May 2024

Under the Guidance of Professor
Dr. Ing Milan Gnjatovic

Aim of the Task:

The aim of this project is to develop an efficient SMS spam filtering system using character N-gram models (Bi-gram, Tri-gram, Four-gram, etc.), aiming to accurately classify text messages as spam or legitimate. This involves training a model on the SMS SPAM Collection dataset to classify SMS and evaluating its performance based on accuracy and other relevant metrics.

Objective:

The primary objective of this project is to design and implement a robust SMS spam filtering system leveraging character n-gram models. Firstly, the project aims to explore and understand the effectiveness of character n-gram models in capturing the intrinsic features of text messages, especially in the context of SMS spam detection. By analyzing the frequency and distribution of character sequences within both spam and legitimate messages, the objective is to identify distinctive patterns that can be utilized for accurate classification.

Introduction:

Character n-gram models offer a promising alternative to word-based approaches by considering sequences of characters within SMS messages. By analyzing the frequency of character n-grams, where n represents the number of consecutive characters in a sequence, this method aims to capture unique patterns and linguistic features present in both legitimate messages and spam.

What is N-grams in NLP and Machine Learning ?

N-grams in NLP refers to contiguous sequences of n words extracted from text for language processing and analysis. An n-gram can be as short as a single word (unigram) or as long as multiple words (bigram, trigram, etc.). These n-grams capture the contextual information and relationships between words in a given text. [1g]

How N-grams in NLP works:

N-grams in NLP can be generated by sliding a window of n words across a sentence or text corpus. By extracting these n-grams, it becomes possible to analyze the frequency of occurrence of certain word sequences, identify collocations or commonly co-occurring words, and model the language patterns in a text. N-grams can also be used as features for training machine learning models in tasks like text classification or sentiment analysis.[1g]

A bit More about N-gram Models:

N-gram models are a type of probabilistic language model used in natural language processing (NLP) and machine learning. They are based on the idea that the probability of a word or character occurring in a sequence depends on the preceding (and sometimes following) words or characters in that sequence.

An "n-gram" refers to a contiguous sequence of n items from a given sample of text or speech. These items can be words, characters, or even phonemes, depending on the application. The value of n determines the size of the sequence.

Here are some example types of n-grams of words and n-grams of Characters:

1. Unigram (1-gram):
   - Example: "the", "quick", "brown", "fox"
   - Character n-grams:
     - "t", "h", "e"
     - "q", "u", "i", "c", "k"
     - "b", "r", "o", "w", "n"
     - "f", "o", "x"
2. Bigram (2-gram):
   - Example: "the quick", "quick brown", "brown fox"
   - Character n-grams:
     - "th", "he", "e "
     - "qu", "ui", "ic", "ck", "k "
     - "br", "ro", "ow", "wn", "n "
     - "fo", "ox"
3. Trigram (3-gram):
   - Example: "the quick brown", "quick brown fox"
   - Character n-grams:
     - "the", "he ", "e q", " qu", "qui", "uic", "ick", "ck ", "k b", " br", "bro", "row", "own", "wn "
     - "qui", "uic", "ick", "ck ", "k b", " br", "bro", "row", "ow ", "w f", " fo", "fox"
4. N-gram:
   - Example: For n=4, "the quick brown fox"
   - Character n-grams:
     - "the ", "he q", "e qu", " qui", "quic", "uick", "ick ", "ck b", "k br", " bro", "brow", "rown", "own ", "wn f", "n fo", " fox"

N-gram models are used in various NLP tasks, including text generation, machine translation, speech recognition, and information retrieval. One of the key applications of n-gram models is in language modeling, where the goal is to estimate the probability of a sequence of words occurring in given language.

The probability of an n-gram occurring can be estimated from a corpus of text by counting the frequency of each n-gram and dividing it by the total number of n-grams in the corpus. This is known as maximum likelihood estimation.

N-gram models are particularly useful because they capture some of the contextual information present in the text. For example, a bigram model can capture the likelihood of a word occurring after another word, providing some context for the prediction.

However, n-gram models suffer from the "sparsity problem," especially when dealing with large vocabularies or long sequences. This occurs when some n-grams in the test data were not seen in the training data, leading to zero probabilities for those unseen n-grams. Techniques like smoothing and backoff are used to address this issue.

Overall, n-gram models provide a simple yet effective way to model the sequential structure of text data and are widely used in various NLP applications.

Bi-gram Model:

Since my project task involves the use of the character Bigram, let me to go into detail about the Bigram Model.

Let us first consider the conditional probability $P(w_m|w1...w_{m-1})$, i.e., the probability that word wm will follow the immediately preceding sequence of words w1 $...w_{m-1}$. Intuitively, this probability can be estimated by counting word sequences in a language corpus: [1b]

$$P(w_m|w_1\dots w_{m-1}) = \frac{C(w_1\dots w_{m-1}w_m)}{C(w_1\dots w_{m-1})},$$

[1b]

where:

- $C(w_1 ...w_{m-1}w_m)$ is the number of occurrences of sequence $C(w_1 ...w_{m-1}w_m)$ in a corpus.
- $C(w_1 ...w_{m-1})$ is the number of occurrences of sequence $C(w_1 ...w_{m-1})$ in a corpus.

However, a limitation arises when dealing with large sequences where certain valid word sequences might not appear in the corpus, leading to a probability estimate of zero. To address this issue, a Markovian assumption is introduced, where the probability of word $wmwm$ depends only on the immediately preceding word w_{m-1}. In the bigram model, this assumption simplifies the conditional probability to only consider the probability of w_m given w_{m-1}: [1b]

$$P(w_m|w_{m-1}) \approx P(w_m|w_{m-1})$$

The maximum likelihood estimation of this probability is obtained by counting occurrences of bigrams in the corpus. If a bigram w_{m-1} w_m does not occur in the corpus, Laplace smoothing can be applied by adding one to each count. This smoothed maximum likelihood estimate is then calculated as:

$$\begin{aligned} P(w_m|w_{m-1}) &= \frac{C(w_{m-1}w_m)+1}{\sum\limits_{w\in V}(C(w_{m-1}w)+1)} \\ &= \frac{C(w_{m-1}w_m)+1}{\sum\limits_{w\in V}C(w_{m-1}w)+|V|} \\ &= \frac{C(w_{m-1}w_m)+1}{C(w_{m-1})+|V|}\ , \end{aligned} \qquad [1b]$$

where:

• $C(w_{m-1}w_m)$ is the number of occurrences of bigram $w_{m-1}w_m$ in a corpus,
• $C(w_{m-1})$ is the number of occurrences of word w_{m-1} in a corpus,
• $|V|$ is the number of words in a vocabulary. [1b]

This approach ensures that the probability estimates do not reach zero, mitigating the problem of zero probabilities.

Phases of SMS Spam Detection in this Project:

Phase 1: Pre-Processing and Data Set Division: -

- **Data Collection:** Collect a dataset of SMS messages labeled as spam or ham (not spam) from - https://archive.ics.uci.edu/ml/datasets/sms+spam+ collection.

- **Preprocessing:** Preprocess the SMS messages by converting the text to lowercase, doing Tokenization and lemmatization.

- **Split Data:** Split your dataset into 80% training and 20% testing sets to evaluate the performance of the model.

Phase 2: Training the dataset: -

In this case, the training set ought to produce two additional training models: one for the Spam class and the other for the Ham class. After that, apply character bigram frequecy for these training models 1 and 2.

Phase 3: Testing the data set: -

In this stage, we use trained SPAM and HAM models to calculate the log probabilities of the messages in order to test our model for the remaining 20% of the data set.
After that, we assess our testing data set using the assessment metric listed below.

1. **Confusion Matrix :** A much better way to evaluate the performance of a classifier is to look at the confusion matrix. The confusion matrix is a tabular representation that summarizes the performance of a classification model by presenting the counts of true positive, true negative, false positive, and false negative predictions for each class. The general idea is to count the number of times instances of class A are classified as class B. Each row in a confusion matrix represents an actual class, while each column represents a predicted class. For example, to know the number of times the classifier confused images of 5s with 3s, you would look in the 5th row and 3rd column of the confusion matrix. A perfect classifier would have only true positives and true negatives, so its confusion matrix would have nonzero values only on its main diagonal (top left to bottom right) [1h]

2. **Accuracy :** Accuracy measures the proportion of correctly classified samples out of the total number of samples. It is calculated as:
   Accuracy = Number of correct predictions/ Total Number of predictions.

3. **Precession :** You can find a lot of information in the confusion matrix, but occasionally you might want a metric that is shorter. An interesting one to look at is the accuracy of the positive predictions; this is called the precision of the classifier. [1h]
   Precision measures the ability of the classifier to correctly identify positive samples (true positives) out of all samples predicted as positive. It is calculated as:

$$P = \frac{\text{true positive}}{\text{true positive} + \text{false positive}} .$$

[1i]

   Precision is beneficial while the value of fake positives is high.

4. **Recall :** Recall measures the ability of the classifier to correctly identify positive samples (true positives) out of all actual positive samples. It is calculated as:

$$R = \frac{\text{true positive}}{\text{true positive} + \text{false negative}} .$$

[1i]

5. **F1 Score** : It is often convenient to combine precision and recall into a single metric called the F1 score, in particular if you need a simple way to compare two classifiers. The F1 score is the harmonic mean of precision and recall. Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values. As a result, the classifier will only get a high F1 score if both recall, and precision are high. [1i]

This Harmonic mean provides a balance between the two metrics.
It is calculated as:

$$F_1 = \frac{2}{P^{-1} + R^{-1}} = \frac{2PR}{P + R} \,. \quad [1i]$$

6. **Macro-average Metrics:** In multi-class classification tasks like MNIST, macro-average metrics compute the average of individual metrics (precision, recall, F1-score) calculated for each class. This approach treats all classes equally and is useful for assessing overall classifier performance across multiple categories.

Explaining the Program Functionality (What is going in the code):

As explained theoretically, each step is defined as a function in the program, and it is explained below. (Program File: ML_Project_Task2_NikhilKumar.ipynb or ML_Project_Task2_NikhilKumar.py)

Phase 1:

1. **load_dataset(file_path):**

    - This function loads the dataset from the specified file path.
    - It returns the dataset as a list of strings, where each string represents a line in the dataset.

2. **preprocess_message(message, lemmatizer):**

    - This function preprocesses a single message by performing tokenization and lemmatization.
    - It converts the message to lowercase and tokenizes it into individual characters.
    - Lemmatization is applied to each character, although in this context, it may not significantly affect the output.
    - Special start and end of message tokens ("<s>" and "</s>") are added to the beginning and end of the tokenized message.
    - The preprocessed message is returned as a list of tokens.

Please note that typically, tokenization and lemmatization are done at the word level, not the character level. This function might not work as expected for this typical text preprocessing task.

3. **preprocess_dataset(dataset, lemmatizer):**

    - This function preprocesses the entire dataset by applying the preprocess_message function to each message.
    - It returns the preprocessed dataset as a list of tuples, where each tuple contains the label and the preprocessed message.

4. **split_dataset(dataset, split_ratio=0.8):**

    - This function splits the dataset into training and testing sets based on the specified split ratio (default is 80% training, 20% testing).
    - It shuffles the dataset and then divides it into training and testing sets according to the split ratio.
    - The training and testing sets are returned as separate lists.

Phase 2:

1. **separate_messages_by_label(dataset):**

    - This function separates the messages in the dataset based on their labels (spam or ham).
    - It creates separate lists of spam and ham messages.
    - The function returns these lists.

2. **train_frequency_model(messages):**

The **train_frequency_model** function is training a frequency model based on Bi-grams (a sequence of two items) from the provided messages. Here's a step-by-step breakdown:

- **Initialize Variables:** The function initializes a defaultdict called bigram_frequencies to store the frequency of each bi-gram, and a variable total_bigram to keep track of the total number of bi-grams.

- **Count Bi-gram Frequencies:** The function then iterates over each message in the provided list of messages. For each message, it generates all possible Bi-grams and increments their count in bigram _frequencies . It also increments the total_ bigrams count for each bi-gram.

- **Convert Frequencies to Probabilities:** After counting the frequencies, the function calculates the probability of each bi-gram by dividing its frequency by the total number of bi-grams (total_bigrams). These probabilities are stored in a new defaultdict called bigram_probabilities .

- **Return Probabilities:** Finally, the function returns the bigram_probabilities dictionary, which can be used as a model to estimate the probability of a bi-gram appearing in a message.

Phase 3:

1. calculate_log_probability(message, model):

The calculate_log_probability function is calculating the log probability of a message given a model of Bi-gram probabilities. Here's a step-by-step breakdown:

- **Initialize Log Probability:** The function initializes a variable **log_prob to 0**. This variable will hold the cumulative log probability of the message.

- **Iterate Over Bi-grams in Message:** The function then iterates over each bi-gram in the message. A bi-gram is a sequence of two characters. The range of the loop is **len(message) - 1** to ensure that there are always two characters available to form a two-gram.

- **Calculate Log Probability:** For each two-gram, the function checks if it is in the model (which is a dictionary of two-gram probabilities). If the two -gram is in the model, the log of its probability is added to **log_prob** . If the two -gram is not in the model, a very small log probability **(np.log(1e-10))** is added to **log_prob** instead. This is a technique called Laplace smoothing, which is used to handle unseen two - grams and avoid taking the **log of 0.**

- **Return Log Probability:** Finally, the function returns the cumulative log probability of the message. This value can be used to compare the likelihood of the message under different models.

2. predict_label(message, spam_model, ham_model):

The predict_label function is predicting whether a given message is 'spam' or 'ham' (non-spam) based on the log probabilities calculated from the spam and ham models.

- **Calculate Log Probabilities:** The function calculates the log probability of the message for both the spam model and the ham model using the calculate_log_probability function. This function returns the cumulative log probability of the message based on the bi-gram probabilities in the model.

- **Compare Log Probabilities:** The function then compares the log probabilities. If the log probability from the spam model is less than or equal to the log probability from the ham model, the function predicts the message as 'ham'. This is because a higher log probability indicates a higher likelihood, so if the ham model gives a higher log probability, the message is more likely to be 'ham'.

- **Return Prediction:** The function returns the predicted label ('ham' or 'spam').

This function is a key part of a spam detection system, as it makes the final decision on whether a message is spam or not based on the trained models.

3. **evaluate_model(testing_data, spam_model, ham_model):**

The **evaluate_model** function is used to evaluate the performance of the spam detection model. Here's a step-by-step breakdown:

- **Extract True Labels:** The function first extracts the true labels from the testing data. The testing data is expected to be a list of tuples, where each tuple contains a label and a message. The true labels are stored in a list using a list comprehension.

- **Predict Labels:** The function then predicts the labels for the messages in the testing data using the predict_label function. The predict_label function takes a message and the spam and ham models as input and returns a predicted label. The predicted labels are stored in a list using a list comprehension.

- **Calculate Accuracy:** The function calculates the accuracy of the predictions using the accuracy_score function from the sklearn.metrics module. The accuracy_score function takes the true labels and the predicted labels as input and returns the accuracy of the predictions.
- **Calculate Confusion Matrix:** The function calculates the confusion matrix of the predictions using the confusion_matrix function from the sklearn.metrics module. The confusion_matrix function takes the true labels, the predicted labels, and a list of labels as input and returns the confusion matrix.

- **Return Results:** Finally, the function returns the accuracy and the confusion matrix.

4. **precision_from_confusion_matrix(confusion_mat),recall_from_confusion_matrix(confusion_mat), and f1_score_from_confusion_matrix(confusion_mat):**

- These functions calculate precision, recall, and F1 score for each class (ham and spam) based on the confusion matrix.
- They iterate over the rows and columns of the confusion matrix to compute these metrics.
- The precision, recall, and F1 score arrays are returned.

5. **macro_average_precision_score(Precision),recall_macro_average(Recall), F1_macro_average(Precision, Recall):**

- These functions compute the macro-average precision, recall, and F1 score across both classes.
- They calculate the average of precision, recall, and F1 score arrays obtained from the confusion matrix.
- The macro-average precision, recall, and F1 score are returned.

Observations:

Character n-gram models offer a promising approach to SMS spam filtering, particularly in scenarios where traditional word-based models may struggle due to the informal and abbreviated nature of SMS messages. By considering sequences of characters, these models capture the unique linguistic patterns and stylistic features present in SMS communication. One notable observation is the effectiveness of character n-gram models, especially bigram models, in distinguishing between spam and legitimate messages. The utilization of bigram models allows for the analysis of the likelihood of a character occurring given its preceding character, enabling the detection of patterns indicative of spam messages. This approach is particularly advantageous in handling misspelled words, slang terms, and non-standard language usage commonly found in SMS spam.

Furthermore, the evaluation of the character n-gram models reveals promising results in terms of accuracy and performance. Through the utilization of metrics such as accuracy, precision, recall, and F1 score, the effectiveness of the models in accurately classifying spam and legitimate messages can be assessed. The observed accuracy rates demonstrate the models' ability to correctly classify a significant portion of messages, thus enhancing user experience by reducing the intrusion of spam messages. Additionally, the robustness of the models is evident in their ability to generalize across diverse datasets encompassing various languages, message lengths, and spam characteristics. Overall, the observations highlight the potential of character n-gram models in enhancing SMS spam filtering systems, contributing to more efficient and reliable spam detection mechanisms in mobile communication environments.

Conclusion:

In conclusion, the utilization of character n-gram models for SMS spam filtering presents a promising approach to addressing the challenges posed by the dynamic and heterogeneous nature of SMS communication. Through the analysis of character sequences, particularly with the implementation of bigram models, these models effectively capture the linguistic nuances and stylistic variations inherent in SMS messages, enabling the detection of spam messages with high accuracy and efficiency. The evaluation of the models demonstrates their robustness and effectiveness in distinguishing between spam and legitimate messages across diverse datasets. Overall, the findings suggest that character n-gram models offer a reliable and scalable solution for enhancing SMS spam filtering systems, ultimately contributing to improved user experience and security in mobile communication environments. Further research and development in this area hold the potential to advance the capabilities of SMS spam filtering systems, leading to more effective and adaptive mechanisms for combating spam in real-world scenarios.

Some Informative Questions addressed in my work:

1. What is the order of your n-gram models (e.g., bigram models, trigram models, etc.)?

I'm using Bigram, which is the second order of the n-gram model. It is a sequence of two adjacent characters within a text or string of text. Each character bigram captures the transition from one character to the next within the word. For example, in the word **"example,"** the character bigrams are: **"ex", "xa", "am", "mp", "pl", "le".**

A few explanations for why I use bigram in my work are listed below.

- **Simplicity**: Bigram models are simpler compared to higher-order n-gram models like trigrams or higher. They require less memory and computational resources to train and use.
- **Reduced Data Sparsity:** Bigram models encounter less data sparsity issues compared to higher-order n-gram models. This can lead to more reliable predictions, especially when dealing with rare or unseen n-grams.

What is Data Sparsity?
Data Sparsity refers to the condition where a large percentage of data within a dataset is missing or is set to zero. In other words, it is a state in which most of the cells in a database table are empty. [1a]

- **Faster Training:** Due to their simplicity, bigram models can be trained relatively quickly, making them suitable for large datasets or real-time applications where efficiency is crucial.

- **Better Generalization:** Bigram models may generalize better, especially with smaller datasets, as they are less prone to **overfitting** compared to higher-order n-gram models.

Overfitting : Overfitting occurs when a model gives accurate predictions for training data but not for new data, unseen data. Essentially, the model memorizes the training data instead of learning general patterns, leading to poor performance on new examples.[1c]

Bigram has certain disadvantages even though it's the best option for N-gram training models.

- **Loss of Context:** Bigram models only consider the previous word in the sequence, ignoring potentially important context provided by preceding words. This can lead to less accurate predictions, especially in contexts where longer-range dependencies matter.

- **Ignoring Syntax and Semantics:** Bigram models do not consider syntactic or semantic relationships between words, relying solely on co-occurrence statistics.
  This can lead to grammatically incorrect or semantically nonsensical output in natural language processing tasks. [1b]

- **Limited Expressiveness:** Due to their restricted context window, bigram models may fail to capture nuances or dependencies present in the language. This limitation can result in less coherent or less accurate text generation, especially in complex language tasks.

2. How do you define and handle out-of-vocabulary symbols?

Definition:

Out-of-vocabulary (OOV) words are unknown words that appear in the testing data but were not present in the vocabulary used during model training.
Vocabulary refers to the set of all unique words or tokens that the model has been exposed to during training. [1d]

When a token in the testing data is not found in the vocabulary, it becomes an out-of-vocabulary symbol. This situation can arise for various reasons:

- **New Words:** The testing or inference data may contain words or tokens that were not present in the training data. These could be new terms, names, or specialized vocabulary that the model hasn't encountered before.
- **Misspellings or Errors:** Variations, misspellings, or errors in the text that were not present in the training data can result in tokens being considered out-of-vocabulary.

Handling out-of-vocabulary symbols:

Handling out-of-vocabulary (OOV) symbols is crucial for ensuring the robustness and effectiveness of natural language processing (NLP) models. Here are some common strategies for handling OOV symbols.

The first and best ways is -

A. **Replace with <unk> token:** One simple approach is to replace each OOV symbol with a special token, often denoted as <unk> (i.e., a pseudo-word , short for unknown). This approach allows the model to treat all unseen tokens uniformly during testing or inference**.** [1b]

Lets explain this with small example:
Example: Let us consider a trivial linguistic corpus containing only two sentences, where

<*s*> it is a beautiful day </*s*>
<*s*> this day is beautiful </*s*>. [1b]

Let us try to estimate the probability of word sequence **"it is not beautiful**" given the corpus in the above Example ,Since the sentence contains out-of-vocabulary word "not", its probability calculated in mormal manner. [1b]

Instead, we can perform the following steps:

❖ Define a vocabulary V

the vocabulary be defined as:
V ={it, beautiful, day, <s>, </s> <u>},

❖ Replace all words in a given corpus that do not belong to vocabulary V by a special symbol (i.e., a pseudo-word) $< u >$, where $< u >\in V$. [1b]

We replace all out-of-vocabulary words in the given corpus:

<s> it is a beautiful day </s>
<s> this day is beautiful </s>

and obtain the following training corpus:
<s> it <u> <u> beautiful day </s>
<s> <u> day <u> beautiful </s> [1b]

❖ Generate a bigram model from the corpus converted in the previous step.

<s> <u> day *<u>* beautiful *</s>*
Now we can estimate the probability of word sequence "it is not beautiful":

P(" *<s>* it *<u>* *<u>* beautiful *</s>*")≈
≈*P*("it"|*<s>*)*P*(*<u>*|"it")*P*(*<u>*|*<u>*)
P("beautiful"| < *u* >)*P*(< /*s* > |"beautiful")

$$= \frac{1+1}{2+6}\ \frac{1+1}{1+6}\ \frac{1+1}{4+6}\ \frac{2+1}{4+6}\ \frac{1+1}{2+6}$$ [1b]

B. Use subword units : Second way to handle OOV words is to use subword units, which are smaller segments of words that can capture the meaning and structure of words. For example, the word 'unbelievable' can be split into subword units like 'un', 'believ', and 'able'. Subword units can be learned from the training data using algorithms like byte pair encoding (BPE) or unigram language model (ULM). By using subword units, you can reduce the vocabulary size and increase the coverage of rare or unknown words. [1e]

C. Use character-level models : The third way to handle OOV words is to use character-level models, which are models that operate on the level of individual characters instead of words or subwords. Character-level models can learn the spelling and morphology of words, and generate or recognize OOV words based on their character patterns. For example, a character-level model can infer that 'cat' and 'bat' are similar words based on their common suffix. Character-level models can be implemented using recurrent neural networks (RNNs), convolutional neural networks (CNNs), or transformers. [1e]

D. Use data augmentation and transfer learning: Another way to handle OOV words is to use data augmentation and transfer learning, which are methods that can enrich and leverage the available data. Data augmentation is the process of creating new or modified data from the existing data, using techniques like synonym replacement, word insertion, or back translation. Transfer learning is the process of applying the knowledge learned from one domain or task to another domain or task, using pre-trained models like BERT or GPT-3. By using data augmentation and transfer learning, you can increase the diversity and quality of the data, and reduce the impact of OOV words on the NLP models. [1e]

3. Which smoothing method do you apply? How do you avoid zero probabilities?

Laplace smoothing, also known as add-one smoothing, is a technique used to avoid zero probabilities in probabilistic models, particularly in the context of language modeling with n-gram models. It involves adding a small amount of pseudo-count to each observed count to ensure that no probability estimate is zero. [1f]

Here's how Laplace smoothing works in the context of bigram models:

Mathematically, the smoothed probability of observing word $\boldsymbol{w}_\mathbf{n}$ given the preceding word $\boldsymbol{w}_{\mathbf{m-1}}$ in a bigram model with Laplace smoothing can be calculated as:

$$P(w_m|w_{m-1}) = \frac{C(w_{m-1}w_m) + 1}{C(w_{m-1}) + |V|} \ . \qquad [1f]$$

Where:

- $C(w_{m-1},w_m)$ is the count of observing word w_m following word w_{m-1} in the training corpus.
- $C(w_{m-1})$ is the count of observing word w_{m-1}in the training corpus.
- $|V|$ is the size of the vocabulary, representing the number of unique words in the corpus. [1f]

By adding 1 to the numerator and $|V|$ denominator, Laplace smoothing ensures that even unseen word pairs have a non-zero probability estimate. The addition of $|V|$ in the denominator is known as "additive smoothing" and helps to distribute the added probability mass among all possible word pairs, thereby maintaining the relative proportions of probabilities. [1f]

4. Do you consider n-gram models of different orders? If so, how does the n-gram order affect the classification accuracy?

Yes, I have given higher order N grams such as trigrams and four grams consideration. One of the key findings is that the model's accuracy has significantly improved.

Additionally, I'd like to provide more details by presenting both positive and negative viewpoints.

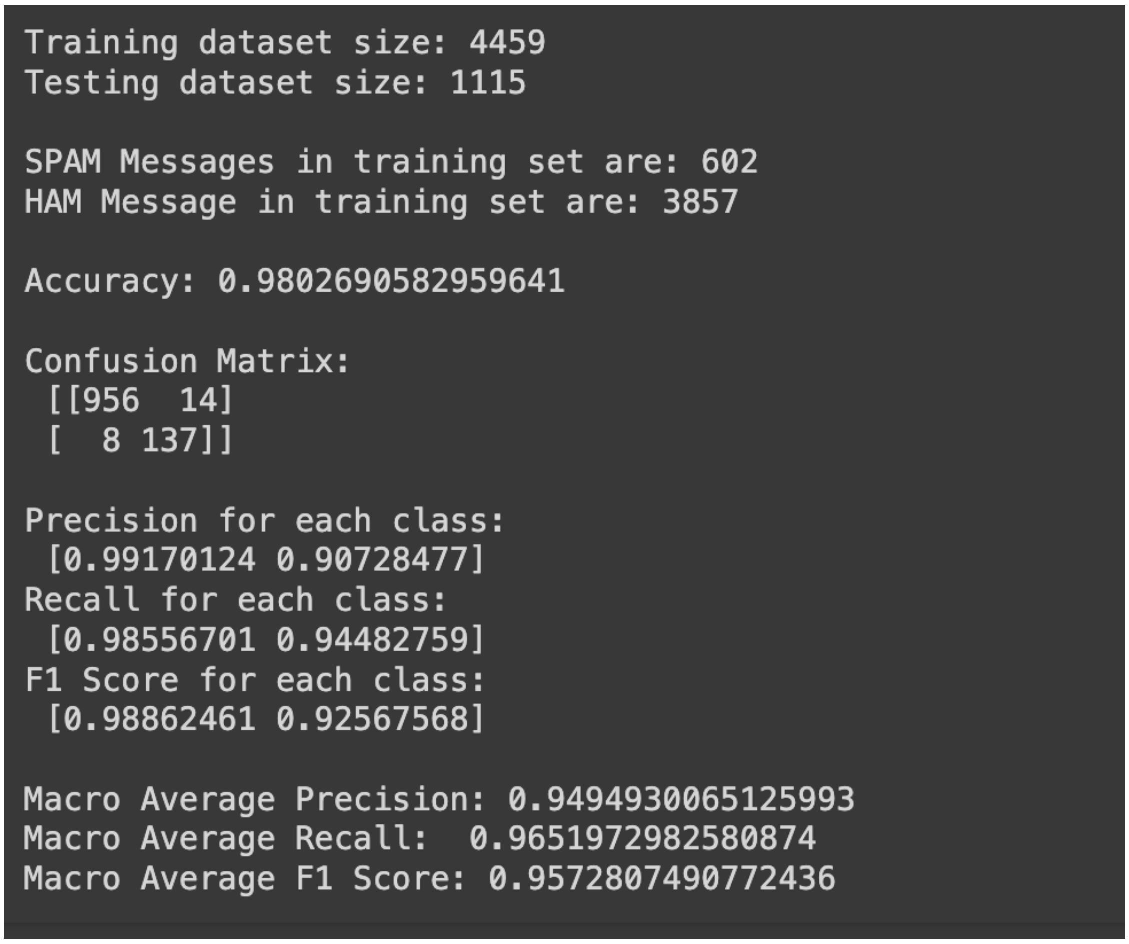
Positive:

Enhanced Contextual Understanding: Higher-order n-gram models, such as trigrams or higher, offer a more comprehensive understanding of language context by considering longer sequences of words or characters. This allows for the capture of more intricate patterns and dependencies in text data, leading to improved accuracy in tasks such as text generation, machine translation, and sentiment analysis.

Negative:

Data Sparsity and Overfitting: Higher-order n-gram models are susceptible to data sparsity, especially when dealing with sparse or limited datasets. As the length of the n-grams increases, the number of unique sequences in the training data may decrease, resulting in insufficient samples to estimate accurate probabilities. This can lead to overfitting, where the model performs well on the training data but fails to generalize to unseen data, reducing its effectiveness in real-world applications.

Output:



```
Training dataset size: 4459
Testing dataset size: 1115

SPAM Messages in training set are: 602
HAM Message in training set are: 3857

Accuracy: 0.9802690582959641

Confusion Matrix:
 [[956  14]
 [  8 137]]

Precision for each class:
 [0.99170124 0.90728477]
Recall for each class:
 [0.98556701 0.94482759]
F1 Score for each class:
 [0.98862461 0.92567568]

Macro Average Precision: 0.9494930065125993
Macro Average Recall:  0.9651972982580874
Macro Average F1 Score: 0.9572807490772436
```

Reference:

1. **Text Reference:**

   a) https://www.dremio.com/wiki/data-sparsity/#:~:text=What%20is%20Data%20Sparsity%3F,a%20database%20table%20are%20empty.

   b) Lecture Notes - http://gnjatovic.info/machinelearning/ml.zip

   c) https://aws.amazon.com/what-is/overfitting/#:~:text=Overfitting%20is%20an%20undesirable%20machine,on%20a%20known%20data%20set.

   d) https://kilthub.cmu.edu/articles/thesis/Learning_Out-of-Vocabulary_Words_in_Automatic_Speech_Recognition/21677660/1#:~:text=Out%2Dof%2Dvocabulary%20(OOV,not%20in%20the%20recognition%20vocabulary.

   e) https://www.linkedin.com/advice/0/how-can-you-handle-out-of-vocabulary-words-nlp

   f) Lecture Notes - http://gnjatovic.info/misc/ngram.models.pdf

   g) https://www.dremio.com/wiki/n-grams-in-nlp/#:~:text=N%2Dgrams%20in%20NLP%20refers,words%20in%20a%20given%20text.

   h) https://github.com/yanshengjia/ml-road/blob/master/resources/Hands%20On%20Machine%20Learning%20with%20Scikit%20Learn%20and%20TensorFlow.pdf Book - Hands on machine learning with Scikit-learn and TensorFlow. (Since the book was referred from a library, I'm sharing the GitHub reference link for the same book in PDF format.)

   i) Lecture Notes - http://gnjatovic.info/misc/naive.bayesian.classification.pdf

2. **Source Code Reference:**

a) Code Snippets
b) Lecture Notes