CMSC 162 Artificial Intelligence
Machine Problem 2

Submitted by:
Chua, Mary Elizabeth E.
Nisay, Deiondre Judd V.

July 14, 2022

I. Introduction

In a world where the digital space is relentlessly used for communication, extensive implications on people's privacy and safety have been compromised. With the growth of technological communications, devious advancements are brought about in order to manipulate victims into giving up their personal information, or just bring unwanted messages for marketing and advertising purposes. This is especially apparent in the conduct of contact tracing where people's contact info are collected for the purpose of health protocols in the COVID-19 pandemic, however this collection of information might fall into the hands of irresponsible people and end up within the reach of scammers and within unethical marketing strategies. However, even before the pandemic, this type of issue has always been prominent; spam messages may be one of the most used types of scamming devices ever since emails and text messages were introduced. However, with this, there have been numerous workarounds for dealing with spam messages; one of which is spam filtering. In this paper, the authors will introduce a machine learning program using Naive Bayes Classification for the purpose of training the said classifier to detect whether or not a given message is spam. In order to do this, the program uses a general data set, given that each line are separate sample messages, and that the first word of every line is its label, "ham" or "spam", meaning that it is not a spam message or that it is a spam message. The program tests the given training data set and the test data set. Afterwards, it will print out calculations about the data concerning its own accuracy. The next part of this paper, Methodology, contains further discussion about the implementation of the algorithms and calculations.

II. Methodology

As stated previously, the program splits the general data set into two parts, the training set and the testing set. The splitting of the general data set into the two said sets will be the choice of the user, and they must keep them in separate files. However, the general rule of thumb is that the training set must be 70% of a part of the data set while the test set be the remaining 30%. So, if a data set with 1000 records will be used, then we shall get 700 sample messages for its training set, and the remaining for its test set. Afterwards, the program uses a Naive Bayes Classification method in order to process the test data. To do this, the training set is read from the given file, all unique words will then be saved into a linked list called `vocabularyV`. The program is made to take in any file, and

therefore must dynamically take in N amounts of data, hence linked lists are the main data structure implemented. In `vocabularyV`, we also have each of the word's statistics or counts, such as the number of times the word has appeared in "ham" labeled messages, number of times the word has appeared in "spam" labeled messages, and its conditional probabilities (p(word|Spam) and p(word|Ham)). After completing the transversal throughout the train set, we would have the complete set of parameters for the Maximum Likelihood Estimation.

Likelihood of the message to be spam

$$P(msg = spam) = \frac{Count(msg=spam)}{Total\ \#\ of\ messages}$$

Likelihood of the message to be ham

$$P(msg = ham) = \frac{Count(msg=ham)}{Total\ \#\ of\ messages}$$

Percentage of the instances that the word appeared in spam messages

$$P(word = "word"|msg = spam) = \frac{Count(word=="word"\&\&\ msg=spam)}{Count(word=="word")}$$

Percentage of the instances that the word appeared in ham messages

$$P(word = "word"|msg = ham) = \frac{Count(word=="word"\&\&\ msg=ham)}{Count(word=="word")}$$

Then with these, we can use the **Naive Bayes Model** for the probability that the message is spam:

$$P(word_1,...,word_N|msg = spam) = \prod_{i=1}^{N} P(word_i|msg = spam)$$

And the probability that the message is ham:

$$P(word_1,...,word_N|msg = ham) = \prod_{i=1}^{N} P(word_i|msg = ham)$$

The higher probability will be the decision, P(<message>|msg=spam) < P(<message>|msg=ham) then the decision is "spam", else if P(<message>|msg=spam) > P(<message>|msg=ham) then the decision is "ham". Else if both probabilities are equal, then the training set probably needs more data. Detailed description about the algorithm will be discussed in the next subchapters.

A. Pseudocode

Before the authors start creating the code, the pseudocode was made in order to understand the general idea of how the algorithm should work. The program revolves around the linked list **`vocabularyV.`** Below is the visualization of how our vocabulary of unique words found in training the classifier and the count variables are being stored in the linked list:

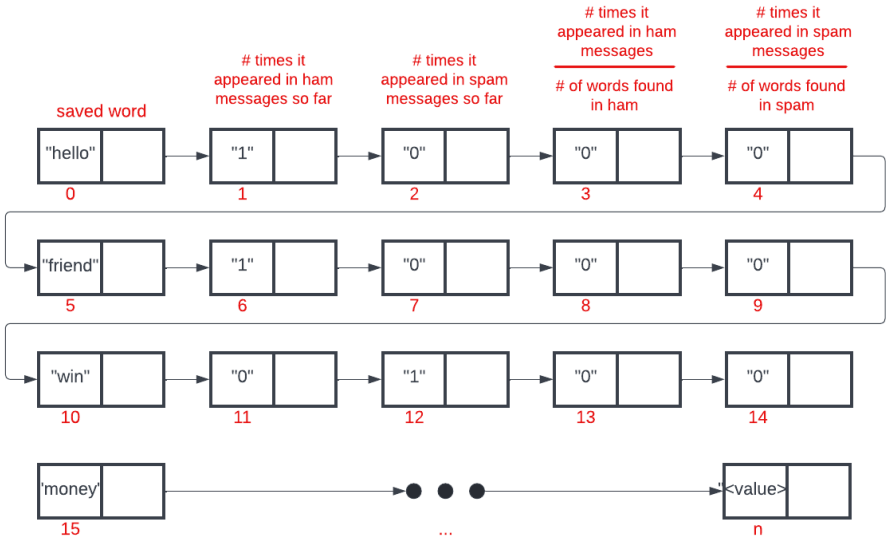

figure 1. Visualization of the Nodes in **vocabularyV**

At the first run of the training part of the program, we read the data set per word. Suppose we have the first few words as "ham hello friend \n spam win money". The figure above shows how each word is read and classified, the first node saves the actual word and every n+5th node after that is the next word saved in the vocabulary. Similarly, the 2nd node saves the number of times it appeared in ham messages so far and so does every n+5th node after that as it increments with every instance its word is found in ham messages. The same goes with the 3rd node. The 4th and 5th node is firstly initialized as 0 since we can only acquire this value after we count all of the instances of every unique word. To better understand how this algorithm works, below is the pseudocode for the program:

PROGRAM PSEUDOCODE

```
// ALGORITHM FOR TRAINING THE CLASSIFIER

initialize the vocabularyV linked list
get the file for the training dataset
(for) read training dataset until empty

      if the current line is empty
            break out of the for loop
      else if the current line is valid
            if the first word of the line is "ham"
                  then we label this as "ham"
                  increment the total number of ham messages
            else if "spam"
                  then we label this as "spam"
```

```
                increment the total number of spam messages

      if there are next words
            if the word already exists in vocabularyV[]
                  then if the message is ham
                        then increment ham count of the current word
                        and increment total ham words
                  else if the message is spam
                        then increment spam count of the current word
                        and increment total spam words
            else if word does not exist
                  create a new set of node in vocabularyV[] linked list
                  initialize the following:
                        add the word in the linked list vocabularyV //n1
                        if it's currently ham
                              add "1" in vocabularyV //n2
                              add "0" in vocabularyV //n3
                        else it's currently spam
                              add "0" in vocabularyV //n2
                              add "1" in vocabularyV //n3
                        add "0" in vocabularyV //n4
                        add "0" in vocabularyV //n5

if there are no detected ham and spam words
      print Invalid training data
else
      for every word that is in vocabularyV
      on the 3rd node after the word, # of ham instances/# of total ham words
      on the 4th node after the word, # of spam instances/# of total spam words

// ALGORITHM FOR TESTING THE CLASSIFIER

get the file for the test dataset
(for) read test dataset until empty
      read first label (if spam or ham)
      for every (next) word in the line
            look up the word in the vocabularyV
            ham probability multiplied by itself multiplied by the 3rd
      node after the word (# of ham instances/# of total ham words)
            spam probability multiplied by itself multiplied by the 4th
      node after the word (# of spam instances/# of total spam words)

      ham probability multiplied by the total number of ham messages
over the overall number of records
```

```
      spam probability multiplied by the total number of spam messages
over the overall number of records

if spam probability < ham probability
      then the message is not spam
else spam probability > ham probability
      then the message is spam
else spam probability = ham probability
      then not able to make a decision

calculate statistics such as precision and recall
```

B. Data Structure and Variables

Data structures used are linked lists, specifically, the linked list in the code is **`vocabularyV`** this is the main structure that deals with saving the "word sets", we reference this frequently as it acts as the lookup table. Furthermore, the following are the list of variables used in the code:

//Training Algorithm

1. Integer
    a. int totalHamCount=0;
        - Total number of ham messages
    b. int totalSpamCount=0;
        - Total number of spam messages
    c. int myTesting = 0;
        - Decision of the user if they want to test another set of data
    d. int hammedWords=0;
        - Total number of words found in ham messages
    e. int spammedWords=0;
        - Total number of words found in spam messages
    f. int wordSets=0;
        - Number of words in **`vocabularyV`**
    g. int temp = 0;
        - Holder variable/identifies if message has spam or ham label
    h. int trainingSet = 3901;
        - Holds the number of records in the training set
    i. int x = 0;
        - Iterator for the function saving every unique word into **`vocabularyV`**
    j. int wordExists = 0;
        - Holder variable/identifies if the current word is already existing in **`vocabularyV`**
    k. int toParse = 0;
        - Takes the string value from the word set and parses it into an integer

2. Double
    a. double totalCount = 0;
        - Total number of records
    b. double hamMessagesProbability = 0;
        - Number of ham messages over the total number of records
    c. double spamMessagesProbability = 0;
        - Number of spam messages over the total number of records
    d. double hamProbability=0;
        - The number of times that the word appeared in ham messages over the total number of words appeared in ham messages
    e. double spamProbability=0;
        - The number of time that the word appeared in spam messages over the total number of words appeared in spam messages

3. String
    a. String label = "";
        - Labels the message if it is a ham or spam
    b. String filename = "";
        - Takes in the file name of the dataset

4. Others
    a. filename = read.nextLine();
        - Reads the filename from the user's input
    b. File f = new File(filename);
        - An object specifying the directory or filename
    c. Scanner s = new Scanner(new File(filename));
        - Scans the file

//Testing Algorithm

1. Double
    a. double TP = 0;
        - TP (true positive): number of spam messages classified as spam
    b. double TN = 0;
        - TN (true negative): number of ham messages classified as ham
    c. double FP = 0;
        - FP (false positive): number of ham messages misclassified as spam
    d. double FN = 0;
        - FN (false negative): number of spam messages misclassified as ham
    e. double precision = 0;
        - Number of spam messages classified as spam over the sum of number of spam messages classified as spam and number of ham messages misclassified as spam

f. double recall = 0;
   - Number of spam messages classified as spam over the sum of the number of spam messages classified as spam and number of spam messages misclassified as ham

2. String
   a. String toTestFile = "";
      - Saves the filename of the testing data set

3. Others
   a. toTestFile = read.nextLine();
      - Reads the user input
   b. File t = new File(toTestFile);
      - An object specifying the directory or filename
   c. Scanner q = new Scanner(new File(toTestFile));
      - Scans the file

C. Code

To discuss further, some of the important implementations and technicalities in written code will be explained via code snippets.

First, we train our program by inputting a text file containing different messages that all start with either “ham” or “spam”. Further details will be explained in the later part of this.

```
System.out.println("\nEnter Training Data File with extension (i.e.
train.txt)");
filename = read.nextLine();
File f = new File(filename);
```

For the error handling to determine whether a file exists or not, we implement this piece of code that loops every time an invalid file name is inputted. If it existed already, then it will simply leave the while-statement.

```
while (!okayFile) {
            if (f.exists()) {
                okayFile=true;
            }
            else{
                System.out.println("\nFile does not exist, please try
again...");
```

After a file has been confirmed valid, it will then be scanned by the program per line. It is set to only take a certain amount of lines by hardcoding it. If the iterations exceed the number of lines set to learn, it will break out of the while loop. On the other hand, if the program iterates less than the limit and it can't detect a next line, it will simply break out of the while loop which would conclude the training.

```
Scanner s = new Scanner(new File(filename));
while (x < trainingSet) {
    if (!s.hasNextLine()){
        break;
    }
    else{
        Scanner s2 = new Scanner(s.nextLine());
        String r1 = s2.next();
        if (r1.equals("ham") || r1.equals("Ham") || r1.equals("HAM")) {
            label = "ham";
            totalHamCount++;
            temp = 1;
            x++;
        }
      else if (r1.equals("spam")||r1.equals("Spam")||r1.equals("SPAM")) {
            label = "spam";
            totalSpamCount++;
            temp = 1;
            x++;
      }
```

The program will take the first word of the line which in the given training data, the first word on every line would start with "ham" or "spam". If it reads "ham", "Ham", or "HAM", it will pass "ham" to the variable label so that we won't have different variations of "ham" lexicographically. Same goes with lines that start with "spam", "Spam", and "SPAM" as it'll only pass the value "spam" to the label variable. After which, the number of ham lines or spam lines will increase which would essentially be equivalent to the number of ham or spam messages. Then temp will be assigned a value of 1 for later use and x will increment which signifies the program has read a new line.

Next, it will check two conditions; whether in the line it has scanned, there exists a next word, and if temp is equal to 1. The reason why temp was set to 1 was so that the training program will STRICTLY read lines/messages that are labeled as "ham" or "spam". The significance of doing so is that if the program reads a line that does not contain "ham" or "spam" in the beginning, it will not be able to determine if the set of words within the line are ham messages or spam messages. So, to save the program from such trouble, we decided to let the scanner skip the line as temp wasn't given a value of 1, therefore skipping the word listing process and proceeding to the next line.

```
while (s2.hasNext() && temp == 1) {
String r2 = s2.next();
if (wordSets != 0) {
    for (int i = 0; i < wordSets; i=i+5) {
        data = vocabularyV.get(i);
        if (r2.equals(data)) {
            if (label == "ham") {
                wordExists = 1;
                dataTemp = vocabularyV.get(i+1);
                toParse = Integer.parseInt(dataTemp) + 1;
                vocabularyV.set(i+1, String.valueOf(toParse));
                hammedWords++;}
            else{
                wordExists = 1;
                dataTemp = vocabularyV.get(i+2);
                toParse = Integer.parseInt(dataTemp) + 1;
                vocabularyV.set(i+2, String.valueOf(toParse));
                spammedWords++;}
```

As it enters the while loop, it will start by reading the next word after “ham” or “spam”. Of course, we’ll do a simple if-statement that will check if there are words already in the vocabulary V linked list. If there are no words in the linked list, it will skip the checking process and proceed to the listing process as a new instance of the word. However, if there exists one word or more in the linked list, the program will then check if the word that the scanner has read already exists in the vocabulary V linked list. If the word does exist in the linked list, it will check if the label has a value of “ham” or “spam”. If it has a value of “ham”, it will set the checker on whether the word exists or not to 1. Then it’ll retrieve the next node which would contain the instances of the word appearing in ham messages. We will have the program transform the instances into an integer value for computation. We’ll add 1 to it and return it back to string form, followed by overwriting the value in the node after the word. Then we increase the total number of words in ham messages. The same goes for when the label has a value of “spam”, except instead of getting and overwriting the node next to the word, the program will get and overwrite the node that is 2 nodes away from the word, followed by increasing the total number of words in spam messages.

To have a better understanding on how the vocabulary V linked list stores the values, here’s an explanation. Each set of nodes, labeled as “wordSets”, contains 5 nodes. The first node in the set contains the word. The second node contains the number of instances the word appeared in ham messages. The third node contains the number of instances the word appeared in spam messages. The fourth node contains the ham probability while the fifth node contains the spam probability which will be computed later, but will be initialized as 0 first. This way or grouping nodes for descriptions of the word is the reason why we have for statements that checks each word

incrementing by 5 and why we always increment our wordSets value by 5 every unique instance of a word.

In the next section of the code, this will list down every unique instance of a word. It will first check if wordExists is equal to 0, which means it checks if the word has not appeared in the vocabulary V linked list. If it has appeared, it will skip this process and proceed to read the next word, if the next exists of course. On the other hand, if the instance of the word is unique, the program will add the word in the first position of the set of nodes, or word set, mentioned before. If the label for the word set for the line before is set to "ham", then the second and third nodes in the word set will be given values of 1 and 0, respectively, followed by incrementing the total number of words in ham messages by 1.

```
if (wordExists == 0) {
    vocabularyV.add(wordSets, r2);
    if (label == "ham") {
        vocabularyV.add(wordSets+1, "1");
        vocabularyV.add(wordSets+2, "0");
        // vocabularyV.add(wordSets+1, "2");        // for smoothing
        // vocabularyV.add(wordSets+2, "1");        // for smoothing
        hammedWords++;
        // hammedWords++;                          // for smoothing
        // spammedWords++;                         // for smoothing
    }
    else{
        vocabularyV.add(wordSets+1, "0");
        vocabularyV.add(wordSets+2, "1");
        // vocabularyV.add(wordSets+1, "1");        // for smoothing
        // vocabularyV.add(wordSets+2, "2");        // for smoothing

        spammedWords++;
        // spammedWords++;                         // for smoothing
        // hammedWords++;                          // for smoothing
    }
    vocabularyV.add(wordSets+3, "0");
    vocabularyV.add(wordSets+4, "0");
    wordSets = wordSets + 5;}
```

However, that's how it would work without smoothing. With smoothing, we set the second and third nodes in the word set with values of 2 and 1, respectively, followed by incrementing the total number of words in ham messages by 2 and total number of words in spam messages by 1. This means that we're adding a virtual count of 1 on every instance of the word in ham messages and

spam messages. The same goes for the other case when the label was given a value of “spam”. The difference would be that for the case of no smoothing, the second and third nodes will be given values of 0 and 1, respectively, followed by incrementing the total number of words in spam messages by 1. On the other hand, when smoothing is applied when the label is “spam”, the second and third nodes will be given values of 1 and 2, respectively, followed by incrementing the total number of words in spam messages by 2 and total number of words in ham messages by 1. After which, we’ll initialize the fourth and fifth nodes with 0 in which these nodes contain the ham probability and spam probability of the word, respectively. Lastly, we increment the value of wordSets by 5 so that the next unique instance will follow the same format.

Of course, after the inner while loop, we set the wordExists variable to 0 so that it could go through the same checking process on whether the word exists in the vocabulary V linked list or not. As for the outer loop, temp will be given a value of 0 again so that the function of checking if the first word of the line would contain “ham” or “spam”, and if it doesn’t contain, it would skip the entire listing process

```
            wordExists = 0;
                }
            }
            temp = 0;
        }
```

After listing the words, their number of instances in both ham messages and spam messages, the program will then compute the ham probabilities and spam probabilities of each word. However, we have conditional statements which would check if the training data that the user has inputted is valid or not. If the total number of words in ham messages or spam messages are equal to zero, or even the number of ham messages of spam messages are equal to zero, this would signify that there is insufficient data for training the program, in which the myTesting value will be given a 1 which would not allow the program to enter the testing part of the program.

```
if (hammedWords == 0 || totalHamCount == 0) {
    System.out.println("Insufficient Ham Messages for training data");
    myTesting = 1;
}
else if (spammedWords == 0 || totalSpamCount == 0) {
    System.out.println("Insufficient Spam Messages for training data");
    myTesting = 1;
}
else{
    for (int j = 0; j < wordSets; j=j+5) {
        // ham probability
        dataTemp = vocabularyV.get(j+1);
```

```
        hamProbability = Double.parseDouble(dataTemp)/hammedWords;
        vocabularyV.set(j+3, String.valueOf(hamProbability));

        //spam probability
        dataTemp = vocabularyV.get(j+2);
        spamProbability = Double.parseDouble(dataTemp)/spammedWords;
        vocabularyV.set(j+4, String.valueOf(spamProbability));
    }
}
```

However, all the conditions weren't met and the training data is valid, the program will go through a for loop retrieving every second node of each word set, which contains the number of instances the word has appeared in ham messages, and dividing it with the total number of words in ham messages. The result will be the ham probability in which the program will overwrite the fourth node to the ham probability computation. The program will also be retrieving every third node of each word set, which contains the number of instances the word has appeared in spam messages, and dividing it with the total number of words in spam messages. The result will be the spam probability in which the program will overwrite the fifth node to the spam probability computation.

With this, the word set is now complete because it contains most of the elements needed for testing such as the word, the number of instances it appeared in both ham messages and spam messages, and the ham probability and spam probability.

After completing the word set, we get the total number of messages by adding the number of ham messages with the number of spam messages. Then, to get the ham messages probability, which is the probability on whether a message is likely to be considered "ham", we divide the number of ham messages by the total number of messages. On the other hand, we divide the number of spam messages by the total number of messages in order to get the spam messages probability, which is the probability on whether a message is likely to be considered "spam".

```
totalCount = totalHamCount + totalSpamCount;
hamMessagesProbability = totalHamCount/totalCount;
spamMessagesProbability = totalSpamCount/totalCount;
```

Now, if the training data was valid, the value of the variable myTesting should remain 0 so that the program will be able to enter the testing part. Once it enters, the program asks the user to input the file that contains the testing data which follows the same format as the training data so that the program can determine the values of TRUE POSITIVES, TRUE NEGATIVES, FALSE POSITIVES, and FALSE NEGATIVES.

```
while (myTesting == 0) {
    String toTestFile = "";
```

```
    okayFile = false;

    System.out.println("\nEnter Test Data with extension (i.e.test.txt)");
    toTestFile = read.nextLine();
    File t = new File(toTestFile);
    while (!okayFile) {
        if (t.exists()) {
            okayFile=true;
        }
        else{
         System.out.println("\nFile does not exist, please try again...");
         System.out.println("\nEnter Test Data File with extension (i.e.
test.txt)");
         toTestFile = read.nextLine();
          t = new File(filename);
        }
     }
```

Once the test data file has been inputted, the program will check if the file exists. If it doesn't, the program will ask the user for an input once again. This repeats until the user enters a file name that exists to leave the while loop.

Afterwards, the program will execute the similar methods of checking the lines and words of the training process. Most especially checking whether the next line exists, whether the first word in the line is "ham" or "spam", and setting the temp value to 1 if the first words are "ham" or "spam". A difference to note here is that checking for "ham" and "spam" will only be used to determine if the program's prediction is correct or not.

```
Scanner q = new Scanner(new File(toTestFile));

while (q.hasNextLine()) {
    temp = 0;
    label = "";
    double yesProbability = 1.000000000000000000;
    double noProbability = 1.000000000000000000;
    String spamOrHam = "";
    Scanner q2 = new Scanner(q.nextLine());

    String z1 = q2.next();
    if (z1.equals("ham") || z1.equals("Ham") || z1.equals("HAM")) {
```

```
        label = "ham";
        temp = 1;}
    else if (z1.equals("spam") || z1.equals("Spam") || z1.equals("SPAM")){
        label = "spam";
        temp = 1;}
while (q2.hasNext() && temp == 1) {
    String q3 = q2.next();
    for (int k = 0; k < wordSets; k=k+5) {
        data = vocabularyV.get(k);
        if (q3.equals(data)) {
            dataTemp = vocabularyV.get(k+4);
            yesProbability = Double.parseDouble(dataTemp)*yesProbability;
            dataTemp = vocabularyV.get(k+3);
            noProbability = Double.parseDouble(dataTemp)*noProbability;
            break;
```

A difference from the training process is that it will always cycle through the vocabulary V linked list to see if a certain word read from the test data exists in the linked list. If the same word exists in the linked list, the program will get the fifth node of the word set which contains the spam probability of the word and multiplies it with the variable yesProbability, which was initially set as 1. Next, it gets the fourth node of the word set which contains the ham probability of the same word and multiplies it with the variable noProbability, which was initially set as 1. The yesProbability determines the probability of the message being a spam message while the noProbability determines the probability of the message being a ham message. This iterates every word in the line, therefore making the values of yesProbability and noProbability smaller.

After the entire line has been read, and the yesProbability and noProbability has been finalized for that line, it will be multiplied one last time, this time with the variables spamMessagesProbability and hamMessagesProbability, respectively.

```
yesProbability = yesProbability*spamMessagesProbability;
noProbability = noProbability*hamMessagesProbability;

if (yesProbability > noProbability){
    spamOrHam = "spam";
}
else{
    spamOrHam = "ham";
}

if (label.equals(spamOrHam)) {
```

```
    if (spamOrHam == "ham") {
        TN++;
    }
    else{
        TP++;
    }
}
else{
    if (spamOrHam == "ham") {
        FN++;
    }
    else{
        FP++;
    }
}
```

Once it has the values of the variables yesProbability and noProbability for the message in question, the program will now compare one over the other. If yesProbability is greater than noProbability, this means the program has determined that the given message is a spam message. However, if noProbability is greater than yesProbability, this means the program has determined that the given message is a ham message.

Afterwards, the program determines if its prediction is TRUE POSITIVE, TRUE NEGATIVE, FALSE POSITIVE, and FALSE NEGATIVE.

If the program's guess is the same as the actual classification of the message, it will then do another check. If the program's guess was "ham", the program will increment the value of TRUE NEGATIVE by 1. On the other hand, if the program's guess was "spam", the program will increment the value of TRUE POSITIVE by one.

However, if the program's guess is different from the actual classification of the message, it will then do another check. If the program's guess was ham, the program will increment the value of FALSE NEGATIVE by 1. On the other hand, if the program's guess was "spam", the program will increment the value of FALSE POSITIVE by 1.

To compute the values of precision and recall, we have to make sure that the divisor for both precision and recall should not be equal to zero. If the divisor of precision is not equal to zero, the value of precision can be computed by dividing TRUE POSITIVE over the sum of TRUE POSITIVE and FALSE POSITIVE

```
if ((TP+FP)!=0) {
    precision = TP/(TP+FP);
```

If the divisor of recall is not equal to zero, the value of recall can be computed by dividing TRUE POSITIVE over the sum of TRUE POSITIVE and FALSE NEGATIVE.

```
}
if ((TP+FN)!=0) {
    recall = TP/(TP+FN);
}
```

Afterwards, we print the values of TRUE POSITIVE, TRUE NEGATIVE, FALSE POSITIVE, and FALSE NEGATIVE out, together with the values of the precision and recall of the program.

After the testing, the program will ask the user if they would like to input another set of test data or terminate the program. An input of 1 will let the program loop back to the testing process to input another set of test data. An input of 0 will terminate the program.

```
Scanner myDecision = new Scanner(System.in);
System.out.println("Would you like to input another set of test data?");
System.out.println("1 - Yes");
System.out.println("0 - No");
```

Of course, there's error handling for this function. The code snippet below will only make it so that the program will only accept 1s and 0s for answers while any other answer will cause the program to display an error message in which the user will have to input their choice again. This will keep on repeating until a value of 1 or 0 has been inputted.

```
while (validLoop==0) {
    if (myDecision.hasNextInt()) {
        reRun = myDecision.nextInt();
        validLoop=1;
    }
    else{
        System.out.println("Error: Invalid input, try again");
        System.out.println();
        myDecision = new Scanner(System.in);
         System.out.print("Would you like to input another set of test
data?");
        System.out.println("1 - Yes");
        System.out.println("0 - No");
    }
```

```
    if (reRun == 1 || reRun == 0) {
        validLoop=1;
    }
    else{
        System.out.println("Error: Invalid input, try again");
        System.out.println();
        myDecision = new Scanner(System.in);
         System.out.print("Would you like to input another set of test
data?");
        validLoop=0;
     }
}
```

Once the user has chosen

III. Results and Discussion

The provided dataset is from UCI Machine Learning Repository by Tiago A. Almeida (talmeida ufscar.br) Department of Computer Science, Federal University of Sao Carlos (UFSCar), Sorocaba, Sao Paulo - Brazil. This has a total of 5574 records. As stated previously, the splitting of training and test data must constitute 70:30 of a part of the dataset. Since we have to test the algorithm 5 times, with and without smoothing, the whole dataset is split into 22 parts. five 3-parts for the five trials on the test data, and one 7-part for the training set of the data. This method of splitting data makes sure that the train and test data is at a 70:30 ratio and that every trial uses completely different records from each other. With this, we would have 1771 records for training and 749 records for testing 5 times, which optimizes the use of the whole dataset as it will total up to 5516.

Furthermore, after every train and test method, some statistical data is also calculated such as the true positive, true negative, false positive, false negative, precision, and recall. The table below shows the summary of the trial results:

Table 1. Summary of Results from 5 Test Runs With and Without Smoothing

Without Smoothing						
Iteration	TP (true positive)	TN (true negative)	FP (false positive)	FN (false negative)	Precision	Recall
1	91	647	2	19	0.978494623655914	0.8272727272727273
2	85	650	1	23	0.9883720930232558	0.7870370370370371
3	80	658	3	18	0.963855421686747	0.8163265306122449
4	77	659	2	21	0.9746835443037974	0.7857142857142857

5	67	665	3	24	0.9571428571428572	0.7362637362637363
With Smoothing						
Iteration	TP (true positive)	TN (true negative)	FP (false positive)	FN (false negative)	Precision	Recall
1	54	647	2	56	0.9642857142857143	0.4909090909090909
2	49	651	0	59	1	0.4537037037037037
3	48	658	3	50	0.9411764705882353	0.4897959183673469
4	36	660	1	62	0.972972972972973	0.3673469387755102
5	50	666	2	41	0.9615384615384616	0.5494505494505495

If the algorithm does not implement smoothing, the first iteration shows that we have 87.27% accuracy in detecting spam messages, 91 out of 110 spam messages were tagged as spam, while we have 99.69% accuracy in detecting ham messages, 647 out of 649 messages were tagged as ham. This then is related to the precision value, while the recall value accounts for how much spam messages were classified and misclassified, showing us how much it missed some of the actual spam messages. There seems to be a low variation in results, we have the averages of 80, 655.8, 2.2, 21, 97.25%, and 79.05% for the TP, TN, FP, FN, Precision, and Recall values, respectively, and standard deviation of 9, 7.26, 0.84, 2.55, 0.01, and 0.04. This means, on average, in every 749 messages that we receive, we would expect about 2 falsely classified spam messages, and 21 falsely classified ham messages. Which seems nonoptimal, since the goal is to completely eradicate the possibility of letting spam through the main inbox, and if this cannot be completely 0, then about 1% of the total messages would be fine, however this goal is not reached.

Next, we have the results for the test runs with smoothing. The first iteration shows that we have detected 49.09% of the spam messages, 54 out of 110 spam messages were correctly tagged as spam. On the other hand, we have 99.69% accuracy in detecting ham messages, 647 out of 649 messages were tagged as ham which is the same value accuracy without smoothing. The average number of correctly labeled spam messages is 47.4, while the correctly labeled ham messages were 656.4. There was an average of 1.6 false positives and an average of 53.6 false negatives. We also had an average of 96.80% precision and 47.02% recall. The number of true positives are significantly lower than the results of the test trials without smoothing, and subsequently have a higher rate of false negatives. This means that it had a lower performance in detecting spam messages. Further, the rate in which the program incorrectly classified ham messages as spam were of no significant increase but for 1 or more differences. Because of this, the recall percentage is significantly lower. The standard deviation for TP, TN, FP, FN, Precision, and Recall values are 6.77, 7.50, 1.14, 8.32, 0.02, and 0.07, respectively. With the implementation of smoothing, for every 749 messages that we receive, we would expect about 53.6 spam messages that would slip through the spam filtering and into our inboxes, and have 1.6 messages be incorrectly labeled as spam.

IV. Conclusion

This program uses the basics of the Naïve Bayes classification algorithm, wherein the training and testing of data uses calculation methods based on the unique words found in the training set. However, no other special features were implemented, and the Naïve Bayes classification's raw performance in handling text processing can be seen on its own. The results show that there are points to improve in the algorithm since there is still a significant amount of false negatives which can potentially harm individuals as it allows malicious content to be sent in their inboxes. Some pointers may be the implementation of other string processing techniques before adding unique words into the vocabulary of the classifier, the dataset showed a significant amount of alphanumeric characters, lack of spaces, and variations of spelling which then affects how the string processing part of both the training and testing model calculates its decision. For example, "and", "And", "and,", "And,", "and…" and other variations will all be considered unique "words". Although the point of the Naïve Bayes classification algorithm is that it is "naive" of how natural language actually works and treats each set of records as is, future research about text classification may tinker with the idea of still using this concept but allow for some preprocessing of the text. This paper then further shows that a lot can possibly be integrated into this base model to improve the efficiency and effectiveness of the classifier, especially when there is constant development and need for processing big data.