1 Introduction

Spam detection is considered as a binary classification problem– a classification with only two possible classes. In this process, unsolicited messages or ”spam” and legitimate messages or ”ham” emails are classified.

The objective of this Machine Problem is to implement the Naive Bayes classifier for spam filtering. Naive Bayes Classifier (NBC) algorithm uses a branch of mathematics known as probability theory to find the greatest opportunity of classification possibilities, by looking at the frequency of each classification in the training data.

2 Methodology

2.1 Materials

2.1.1 Software

To simulate the Naive Bayes Classifier, the program code was developed using Java. Java is an object-oriented programming language and is supported by platform-specific extensions from Visual Studio Code. The list of software used is shown in table 1.

Software	Description
Operating System	macOS Monterey
IDE	Visual Studio Code Version 1.74.3

Table 1: Software

2.1.2 Hardware

An Intel i5 Quad-core, 8 Gigabyte RAM and 128 GB Hard Disk was used to develop the program code. At the same time, it was where the Naive Bayes classifier were tested and evaluated with and without smoothing. Table 2 summarizes the hardware used to create and run the program.

Hardware	Speicification
Laptop	Intel i5 Quadcore, 8GB RAM, 128GB HD

Table 2: Hardware

2.2 Data

A SMS Spam collection data set from UCI was downloaded and used for the training data and test data for the program. The data set provided consists of 5574 SMS messages that have been labeled as either ”spam” or ”ham”. The first word per line is the label while the remaining words are the text message.

2.3 Classes

The program consisted of one sub class, namely `dataSet`, that was called and used throughout the main class `naiveBayesClassifier`.

2.3.1 `naiveBayesClassifier`

The main class, `naiveBayesClassifier`, is where the downloaded data set `dataSet` was scanned, pre-processed `formVocab(dataSet)`, trained `train()`, and tested `test()`. Basically, it is where the whole Naive Bayes Classification algorithm was implemented. The smoothing `smoothing` function is also included in here.

The overall vocabulary `vocab`, training data `trainingData`, and test data `testData` were initialized here, aside from the ids `ids`, word frequencies `wordFreq`, and `vocabDistinct`– set of unique words in the vocabulary. `int` variables such as the size of training data `trainLen`, `numSpams` (number of spams), `numHams` (number of hams), `truePositive` (TP - number of spam messages classified as spam), `trueNegative` (TN - number of ham messages classified as ham), `falsePositive` (FP - number of ham messages misclassified as spam), and `falseNegative` (FN - number of spam messages misclassified as ham) had an initial value of 0. Probability of spam messages `probSpam` and ham messages `probHam`, and probability of a message to be a spam `probSpamMsg` and ham `probHamMsg` were initialized as `double` data type.

2.3.2 `dataSet`

The class `dataSet` was used for the storing of the `label`, `words[]` of each message and the appearance of each unique `word` in every spam message `spamFreq` and ham message `hamFreq`– `(word, spamFreq, hamFreq)`. It also includes four functions mainly for the frequencies: `addSpamFreq` , `addHamFreq`, `getSpamFreq`, `getHamFreq`.

2.4 Functions

2.4.1 `formVocab(FileInputStream spamData)`

The function `formVocab` marks the start of the Naive Bayes classification. It scans the data from the file input `spamData` and stores it to `vocab` after it is pre-processed. The detailed steps for the `formVocab` are as follows:

Steps for the function `formVocab`:

1. A Scanner `data` is initialized for reading the file and an `ArrayDeque<String[]> dataToWords` to store the messages after tokenization.
2. While `data` has next line, each line is added to `dataToWords` after being converted `toLowerCase()`, the punctuations were removed with `replaceAll()`, and text were `split()` into words (seprated by space).
3. Using an iterator `numIterator` to loop through `dataToWords`, each element in the `ArrayDeque` is converted and stored as `List<String> sentence`.
4. The first element in `sentence`, which is either ham or spam, is retrieved and assigned to variable `String label`.
5. For the rest of the elements in `sentence`, `stream()` is used to filter the text excluding the `label` and then stores each element to `String[] words`.
6. Now `dataSet(label, words)` was assigned to `dataSet ds` and `add(ds)` to `vocab`.
7. Steps 3-6 are repeated until the last element of `dataToWords`.

2.4.2 train()

The `train()` function train the classifier by splitting the data for training and test, storing unique words, counting the frequency of every word occurring per message based on label, and computing the probability of spam and ham messages. The detailed steps for the `train` are as follows:

Steps for the class method `train`:

1. Set `trainLen` to 70% of `vocab.size()` for the training. With `vocab.stream()`, 70% of data from `vocab` is collected and stored to `trainingData`. Skipping `trainLen`, which gives 30% of `vocab`, the rest of the data is collected and stored to `testData`.

2. For every `dataSet ds` in `trainingData`, each word in `ds.words` is searched and stored to `HashSet vocabDistinct` for no duplicates. In addition, `numSpams` is incremented if `ds.label.equals("spam")`, else, increment `numHams`.

3. Now, for each `word` in `vocabDistinct`, a new `dataSet dataFreq` with `dataSet (word, spamFreq: 0, hamFreq: 0)` as initial value is created.

4. Within the loop, `ds.words` in every `dataSet ds` message in the `trainingData` is searched and retrieved by using `stream()` and collects them into `List<String> trainingWords`. Frequency is then counted with `Collections.frequency(trainingWords, word)`, assigning the value to `int freq`.

5. If `ds.label` in current `ds` is ”spam”, `addSpamFreq()` with the given `freq` to the current `dataFreq`. Otherwise, if `ds.label` is ”ham”, `dataFreq.addHamFreq(freq)` is done instead.

6. Once done counting, `dataFreq` is added to `wordFreq` that will be used later on in the `test()` function.

7. With the total `numSpams` and `numHams`, the probability of spam messages `probSpam` and probability of ham messages `probHam` can now be initially calculated with `numSpams / (trainLen + 0.0)` and `numHams / (trainLen + 0.0)`, respectively.

2.4.3 test()

Subsequently, the `test()` is used as an assessment on the implemented code for classifying an unknown message by trying it on the test set `testData`. The `testData` is shuffled for random testing and is divided into 5 for the 5 trials. The retrieved data is then tested without and with smoothing (i.e. if frequency count is 0, virtual counts are added). At the same time, probabilities are computed and values for `trueNegative`, `truePositive`, `falseNegative`, and `falsePositive` are monitored and incremented if conditions are met. Before removing tested data from the list, `computeAccuracy(int trialNumber, boolean withSmoothing)` is called once the last element of each trial is tested. The detailed steps for the `test()` are as follows:

Steps for the class method `test()`:

1. `testData` is copied to `List<dataSet> testList` to shuffle the data with `Collections.shuffle(testList)` for random testing.

2. The `testList` is then divided into 5 by dividing the `testData.size()` by 5 and assigning the value to `int predictLength`. In case the `predictLength` is greater than the remaining `testList.size()`,

predictLength will be equal to testList.size(). With the computed predictedLength, the sublist from testList is retrieved, starting from index 0 to predictLength, and passed to the List<dataSet> toPredictList.

3. In the first loop, each dataSet ds in toPredictList is processed without smoothing given that Boolean withSmoothing was initialized as false. On the second loop, with the same set of data, ds will be processed withSmoothing as true. The initial value in solving for probSpamMsg is also set to probSpam and probHamMsg to probHam.

4. For every word in ds.words, a match of it is searched through the wordFreq with stream() and then collected as List<dataSet> dataWord. If the dataWord.isEmpty() and withSmoothing is true, the smoothing(spamWord:0, numSpams) is called for the probSpamMsg and smoothing(hamWord:0, numHams) for the probHamMsg. Else if the !dataWord.isEmpty(), the spam frequency of the word is retrieved with getSpamFreq() and assigned to int spamWord. The same thing goes for the ham frequency of the word with getHamFreq() but is assigned to int hamWord.

5. Within the else if, if spamWord is 0 and withSmoothing is true, apply smoothing(spamWOrd, numSpams) for the probSpamMsg. Else, multiply spamWord / (numSpams + 0.0) to the current probSpamMsg. On the other hand, if hamWord is 0, and withSmoothing is ture, apply smoothing(hamWord, numSpams) for the probHamMsh. Else, multiply hamWord / numHams + 0.0) to the current probHamMsg.

6. After searching through the words in ds.words, conditions are checked if they're met in incrementing the values of truePositive (prediction is spam, label is spam), falsePositive (prediction is spam, label is ham), trueNegative (prediction is ham, label is ham), and falseNegative (prediction is ham, label is spam).

7. Once the last element in the data set is tested, computeAccuracy(i, withSmoothing) function is called to compute for the accuracy with the collected values.

8. Values for truePositive, trueNegative, falsePositive, and falseNegative are then reset to 0 after the all the data set in current toPredictList is tested. The tested sublist is then removed from the testList.

2.4.4 computeAccuracy(int trialNumber, boolean withSmoothing)

Given the truePositive and falsePositive values, the value of precision is solved by dividing truePositive by (truePositive + falsePositive + 0.0). On the other hand, recall is solved by the dividing truePositive by (truePositive + falseNegative + 0.0). (Note: the trialNumber and withSmoothing are only for output print purposes).

2.4.5 smoothing (int msg, int num

Given the frequency of the word msg occuring in a spam or ham message as 0 and the number of spams or hams num, virtual counts are added. It then returns the probability prob which is equal to (msg + 1.0) / (num + 2.0).

2.4.6 addSpamFreq(int freq)

Increments the count of spam frequency of a word. This is equal to telling the classifier, that this word has occurred once more in a spam message.

2.4.7 `addHamFreq(int freq)`

Increments the count of ham frequency of a word. This is equal to telling the classifier, that this word has occurred once more in a message with this label.

2.4.8 `getSpamFreq()`

Retrieves the number of occurrences of a word in every spam messages.

2.4.9 `getHamFreq()`

Retrieves the number of occurrences of a word in every ham messages.

3 Results and Discussion

From the implemented Naive Bayes Classifier, the given data were trained and tested with a 70 (training data) - 30 (test data) partitioning. For each trial, the same set of data was tested with and without smoothing. A function was also used to compute the accuracy using the precision and recall measures. The formulas for precision and recall are as follows:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

With the implemented NBC algorithm on the given data, below are the results of each trial, with and without smoothing.

Trial #	Without Smoothing			
	TP	**TN**	**FP**	**FN**
1	33	285	1	16
2	32	286	1	16
3	27	294	1	13
4	36	283	2	14
5	38	285	2	8

Table 3: TP, TN, FP, and FN of the trials without smoothing

Table 3 shows the values of TP, TN, FP, and FN per trial without smoothing. On the first trial, the value of TP is 33, the TN is 285, FP is at 1, and FN has the value of 16. Using equation 1, the solved precision is 0.9705882352941176 and the recall is 0.673469387755102 using equation 2. For trial #2, the TP is 32, TN is 286, FP is 1, and FN is 16. Thus, the precision for trial #2 is 0.9696969696969697 and has the recall value 0.6666666666666666. With a TP value of 27, 294 for TN, 1 for FP, and 13 for FN, trial #3 has a precision value of 0.9642857142857143 and a recall value 0.675. On the fourth trial, the value of TP is 36, TN is 283, FP is at 2, and FN has the value of 14. The solved precision for trial #4 is 0.9473684210526315 and the recall is 0.72. For the last trial, the TP is 38, TN is 285, FP is 2, and FN is 8. With these values, the precision for trial #5 is 0.95 and has the recall value 0.8260869565217391.

Trial #	With Smoothing			
	TP	**TN**	**FP**	**FN**
1	49	227	59	0
2	47	219	68	1
3	40	235	60	0
4	50	235	50	0
5	46	223	64	0

Table 4: TP, TN, FP, and FN of the trials with smoothing

In table **??**, TP, TN, FP, and FN values of the 5 trials with smoothing are shown. For trial #1, the TP is 49, TN is 227, FP is 59, and FN is 0. With these values, trial #1 has a precision value of 0.4537037037037037 and a recall 1.0. On the second trial, the value of TP is 47, TN is 219, FP is 68, and FN is 1. Solving the precision with equation 1, we have the value 0.40869565217391307. FOr the recall value, the equation 2 is used and its value is 0.9791666666666666. With a TP value of 40, TN at 235, 60 for FP, and 0 for FN, the precision value of trial #3 is 0.4 and has a recall value of 1.0. For trial #4, the TP is 50, TN is 235, FP is 50, and FN is 0. Thus, the precision for the fourth trial is 0.5 and the recall is 1.0. On the last trial, the TP is 46, TN is 223, FP is 64, and FN is 0. With these values, the precision is 0.41818181818181815 and the recall is 1.0.

4 Conclusion

For our classification problem, precision was used to measure the spam messages that the implemented algorithm correctly identified as a spam message out of all the messages. On the other hand, recall measured the correctly identified spam message out of all the spam messages. Recall also serves as a measure of how well the algorithm can identify the relevant data.

Given the results shown in table 3, the average precision without smoothing is 0.96038786806589 which means that the implemented algorithm is correct 96% of the time. Additionally, the average recall of table **??** is 0.7122446021887. With these values, the implemented algorithm gets almost all of the messages as spam who are really spam correct. Meanwhile, table 4 results have an average precision of 0.43611623481189. Hence, when smoothing is applied, the implemented algorithm is correct 43.6% of the time. On the contrary, the average recall value of the the table is 0.995833333333333. For this reason, a lot of messages were classified as spam, many of them was in the set of actual spam messages, but a lot of them were also ham messages.

Comparing, the average precision and recall of the trials, without smoothing results to high precision and high recall for the classification problem.