Jacob Tucker
CS 482 Artificial Intelligence
Project 2
Naive Bayes ham/spam classification

Task: Given raw data in the form of a csv containing ham/spam classifications and sms messages, create an algorithm that will classify sms messages as either ham or spam with a certain probability. One program, 'training.cc' will parse and count word frequencies for each class and generate two output csv files. One for ham classified message word counts and one for spam classified word counts. Another program, 'classify.cc' will use these word count files.

Datasets
For training and testing my model, I used a 70/30 split of the given spam.csv for training/testing respectively. I used the first 70% of the spam.csv file to create the trainingSpam.csv and the remaining 30% to create the testingSpam.csv.

training.cc
Training.cc is used to parse the raw data input and count word frequencies for each class; ham and spam. The program checks for the correct number of arguments and prints a help message if the number of arguments is not seven. One for the name of the program, three for flags, and three for file names following those flags. The program assumes that the working directory is jacob_tucker_proj_2/Code/build/ and also assumes the files to be read are in the parent directory Code/. The program prepends the file name arguments given with "../" and will look for the input files in Code/ as well as create output files in Code/.

Data structure
All functionality is inside of the main function. As data is read, the program cleans the raw data by removing punctuation and transforming the strings to lowercase. This simplifies the comparison process. Words and word counts are stored in unordered maps. There is one uMap for ham classified words and one for spam classified words. There is also a uMap pointer used to switch between the two data structures when parsing the input file. As each line is read in, the classification of that message is read and sets the uMap pointer to the corresponding uMap. As each word from the message is read, the program checks the uMap for an existing entry and if one exists, it increments the value associated with that word. If an entry does not exist, it creates a new entry with an associated value of one.

Writing
Once the input file is read and the two uMaps are complete, the program writes the values to two output files, one for ham and one for spam. These output files contain the total number of words found inside each class on the first row. The rest of the file is rows containing the word in the first column, and the count of that word in the second, separated by a comma. Each row is separated by a new line. The count of the rows is also the count of unique words for each class minus one for the first line containing the total word count.

classify.cc
Classify.cc uses the generated output files from training to calculate the relative probability of spam or ham of the raw input given to it. The program also creates an output csv file of the original messages and their classification defined by the Naive Bayes algorithm.

Data structure
Similar to training, the generated words and word counts are read in and stored in unordered maps. The data to be written is output to the file as the classifications are made so they do not have to be stored separately. The two uMaps, along with the total word count of each class are stored for use by the main calculation algorithm. The original message is stored before it is cleaned so that the message can be written to the output file without any alteration.

Naive Bayes Calculation
The formula used to create a relative probability of a class given cleaned words from a message.

$$P(c) \ \times \ \prod_{1}^{n} P_n(w|c)$$

where n is the number of words in the message, P(c) is the probability of classification and

$$P(w|c) \ = \ \frac{count(w,c) + 1}{count(c) + |V|}$$

where count(w,c) is the word count stored in the class uMap, count(c) is the total words from the class read and stored from the file, and |V| is the total unique words from both classes which is calculated by adding the sizes of each uMap.

The probability of classification is calculated by dividing either spam or ham total word count, by the sum of both total word counts. Using the testing set, this resulted in approx. 0.79 probability of ham and approx. 0.21 probability of spam.

Each classification is calculated during the parsing loop. Before the loop, two doubles, probIsHam and probIsSpam are initialized to the values of their probability of classification. During the loop, each word in the message's conditional probability (P(w|c)) is calculated and multiplied by and updates probIsHam and probIsSpam. Once all the words are read in and multiplied into the doubles, the program compares the relative probabilities and classifies based on which has the higher relative probability.

Writing
During the main parsing loop of the input file, the classification and pre-cleaned messages are saved. As soon as a classification is made on a message, the program writes its Naive bayes classification and the original message to the output file. The output file is formatted into a csv with the same structure as the original spam.csv, with only the classifications being changed.

Results
The program keeps track of the total number of messages in the input file as well as number of correct classifications. When a message is classified, the program compares its naive Bayes classification against the true classification from the file and only increments the correct classifications when these match. Once all of the messages have been classified, a percentage is calculated from the number of correct classifications over the total amount of messages.

Using the trainingSpam.csv and the testingSpam.csv defined earlier, the program achieves a 96.93% accuracy on the testing data set. I also tested using the technique of ignoring words which do not appear more than a certain number, but found that the accuracy went down a little to approx. 94% - 96% depending on the cutoff point for number of occurrences. This may have to do with the data itself or some other factor not accounted for. However I chose to leave this technique out due to the appearance of higher accuracy without using it.

Included files.
All files needed to build and run the programs are included in the zip. I also included my trainingSpam.csv and testingSpam.csv that were used in my testing. Also included are the generated files from the tests: spamCount.csv and hamCount.csv (word counts generated by training), and classOut.csv (the final classification output file generated by classify.cc).

Build and run instructions are located in the README.md formatted for GitHub.
https://github.com/JacobTucker22/CS482/tree/main/jacob_tucker_proj_2