Project 2:
Naïve Bayes for Spam Classification

Natalie Tipton

Grand Valley State University
CIS 678
Dr. Greg Wolffe

February 11, 2020

Introduction

The Naïve Bayes algorithm completes classification using probabilities which are estimated via supervised learning. It assumes that each attribute is completely independent of one another. This project is a spam classifier for text messages. Therefore, each attribute, or word, is assumed to be entirely independent of every word around it. While this is clearly not true, the algorithm can still perform surprisingly well, without the extremely complex computation that ignoring that assumption would involve. This model will make a decision about whether a text message is spam or not based upon the probabilities that each word within the message belongs to a spam message or not.

Procedure

The completion of this project required a training set of data and a test set of data. As only one model was going to be used, a validation set was not necessary. The original data set provided was split up so that 75 percent of it was used for training and 25 percent of it was used for testing. The training portion of this algorithm began by reading the data line by line. This allowed for each text message to be analyzed separately from one another. Each line was labelled as spam, if it was classified as a spam message, or ham, if it was classified as not a spam message.

To begin processing, each line was split at the occurrence of either ham or spam, as the classification was present as the first word of every message. The remaining portion of the message was then placed word-by-word into a list for either ham or spam messages. This is how the separate lists of all words present in each type of message was created. The words for all messages were also put into another list that would be used to create a vocabulary of all words seen. This functionality can be seen below.

```
for line in lines:
            if line.startswith("ham"):
                # Get the whole string
                ham_count += 1
                raw_text = "".join(line.split("ham"))[1:].rstrip()
                # split string by spaces and append each word appropriately
                for val in raw_text.split(" "):
                    all_words.append(val)
                    ham_raw.append(val)

            if line.startswith("spam"):
                spam_count += 1
                raw_text = "".join(line.split("spam"))[1:].rstrip()
                for val in raw_text.split(" "):
```

```
                all_words.append(val)
                spam_raw.append(val)
```

Next, the words present in the vocabulary, spam, and ham lists were cleaned. It was noted that the presence of capital letters typically does not change the context a word is used in, so all letters were turned lowercase so that the case of a letter would not affect the algorithms ability to properly recognize it as a word present in the vocabulary. After that, all punctuation was removed so that words were not specific to the punctuation that followed them, nor were they specific to any apostrophes that did or did not exist within them. Cleaning of the words within the list concluded with eliminating any words that were purely numeric. As numbers are unlikely to be repeated in multiple texts, including these could have thrown off probabilities for classification. If they did not throw off probabilities, they would not have been of practical use, so they would have just taken up space and time to be analyzed for no benefit. This portion of the code is shown below.

```
def clean_words(all_words):

    cleaned_words = []
    for word in all_words:
        word = word.lower()

        for character in word:
            if character in string.punctuation:
                word = word.replace(character, "")

        if not word.isnumeric() and word != "":
            cleaned_words.append(word)

    return cleaned_words
```

After the lists were cleaned, the subclass for counting hashable elements, Counter, was implemented. This allowed for quick summations of how many times each word occurred in each individual list. This was necessary for calculating the probability that each word would be found in a ham or spam message. To complete manipulation of the lists of words in all messages, the five most common words in all messages were determined using the Counter tool most_common. These most common words would be those that commonly are found in both spam and ham messages and, therefore, would not be useful for determining a hypothesized classification. Since they would not be useful and they occur so often, they would only increase computational expense. Included in the five most common words were to, i, you, a, and the.

After these five words were determined, the list for all words was turned into a set in order to eliminate all duplicate words. Then, those most common words were eliminated from the vocabulary using the remove function so that they would not be accounted for when the test data was run. After testing the model when it was created with and without these five words, it was seen that it functioned slightly better after they were removed from the vocabulary than when they remained.

Next, the probability that each word in the vocabulary set would be found in either a spam or ham message was calculated using Equation 1,

$$P(w_k \mid c_j) = (n_k + 1) / (n + |\text{Vocabulary}|) \quad (1)$$

where w_k is the kth word in the vocabulary, c_j is the class spam or ham, n_k is the number of times that word was found in the given class, and n is the total number of word positions in the class's list of words. These probabilities for each word were saved into a dictionary for their respective classes so that the probability that each word was found in either spam or ham could be referenced later for classification of the test data. After probabilities for each class were found, the difference between the probabilities for each word, respectively, was found for future analysis. This section is shown below.

```
for word in vocab:
            prob_spam[word] = (cleaned_spam_word_counts[word] + 1) / (
                spam_size + vocab_size
            )
            prob_ham[word] = (cleaned_ham_word_counts[word] + 1) / (
                ham_size + vocab_size
            )

            # find the difference between spam and ham probabilities
            # for future analysis of most obvious classifications
            prob_diff[word] = prob_ham[word] - prob_spam[word]
```

That concluded the training portion of this code. Next, the test set was read in and the messages were classified using the model of probabilities for each word. Once again, the messages were read line-by-line. The labelled class was stripped off of each message and placed into a list. The occurrences of ham and spam messages were counted for future analysis. Next, the message without the identifying classifier was saved into a string and looped through word-by-word. The first thing that was checked was if the current word existed in the vocabulary. If it did not, the remainder of the loop was skipped using the continue keyword and

it moved on to the next word in the message. If the word was found in the vocabulary, the probability that it would be found in a spam or ham message were accounted for in the big product portion of Equation 2 for classification. Since this big product involves multiplying many probabilities that are very small, logarithmic rules for addition were used to create Equation 3 and remove the risk of the classification probability approaching zero. After the summation of probabilities was completed, it was plugged into the rest of Equation 3 to determine the Naïve Bayes classification probability for the word in each class.

$$C_{NB} = \max_{c_j \in C} \left(P(c_j) \prod_{i \in Positions} P(a_i|c_j) \right) \tag{2}$$

$$C_{NB} = max_{c_j \varepsilon C} (log_{10} P(c_j) \sum_{i \varepsilon Positions} log_{10} P(a_i|c_j)) \tag{3}$$

This section of the code is shown below.

```
for line in lines:
            # reset the big product for naive bayes calculation back to 0
            big_product_ham = 0
            big_product_spam = 0

            # pull the true classification off the message into a list
            true_class.append(line.split()[0])
            # count up the true occurrences for percentage calculations
            if line.split()[0] == "ham":
                test_ham_count += 1
            if line.split()[0] == "spam":
                test_spam_count += 1

            # leave only the message w/o the classification remaining
            message = line.split()[1:]

            # for each word in the message
            for word in message:
                # do not count if word is not in vocabulary
                if word not in vocab:
                    continue

                # naive bayes formula using log rules
```

```
                big_product_ham += math.log10(prob_ham[word])
                big_product_spam += math.log10(prob_spam[word])

            cnb_ham = math.log10(prob_of_ham_message) + big_product_ham
            cnb_spam = math.log10(prob_of_spam_message) + big_product_spam
```

Next, the probabilities for each word calculated using Equation 3 were compared between classes to make a hypothesized classification. If the Naïve Bayes probability was greater for ham than spam or if the probabilities were equal, the message was labeled as not spam and the word ham was appended onto a list of hypothesized classifications. The decision was made to classify equal probabilities as not spam because it is safer to allow that message through and have the user make the decision as opposed to risk a non-spam message being classified incorrectly as spam and being thrown out before the user can see it. That left the case where the Naïve Bayes probability was greater for spam than ham as the only chance for the message to actually get classified as spam. At this point, the word spam was appended onto the list of hypothesized classifications. This classification portion of code is shown below.

```
if cnb_ham > cnb_spam:
                hyp_class.append("ham")
                hyp_ham_count += 1
            elif cnb_ham < cnb_spam:
                hyp_class.append("spam")
                hyp_spam_count += 1
            else:
                hyp_class.append("ham")
                hyp_ham_count += 1
```

After classification was complete, assessment of how the model performed was necessary. In order to do that, each element of the list created with the given labels, true_class, was compared to each respective element of the list created upon hypothesized classification, hyp_class. If the elements of equal index in both lists were spam, the classification was considered a true positive. If equal index elements were both ham, the classification was a true negative. If the element in the true_class list was spam and the hyp_class list was ham, it was a false negative classification and if true_class said ham and hyp_class said spam, it was a false positive. Here, calling a message spam is considered a positive classification. These counts could then be used to calculate performance metrics of the model as mentioned in the results and discussion section. The counting of true and false positives and negatives can be seen below.

```
or i in range(0, len(true_class)):
```

```
        if true_class[i] == "spam" and hyp_class[i] == "spam":
            true_pos += 1
        elif true_class[i] == "ham" and hyp_class[i] == "ham":
            true_neg += 1
        elif true_class[i] == "spam" and hyp_class[i] == "ham":
            false_neg += 1
        elif true_class[i] == "ham" and hyp_class[i] == "spam":
            false_pos += 1
```

Results and Discussion

This model performed quite well. The terminal output for the test data is shown below in Figure 1. The model was able to accurately classify the test data 96.2 percent of the time. This is significantly higher than the 50 percent accuracy rate that would result if the classification was to be guessed at random every time. It is also higher than the nearly 87 percent accuracy rate if the classification of not spam was guessed every time. The accuracy was calculated using Equation 4.

$$Accuracy = \frac{True\ Positives + True\ Negatives}{Total\ Number\ of\ Messages} \tag{4}$$

The true negative rate of 98.8 percent shows that the model is very good at correctly classifying messages that are not spam. This is extremely important, since a spam filter would not want to improperly classify a real message as spam, since that may result in the user never seeing a message that was genuinely meant for them. The true negative rate was calculated using Equation 5.

$$TNR = \frac{True\ Negatives}{True\ Negatives + False\ Positives} \tag{5}$$

The recall metric shows the true positive rate. That is, it tells how often the model correctly classified a message as spam. Since it is riskier to classify a message as spam than not spam, it is okay that this value is slightly lower, at 79.1 percent, than the true negative ratio. The recall was calculated using Equation 6.

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} \tag{6}$$

The last metric that was calculated to describe the functionality of this model is precision. This tells how likely a classification that a message is spam is actually correct. In other words, it

shows the confidence that a spam classification holds. This model had a precision of 90.6 percent. This was calculated using Equation 7.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} \tag{7}$$

```
true positives = 144
true negatives = 1198
false positives = 15
false negatives = 38

Recall = 79.12087912087912
Precision = 90.56603773584906
True Negative Rate = 98.7633965375103
correct classification = 96.20071684587813
```

Figure 1. Terminal output of test data

One way to visualize the effectiveness of this model is to look at pie charts, created with matplotlib, of the percentage of messages that are truly classified as spam and not spam and the percentage of messages that are hypothesized by the model to be spam and not spam. Figures 2 and 3 show charts of classification in the training and the test data sets based on the classification label they came in with. As shown in Figure 2, the training set was 13.5 percent spam messages and 86.5 percent not spam messages. Figure 3 shows that the training set gave a relatively accurate picture of the test set as there, the messages had a very similar distribution with 13 percent of them being spam and 87 percent being not spam. These charts were created using the code, or similar code, to that shown below.

```
labels = "Spam", "Ham"
sizes = [prob_of_spam_message, prob_of_ham_message]
explode = (0.1, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')


fig1, ax1 = plt.subplots()
ax1.pie(
    sizes,
    explode=explode,
    labels=labels,
    autopct="%1.1f%%",
    shadow=True,
```

```
        startangle=90,
    )
    ax1.axis("equal")  # Equal aspect ratio ensures that pie is drawn as a circle.
    plt.title("True Training Percentages")
    plt.show()
```

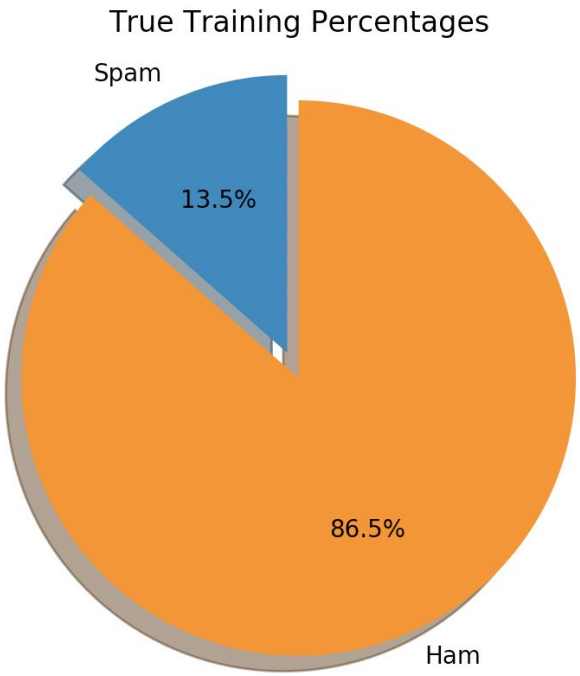

Figure 2. Pie chart of the percentage of spam and not spam messages in the training set based on their true classification

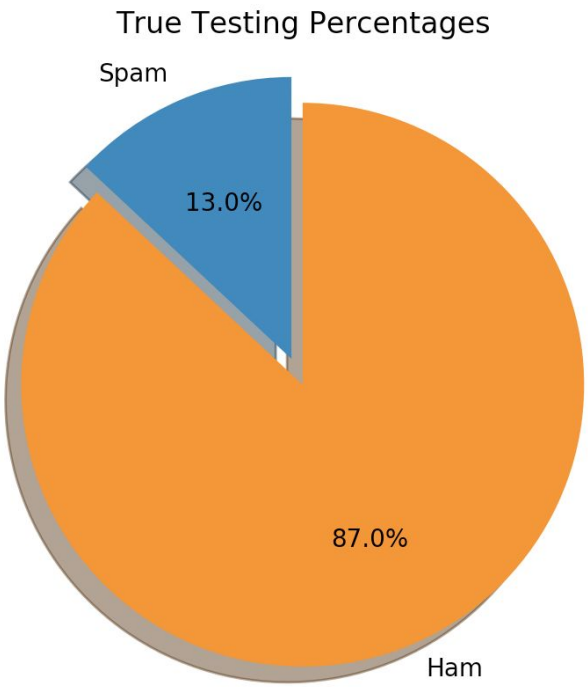

Figure 3. Pie chart of the percentage of spam and not spam messages in the test set based on their true classification

Figure 4 shows another pie chart with the percentage of hypothesized spam and not spam messages in the training set. Here, it is clear that the model inaccurately determined that there was 1.6 percent less messages in spam than there truly were. As mentioned before, this is not a bad number, as one would prefer to see the error go in that direction as opposed to having a larger hypothesized percentage of spam messages than there truly were.

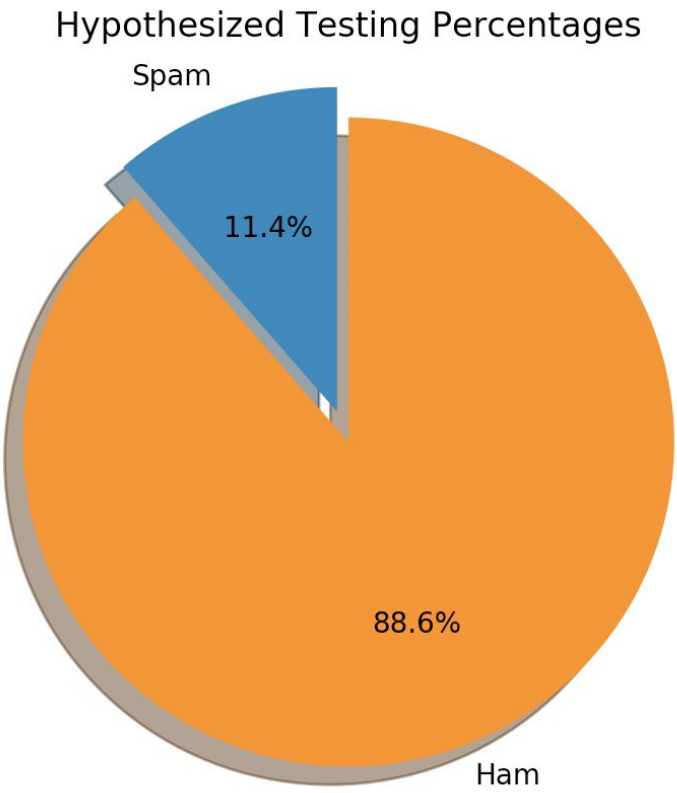

Figure 4. Pie chart of the percentage of spam and not spam messages in the test set based on model hypothesis classification

Another piece of information that seemed like it would be interesting to consider was the words that were most telling for the two classifications as determined during training. That is, the words with the greatest range between the probability that they were found in a spam message and the probability that they were found in a not spam message. Figure 5 shows the five words that had the greatest difference in the direction of being found in a spam message. These words include call, free, txt, claim, and your. The orange bars show the probability of the corresponding word being found in a spam message and the blue bars show the probability of the word being found in a message that was not spam. These words would have had a large amount of weight in the big product for classification of the message. It is clear that for each word, there is a large difference between the probability of each classification, with the probability of spam being much higher than the probability of not spam. These bar charts were also created with matplotlib, using the code below.

```
plt.figure(4)
        N = 5
```

```
    # showing probability for the words determined to be most telling for spam
    top_spam_probs = (
        prob_spam["call"],
        prob_spam["free"],
        prob_spam["txt"],
        prob_spam["claim"],
        prob_spam["your"],
    )
    low_ham_probs = (
        prob_ham["call"],
        prob_ham["free"],
        prob_ham["txt"],
        prob_ham["claim"],
        prob_ham["your"],
    )

    ind = np.arange(N)  # the x locations for the groups
    width = 0.35  # the width of the bars: can also be len(x) sequence

    p1 = plt.bar(ind, low_ham_probs, width)
    p2 = plt.bar(ind, top_spam_probs, width, bottom=low_ham_probs)

    plt.ylabel("Probability of Word")
    plt.title("Probability of Spam or Ham\nFor Most Telling Spam Words")
    plt.xticks(ind, ("call", "free", "txt", "claim", "your"))
    plt.legend((p1[0], p2[0]), ("Ham", "Spam"))

    plt.show()
```

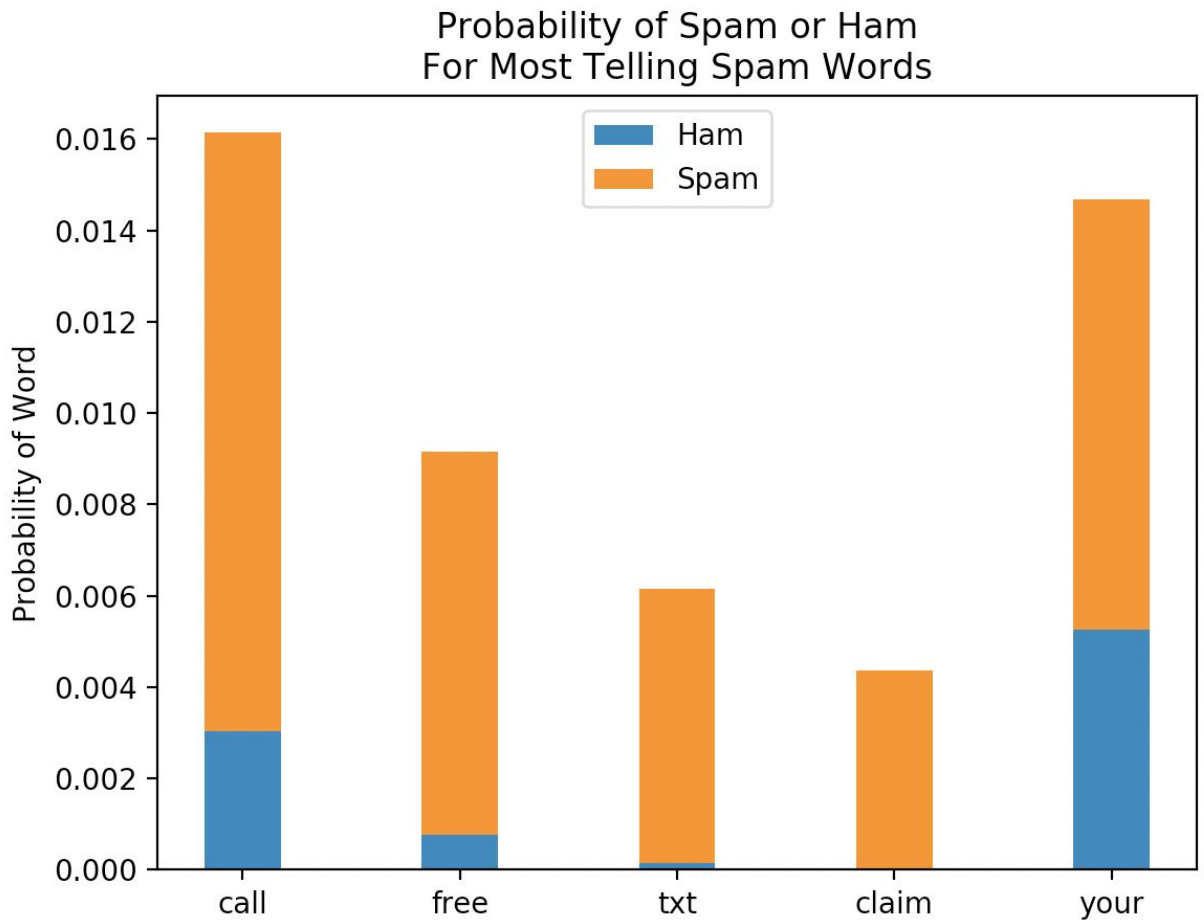

Figure 5. Probabilities that the top five most telling spam words would be found in either a spam or not spam message

Finally, the same graph as above was shown for the top five most telling words for messages that were not spam in Figure 6. These words include my, me, in, it, and u. Here, the colors on the bars have flipped, as the probability that the word would be found in a message that was not spam is now orange and the probability that the word would be found in a spam message is now blue. Once again, the orange bars are all much greater than the blue bars, showing that those particular words were much more likely to be found in a normal message that was not spam and would contribute strongly to the classification of the message as such.

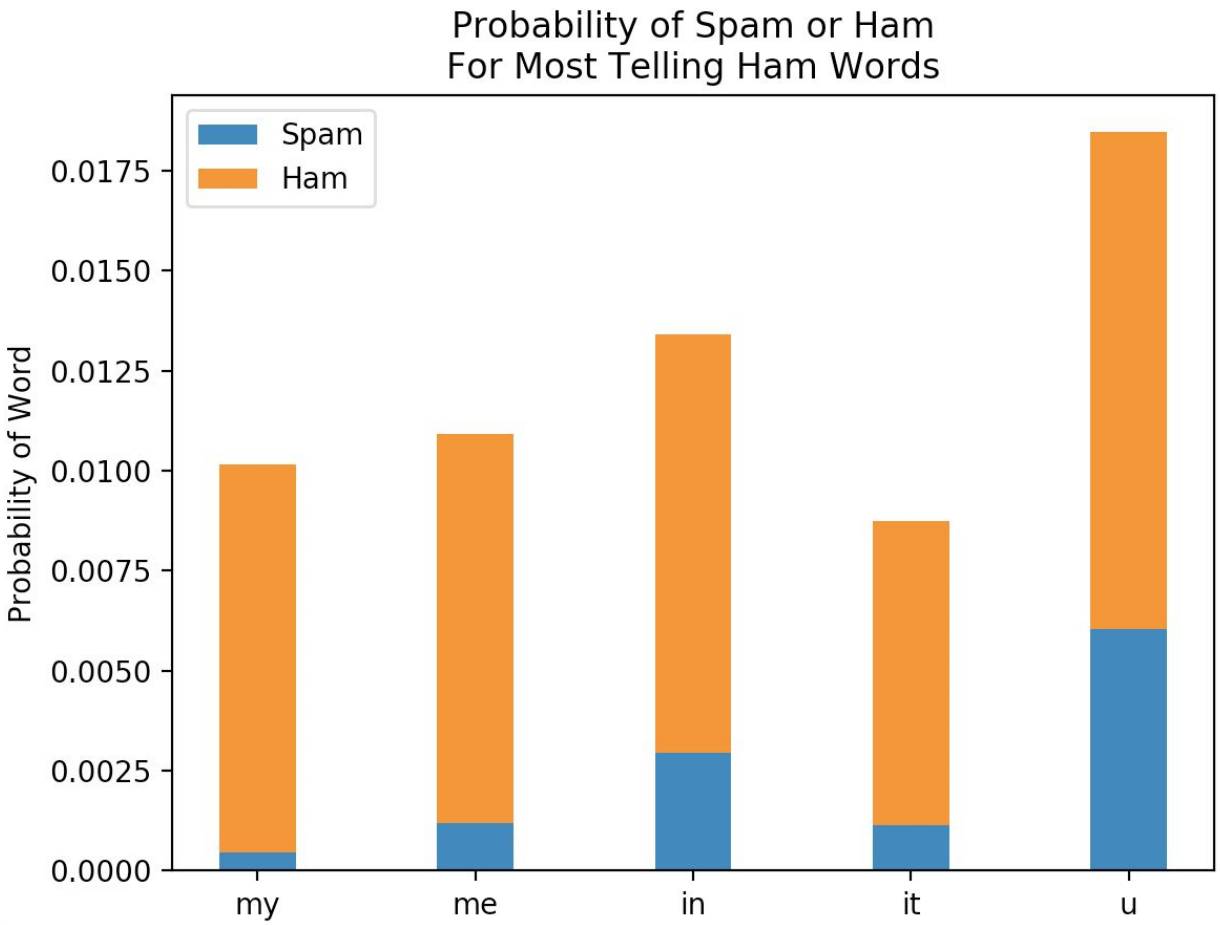

Figure 6. Probabilities that the top five most telling ham words would be found in either a spam or not spam message

Additional time with this project could allow for more pre-processing of the messages that may have resulted in better performance of the model. Additional processes that could have been included would be to remove all small words, such as those two letters in length or less. This could have removed more words that do not give much information about the class of the message. In addition to this, stemming the words in the vocabulary could be very useful. This involves removing the endings of words so that only the root remained. This would allow for more forms of words that currently exist in the vocabulary to be recognized, even if they were not originally found in the training data. These steps may allow for the model to better classify the data without overtraining it to the specific messages seen in the provided training data.

Appendix - Full Source Code

```
# Written by Natalie Tipton
# February 2, 20202
# CIS 678 - Machine Learning
# Dr. Greg Wolfefe
# Spam/Ham Classification of text messages using
#   the Naive Bayes algorithm

from collections import Counter
import string
import math
import heapq
import matplotlib.pyplot as plt
import numpy as np

#####################################################
# Function to clean incoming data:
#   Turn all letters lower case
#   Remove all punctuation
#   Remove all numeric-only strings


def clean_words(all_words):

   cleaned_words = []
   for word in all_words:
       word = word.lower()

       for character in word:
           if character in string.punctuation:
               word = word.replace(character, "")

       if not word.isnumeric() and word != "":
           cleaned_words.append(word)

   return cleaned_words
```

```
################################################

if __name__ == "__main__":

    # Raw text sorting
    ham_raw = []
    spam_raw = []

    # Unique words for ham or spam
    all_words = []
    vocab = set()

    # Probabilities of words in each class
    prob_ham = {}
    prob_spam = {}
    prob_diff = {}

    # initial counts for number of ham/spam messages
    ham_count = 0
    spam_count = 0

    # read in file one line at a time
    with open("./train.data") as f:
        lines = f.readlines()

        # Read all of the lines and append them to their classification
        for line in lines:
            if line.startswith("ham"):
                # Get the whole string
                ham_count += 1
                raw_text = "".join(line.split("ham"))[1:].rstrip()
                # split string by spaces and append each word appropriately
                for val in raw_text.split(" "):
                    all_words.append(val)
                    ham_raw.append(val)

            if line.startswith("spam"):
                spam_count += 1
```

```
        raw_text = "".join(line.split("spam"))[1:].rstrip()
        for val in raw_text.split(" "):
            all_words.append(val)
            spam_raw.append(val)

# clean words in all classifications
cleaned_words = clean_words(all_words)
cleaned_ham_words = clean_words(ham_raw)
cleaned_spam_words = clean_words(spam_raw)

# count up occurrences of each word in all places
cleaned_words_counts = Counter(cleaned_words)
cleaned_ham_word_counts = Counter(cleaned_ham_words)
cleaned_spam_word_counts = Counter(cleaned_spam_words)

# create vocabulary with no repeating words
vocab = set(cleaned_words)

# remove 5 most common words in all messages
vocab.remove("to")
vocab.remove("i")
vocab.remove("you")
vocab.remove("a")
vocab.remove("the")

# find sizes of all different sets of words
vocab_size = len(vocab)
spam_size = len(cleaned_spam_words)
ham_size = len(cleaned_ham_words)

# make dictionaries of spam and ham words with the probability
# that each word will occur in either classification
for word in vocab:
    prob_spam[word] = (cleaned_spam_word_counts[word] + 1) / (
        spam_size + vocab_size
    )
    prob_ham[word] = (cleaned_ham_word_counts[word] + 1) / (
        ham_size + vocab_size
    )
```

```
            # find the difference between spam and ham probabilities
            # for future analysis of most obvious classifications
            prob_diff[word] = prob_ham[word] - prob_spam[word]

        # Use this to find words that are common in everything
        # and maybe not count those for bayes
        # print(cleaned_words_counts.most_common(5))

        # calculate overall percentage of spam and ham messages in training
        prob_of_ham_message = ham_count / (ham_count + spam_count)
        prob_of_spam_message = spam_count / (ham_count + spam_count)

# lists of true and hypothesized classifications
true_class = []
hyp_class = []

# open test data line by line
with open("./test.data") as f:
    lines = f.readlines()

    # set counters for true and hypothesized classes to 0
    test_ham_count = 0
    test_spam_count = 0
    hyp_ham_count = 0
    hyp_spam_count = 0

    # go through lines 1 at a time
    for line in lines:
        # reset the big product for naive bayes calculation back to 0
        big_product_ham = 0
        big_product_spam = 0

        # pull the true classification off the message into a list
        true_class.append(line.split()[0])
        # count up the true occurrences for percentage calculations
        if line.split()[0] == "ham":
            test_ham_count += 1
        if line.split()[0] == "spam":
```

```
            test_spam_count += 1

        # leave only the message w/o the classification remaining
        message = line.split()[1:]

        # for each word in the message
        for word in message:
            # do not count if word is not in vocabulary
            if word not in vocab:
                continue

            # naive bayes formula using log rules
            big_product_ham += math.log10(prob_ham[word])
            big_product_spam += math.log10(prob_spam[word])

        cnb_ham = math.log10(prob_of_ham_message) + big_product_ham
        cnb_spam = math.log10(prob_of_spam_message) + big_product_spam

        # classify message
        if cnb_ham > cnb_spam:
            hyp_class.append("ham")
            hyp_ham_count += 1
        elif cnb_ham < cnb_spam:
            hyp_class.append("spam")
            hyp_spam_count += 1
        else:
            hyp_class.append("ham")
            hyp_ham_count += 1

true_pos = 0
true_neg = 0
false_pos = 0
false_neg = 0

# count up true and false pos and negs
for i in range(0, len(true_class)):
    if true_class[i] == "spam" and hyp_class[i] == "spam":
        true_pos += 1
    elif true_class[i] == "ham" and hyp_class[i] == "ham":
```

```
            true_neg += 1
        elif true_class[i] == "spam" and hyp_class[i] == "ham":
            false_neg += 1
        elif true_class[i] == "ham" and hyp_class[i] == "spam":
            false_pos += 1

    # calculate metrics for model
    correct = (
        100 * (true_pos + true_neg) / (true_pos + true_neg + false_pos + false_neg)
    )
    recall = 100 * (true_pos / (true_pos + false_neg))
    tnr = 100 * (true_neg / (true_neg + false_pos))
    precision = 100 * (true_pos / (true_pos + false_pos))

    ################################################
    # Output
    #   Printed metrics
    #   Pie Charts
    #   Bar Charts

    print("\ntrue positives =", true_pos)
    print("true negatives =", true_neg)
    print("false positives =", false_pos)
    print("false negatives =", false_neg)
    print("\nRecall =", recall)
    print("Precision =", precision)
    print("True Negative Rate =", tnr)
    print("correct classification =", correct, "\n")

    # determining what the most telling spam and ham words are
    # print(heapq.nlargest(5, prob_diff, key=prob_diff.get))
    # print(heapq.nsmallest(5, prob_diff, key=prob_diff.get))

    # create pie charts to show percentage of spam and ham
    # messages in training and testing and what the model
    # determined for the testing set
    labels = "Spam", "Ham"
    sizes = [prob_of_spam_message, prob_of_ham_message]
    explode = (0.1, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')
```

```
fig1, ax1 = plt.subplots()
ax1.pie(
    sizes,
    explode=explode,
    labels=labels,
    autopct="%1.1f%%",
    shadow=True,
    startangle=90,
)
ax1.axis("equal")  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.title("True Training Percentages")
plt.show()

labels = "Spam", "Ham"
sizes = [
    test_spam_count / (test_spam_count + test_ham_count),
    test_ham_count / (test_spam_count + test_ham_count),
]
explode = (0.1, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')

fig2, ax1 = plt.subplots()
ax1.pie(
    sizes,
    explode=explode,
    labels=labels,
    autopct="%1.1f%%",
    shadow=True,
    startangle=90,
)
ax1.axis("equal")  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.title("True Testing Percentages")
plt.show()

labels = "Spam", "Ham"
sizes = [
    hyp_spam_count / (hyp_spam_count + hyp_ham_count),
    hyp_ham_count / (hyp_spam_count + hyp_ham_count),
]
```

```
explode = (0.1, 0)  # only "explode" the 2nd slice (i.e. 'Hogs')

fig3, ax1 = plt.subplots()
ax1.pie(
    sizes,
    explode=explode,
    labels=labels,
    autopct="%1.1f%%",
    shadow=True,
    startangle=90,
)

ax1.axis("equal")  # Equal aspect ratio ensures that pie is drawn as a circle.
plt.title("Hypothesized Testing Percentages")
plt.show()

# create stacked bar charts to show the probability of spam or ham
# for the most telling spam words and then the most telling ham words
plt.figure(4)
N = 5

# showing probability for the words determined to be most telling for spam
top_spam_probs = (
    prob_spam["call"],
    prob_spam["free"],
    prob_spam["txt"],
    prob_spam["claim"],
    prob_spam["your"],
)
low_ham_probs = (
    prob_ham["call"],
    prob_ham["free"],
    prob_ham["txt"],
    prob_ham["claim"],
    prob_ham["your"],
)

ind = np.arange(N)  # the x locations for the groups
width = 0.35  # the width of the bars: can also be len(x) sequence
```

```
p1 = plt.bar(ind, low_ham_probs, width)
p2 = plt.bar(ind, top_spam_probs, width, bottom=low_ham_probs)

plt.ylabel("Probability of Word")
plt.title("Probability of Spam or Ham\nFor Most Telling Spam Words")
plt.xticks(ind, ("call", "free", "txt", "claim", "your"))
plt.legend((p1[0], p2[0]), ("Ham", "Spam"))

plt.show()

plt.figure(5)
N = 5

# showing probability for the words determined to be most telling for ham
low_spam_probs = (
    prob_spam["my"],
    prob_spam["me"],
    prob_spam["in"],
    prob_spam["it"],
    prob_spam["u"],
)
top_ham_probs = (
    prob_ham["my"],
    prob_ham["me"],
    prob_ham["in"],
    prob_ham["it"],
    prob_ham["u"],
)

ind = np.arange(N)  # the x locations for the groups
width = 0.35  # the width of the bars: can also be len(x) sequence

p1 = plt.bar(ind, low_spam_probs, width)
p2 = plt.bar(ind, top_ham_probs, width, bottom=low_spam_probs)

plt.ylabel("Probability of Word")
plt.title("Probability of Spam or Ham\nFor Most Telling Ham Words")
plt.xticks(ind, ("my", "me", "in", "it", "u"))
```

```
plt.legend((p1[0], p2[0]), ("Spam", "Ham"))


plt.show()
```