Devyn Hughes, Sean Deery

IST 664

9/18/2023

Ham vs Spam

Final Project

Introduction

Spam emails have a significant impact on individuals and organizations. These emails can contain malware that can infect, damage, or gain access to computer systems. This can result in the theft of passwords, money, and other sensitive data. It can also alter computing functions to not work as they should or give hackers access to monitor users. Ransomware is a type of malware that blocks access to a computer system or files until a ransom is paid. Ransomware is a major threat to organizations causing millions of dollars in financial loss and damage to the organization's reputation.

Organizations can take different steps to mitigate the risk of malware. These steps include employee training on security awareness, monitoring network traffic for suspicious activity, utilizing sandbox security measures, and filtering out spam emails from employees' inboxes. Spam emails not only contain malicious software but wastes the time of employees which decreases productivity

Companies are tasked with finding a way to separate out these spam emails using any of the information sent over with the email. It can be impossible to find out who is actually sending it, but it is possible to at least not allow it to get to end users. The information that gets sent over in an email includes the sender, the title, and body along with other pieces of metadata. This analysis seeks to create a model that can predict whether an email is spam or ham using the subject and body of the email.

Data

The data used in this analysis was retrieved from the Enrom-Spam dataset. The dataset includes emails generated by Enron employees in the years leading up to the Company's collapse in 2001. The data includes 5,172 emails stored as text documents in folders named ham and spam. The legitimate emails were labeled as "ham" while the spam emails were labeled as "spam".

Importing Libraries

```
# basic
import os
import sys
import random
import pandas as pd

# text processing
import nltk
from nltk.corpus import stopwords

# visulaizations
import matplotlib.pyplot as plt
import seaborn as sns
from wordcloud import WordCloud
from sklearn import metrics
```

To prepare for this project we imported several libraries. The os and sys library were used to be able to extract the data from our directory for processing. The last two libraries in our basic category are random and pandas. The random library was to randomize the data in a later procedure while pandas was utilized for structuring and organizing our data. For the actual processing of the data, we imported NLTK to assist with any evaluations such as frequency distributions and the creation of stop words. Lastly, we imported matplotlib.pyplot, seaborn, wordcloud, and sklearn. These were used for visualizations that were demonstrated later in the project.

Retrieve the Data

```
# define the path to the corpus
path = "C:/Users/dev_h/FinalProjectData/FinalProjectData/EmailSpamCorpora/corpus

# create an empty list for labels and text data
raw_data_list = []

# iterate through each file and append labels and text to lists
for label in os.listdir(path):
    for file in os.listdir(path + "/" + label):
        file = open(path + "/" + label + "/" + file, 'r', encoding="latin-1")
        raw_data_list.append((label, str(file.read())))
        file.close()
```

After downloading the data, the path to the corpus folder was defined for our machine to extract the information from. An empty list called raw_data_list is defined to add each email to. The ham and spam folders are then accessed. Each email in each folder is then iteratively extracted using the encoding “latin-1” and added to the raw_data_list along with its label.

Without the encoding “latin-1”, the program would not understand some of the characters in the emails and would return an error.

Example Email

```
# print out an example email
print("Example: \n\n" +
      "label: " + raw_data_list[1][0] + "\n\n" +
      raw_data_list[1][1])
```

```
Example:

label: ham

Subject: vastar resources , inc .
gary , production from the high island larger block a - 1 # 2 commenced on
saturday at 2 : 00 p . m . at about 6 , 500 gross . carlos expects between 9
, 500 and
10 , 000 gross for tomorrow . vastar owns 68 % of the gross production .
george x 3 - 6992
- - - - - - - - - - - - - - - - - - - - - - - - forwarded by george weissman / ho
u / ect on 12 / 13 / 99 10 : 16
am - - - - - - - - - - - - - - - - - - - - - - - - -
daren j farmer
12 / 10 / 99 10 : 38 am
to : carlos j rodriguez / hou / ect @ ect
cc : george weissman / hou / ect @ ect , melissa graves / hou / ect @ ect
subject : vastar resources , inc .
carlos ,
```

To ensure the data was properly formatted and converted, an example from the raw_data_list is printed out showing the label and the text from the email.

Wordcloud of Raw Data List

In data analysis, visualizations are an effective way for people to understand data. Word clouds are visualizations that display the frequency of words within a text by making frequent words bigger and and less frequent words smaller. With a vast number of emails between the two labels, this is a good way to condense that information. Utilizing a word cloud, we were able to get a visual representation of the most commonly used words. For example, “ect”, “hou”, and “will” were some of the most common words throughout the text in the **raw_data_list**. This is important because it means there could be some similarities or differences that need to be identified between the ham and spam emails.

```
def create_wordcloud(data_list):
    # add all of the text together
    all_text = ' '.join([text for label, text in data_list])

    # Create and generate a word cloud image:
    wordcloud = WordCloud(stopwords=None,  include_numbers=True).generate(all_te

    # Display the generated image:
    plt.imshow(wordcloud, interpolation='bilinear')
    plt.axis("off")
    plt.show()
```

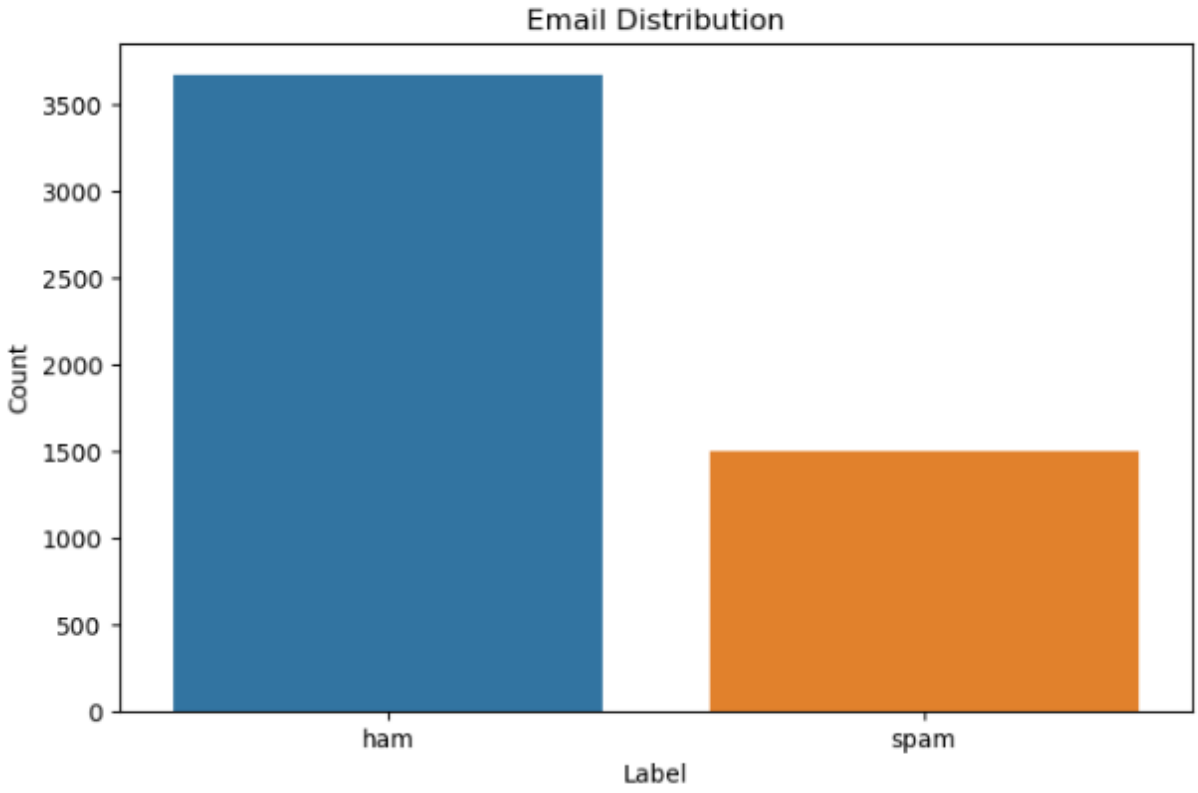
The word cloud is created with a function called **create_wordcloud**. This function creates a new string of all the words in each of the emails. The string is then passed to the WordCloud generator and then displayed with a bilinear interpolation for a smoother image.

Counts of Ham and Spam Emails

```
# define a function to display the counts of each label
def display_label_counts(data_list):

    # print the total numbre of emails
    print("total: " + str(len(data_list)))

    # get the counts for each label
    ham_count = len([label for label, text in data_list if label=='ham'])
    spam_count = len([label for label, text in data_list if label=='spam'])

    # create a dataframe of the label counts
    label_groups = pd.DataFrame({'Label': ["ham", "spam"], 'Count': [ham_count, spam_count]})
    print(label_groups)

    # create a barplot of the label counts
    plt.figure(figsize=(8,5))
    ax = sns.barplot(x="Label", y="Count", data=label_groups)
    ax.set_xticklabels(ax.get_xticklabels(), rotation=0)
    ax.set_title(label="Email Distribution")
    plt.show()
```

The next step in exploring the dataset was to look at the distribution of the data. The function display_label_counts was created to display the total number of emails along with the total for each label, and to plot the distribution in a bar plot.

```
total: 5172
  Label  Count
0   ham   3672
1  spam   1500
```



The total number of emails in the data equals 5172. There were 3672 spam emails and 1500 ham emails. The counts and bar plot show the data is unbalanced and could cause an issue with the predictions being biased towards the ham dataset.

Data Pre-processing

```
raw_data_list[6]
```

```
('ham',
 'Subject: meter 1517 - jan 1999\ngeorge ,\ni need the following done :\njan 13
\nzero out 012 - 27049 - 02 - 001 receipt package id 2666\nallocate flow of 149
to 012 - 64610 - 02 - 055 deliv package id 392\njan 26\nzero out 012 - 27049 -
02 - 001 receipt package id 3011\nzero out 012 - 64610 - 02 - 055 deliv package
id 392\nthese were buybacks that were incorrectly nominated to transport contra
cts\n( ect 201 receipt )\nlet me know when this is done\nhc')
```

Shuffle the data

```
# shuffle the order of ham and spam emails in the list
random.shuffle(raw_data_list)
```

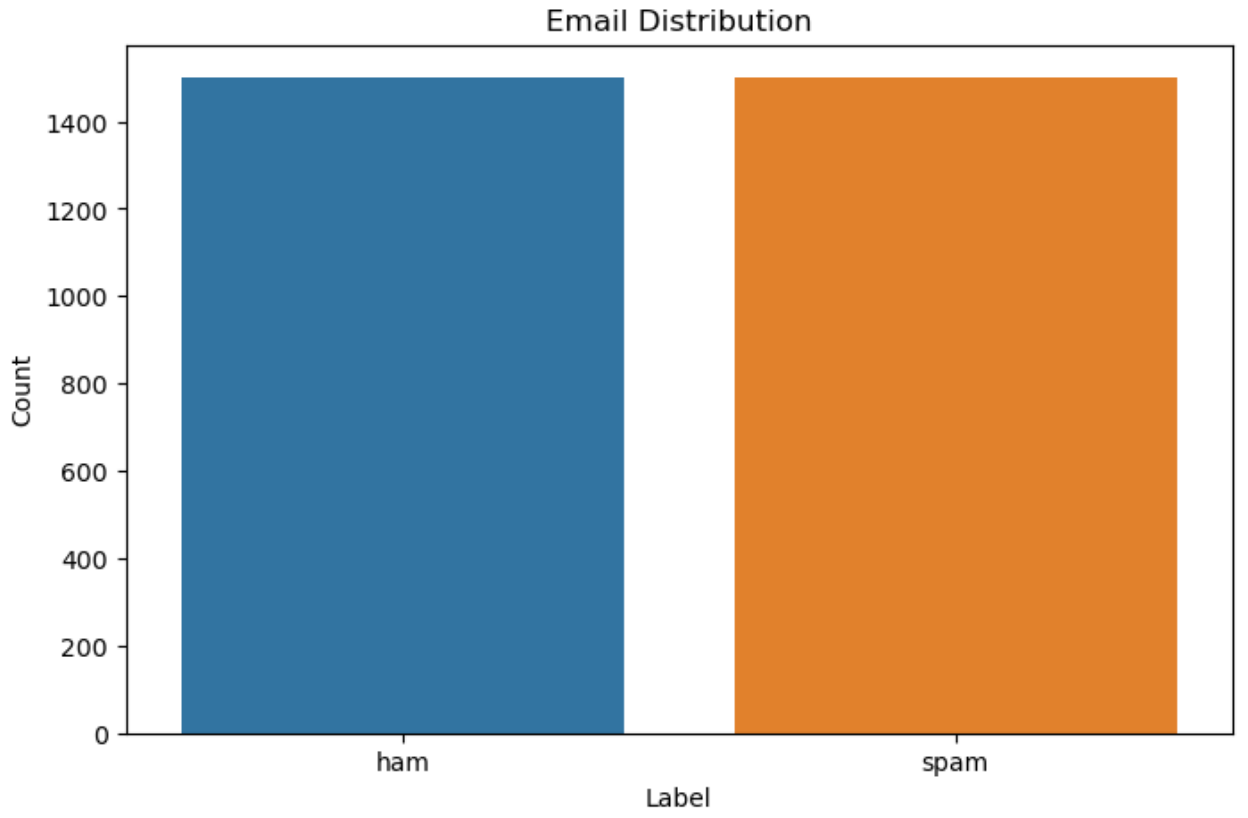
Going into pre-processing our data set, we started by confirming our information was properly listed. Then, we shuffled the information in the **raw_data_list**. This was to make sure that the training set and the test set was truly selected at random since the emails between the two labels were listed categorically.

```
# create an empty list to store the balanced dataset
balanced_data_list = []

# set spam and ham counters to 0
ham_counter = 0
spam_counter = 0

# iteratevily add data to the balanced data list until each label has 1500
for label, text in raw_data_list:
    if ham_counter==1500 and spam_counter==1500:
        break
    elif label=="ham" and ham_counter<1500:
        balanced_data_list.append((label, text))
        ham_counter += 1
    elif label=="spam" and spam_counter<1500:
        balanced_data_list.append((label, text))
        spam_counter += 1
```

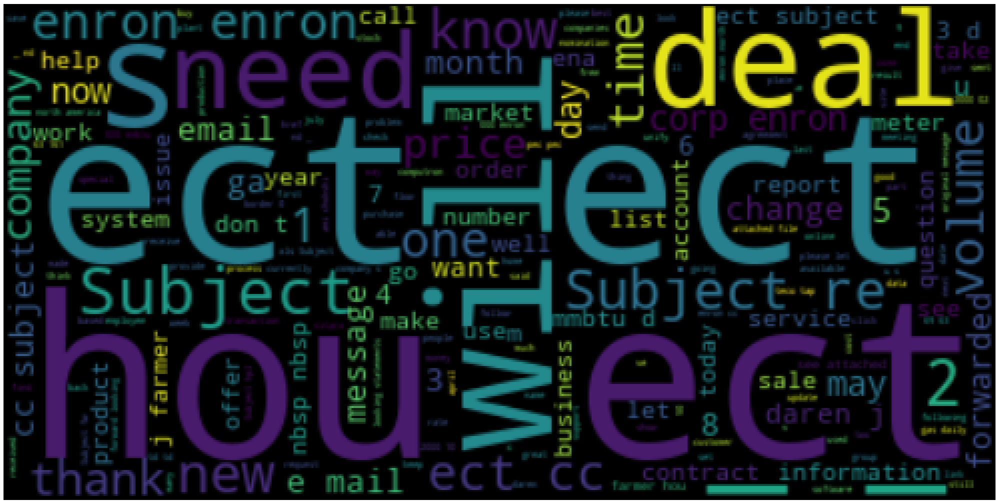
```
# shuffle the order of ham and spam emails in the list
random.shuffle(balanced_data_list)
```

```
# get the counts for each label
display_label_counts(balanced_data_list)
```

After creating the base list of pure raw data, we then forged a balanced set to perform testing with. In the code above, we created a list called **balanced_data_list.** Once that was created, we created counters for each of the labels. To iterate the information with a limit of 1500 per label, we used a for loop. In this for loop, for each label and text in raw_data_list that equaled ham or spam, we had Python pull 1500 into the balanced data list. This created a balanced list of 1500 ham emails and 1500 spam emails. Once this action was completed, the loop would break once the counters both equaled 1500. Here is a visualization of the result below:



After verifying our balanced data set was complete, we used the same wordcloud function to obtain a different wordcloud.



Based on the word cloud created for the raw data list and the balanced list, we can see that some words still occur in high frequency between the two sets. Some noticeable differences is the word "deal" has a higher frequency than how it was used in the raw data set. Also, need was shown to occur more profusely in the balanced data set.

Tokenizing the Data

```
# get a list of stopwords from nltk
nltk_stopwords_list = nltk.corpus.stopwords.words('english')

# create a list of additional stopwords
more_stopwords_list = ["Subject"]

# add the two stopword lists together
my_stopwords_list = nltk_stopwords_list + more_stopwords_list
```

In order to further our project, we wanted to set up stop words that were based on the nltk database. Creating a list called **nltk_stopwords_list**, we used the corpus called Stop Words for the English language. Also, we added another word into the list called "Subject". The reason for doing this was because all the emails started off with the word "Subject". Including this word would not help classify the text and would only add noise. Now with both lists identified, we combined the two to create the **my_stopwords_list** for our features in later use.

```
# define a function to tokenize data with a stopwords option
def tokenize_data(data, stopwords_list=[]):

    # create an empty list for the labels and tokenized text
    tokenized_list = []

    # tokenize and clean each email
    for label, text in data:

        # tokenize the text
        tokens = nltk.word_tokenize(text)

        # remove the stopwords
        new_tokens = [word for word in tokens if not word in stopwords_list]

        # append the email to the tokenized data list
        tokenized_list.append((label, new_tokens))

    # return the tokenized list
    return tokenized_list
```

To create our tokens, we used a function called **tokenize_data**. In this function, we created a list called **tokenized_list**. Using a for loop, we had Python tokenize the text in each label. After that, we used a new list called **new_tokens** to iterate all the tokens that are not included in the stopwords_list. This will take all the stop words out of the tokens list. After performing this action, we appended all the new tokens into the **tokenized_list** and returned it.

```
# tokenize the raw data list without removing stopwords
raw_base_tok_list = tokenize_data(raw_data_list)

# tokenize the raw data list removing stopwords
raw_sw_tok_list = tokenize_data(raw_data_list, my_stopwords_list)

# tokenize the balanced data list without removing stopwords
balanced_base_tok_list = tokenize_data(balanced_data_list)

# tokenize the balanced data list removing stopwords
balanced_sw_tok_list = tokenize_data(balanced_data_list, my_stopwords_list)
```

With the function created, we can now begin tokenizing all of our separate lists. We created a total of four lists. Two of them are made from the raw data set and the other two are made from the balanced data set. Using the **tokenize_data** function, we applied it to all of our lists (**raw_base_tok_list, raw_sw_tok_list, balanced_base_tok_list, and balanced_sw_tok_list**). This provided all the tokens or “bag of words” for our feature set creation.

Unigram Featuresets

```
def get_word_frequencies(words_list):

    # get the frequency of the words
    words_freq_list = nltk.FreqDist(words_list)

    # print the number of words
    print(f'Number of words: {str(len(words_freq_list))}')

    # print the most common words
    print(f'Most common words: \n {words_freq_list.most_common(50)}')

    return words_freq_list

def unigram_features(tokenized_data_list):

    print("All Words")

    # get all the words
    all_words_list = [word for (label,text) in tokenized_data_list for word in text]

    # get the frequency of the words
    all_words_freq_list = get_word_frequencies(all_words_list)

    # get the 2000 most frequently appearing keywords in the corpus
    most_freq_words_list = all_words_freq_list.most_common(2000)

    # create the list of words to use in the feature sets
    word_features_list = [word for (word, count) in most_freq_words_list]

    print("\nHam Words")

    # get all the ham words
    ham_words_list = [word for (label,text) in tokenized_data_list if label=="ham" for word in text]

    # get the frequency of the ham words
    ham_words_freq_list = get_word_frequencies(ham_words_list)

    print("\nSpam Words")

    # get all the spam words
    spam_words_list = [word for (label,text) in tokenized_data_list if label=="spam" for word in text]

    # get the frequency of the spam words
    spam_words_freq_list = get_word_frequencies(spam_words_list)

    return word_features_list
```

The first list of features was created using the word frequencies to get the top 2000 most frequent words. The unigram_features function takes in a tokenized data list, gets a list of all the words, and then passes it to the get_word_freqeuncies function. This function gets the frequencies of each unique word with NLTK's FreqDist function, prints the number of words along with the top 50 words, and then return s the list of the top 2000 most frequent words. The same process is applied to the ham and spam data to display the most frequent words in each label. The unigram_features function then returns the top 2000 most frequent words in the dataset which are then used to create the featuresets for unigram frequencies.

```
# get the features for the raw data without stopwords removed
raw_base_features = unigram_features(raw_base_tok_list)
```

Using the function on the unbalanced tokenized data, we determined a total of 50,566 total words. The five most common words used were actually punctuation marks. We didn't exclude these from the model because we wanted to test if punctuation helped the model or not. Ham

had a total of 20,249 total words and spam had 38,7899. This could mean that ham emails have a more distinct usage of words while spam emails vary greatly.

```
# get the features for the raw data with stopwords removed
raw_sw_features = unigram_features(raw_sw_tok_list)
```

```
# get the features for the balanced data without stopwords removed
balanced_base_features = unigram_features(balanced_base_tok_list)
```

```
# get the features for the balanced data with stopwords removed
balanced_sw_features = unigram_features(balanced_sw_tok_list)
```

Reviewing the next three lists of features, the unbalanced data with stop words removed have a total of 50,413 words. For the ham emails, there were a total of 20,100 words and spam had 38,649 words. For the balanced data, there were a total of 45,129 words with ham equaling 12,869 and spam equating to 38,799. Lastly, the balanced list with stop words removed had a total of 44,976 words. For the ham and spam words, ham emails had 12,723 words and spam had 38,649 words. Analyzing this information, there looks to be a trend where spam unigrams are higher than ham unigrams.

```
# define function to get the word counts of each email
def word_frequency_features(tokens, word_features_list):
    document_word_freq = nltk.FreqDist(tokens)
    features = {}
    for word in word_features_list:
        if word in document_word_freq.keys():
            features[word] = document_word_freq[word]
        else:
            features[word] = 0
    return features

# use the function to create the featuresets
raw_base_featuresets = [(word_frequency_features(tokens, raw_base_features), label) for (label, tok

# use the function to create the featuresets
raw_sw_featuresets = [(word_frequency_features(tokens, raw_sw_features), label) for (label, tokens)

# use the function to create the featuresets
balanced_base_featuresets = [(word_frequency_features(tokens, balanced_base_features), label) for (

# use the function to create the featuresets
balanced_sw_featuresets = [(word_frequency_features(tokens, balanced_sw_features), label) for (labe
```

The word_frequency_features function is then used to add the counts of words from each email to the features created in the unigram_features function. The counts of each word in the email is retrieved with FreqDist. The frequencies are then combined with the feature list to create a

dictionary of words and the frequencies of those words in the email. This process is completed for the unbalanced dataset, the unbalanced dataset with stopwords removed, the balanced dataset, and then the balanced dataset with stopwords removed.

Verb Phrase Featuresets

```
# tag the text, get freq info
tagged_dataset = [(nltk.pos_tag(token), label) for label, token in balanced_sw_tok_list]
```

Verb phrase featuresets were then created to capture the predictive quality of calls to action that many spam emails contain. To acquire the verb phrases, the tokens were first tagged with the part of speech tag, resulting in the tagged_dataset.

```
def get_phrase_frequencies(phrases_list):

    # get the frequency of the phrases
    phrases_freq_list = nltk.FreqDist(phrases_list)

    # print the number of words
    print(f'Number of phrases: {str(len(phrases_freq_list))}')

    # print the most common words
    print(f'Most common phrases: \n {phrases_freq_list.most_common(50)}')

    return phrases_freq_list
```

The get_phrase_frequencies function was then created to assist in getting the verb phrase frequencies just like the unigrams.

```
# CREATING FUNCTIONS TO REUSE CODE FOR RETRIEVING DIFFERENTE TYPES OF PHRASES
def verb_phrases(tagged_dataset):

    print("All Verb Phrases")

    # create the grammar regex
    grammar = """
        NP: {<DT>?<JJ>*<NN>}
        PP: {<IN><NP>}
        VP: {<VB.*><NP|PP>+$}
    """

    # get a list of lists containing tagged words of emails
    tagged_email_list = [tagged_email for (tagged_email, label) in tagged_dataset]

    # create an empty list to store verb phrases
    vp_chunks = []

    # parse each of the emails and get the verb phrases
    for tagged_tokens in tagged_email_list:
        # create the parser
        parser = nltk.RegexpParser(grammar, loop=2)
        # Parse the tagged text
        tree = parser.parse(tagged_tokens)
        # get the verb phrases
        for subtree in tree.subtrees():
            if subtree.label() == 'VP':
                vp_chunks.append(' '.join([word for word, tag in subtree.leaves()]))
                #print(' '.join([word for word, tag in subtree.leaves()]))

    #print(' '.join([word for word, tag in vp_chunks[0].leaves()]))

    # get the frequency of the phrases
    all_vp_freq_list = get_phrase_frequencies(vp_chunks)

    # get the 2000 most frequently appearing phrases in the corpus
    most_freq_vp_list = all_vp_freq_list.most_common(2000)

    # create the list of phrases to use in the feature sets
    vp_features_list = [word for (word, count) in most_freq_vp_list]

    return vp_features_list
```

The verb_phrase function was then created to return the features that would be used to make the featuresets. The grammar used to extract the verb phrases is defined in regex, and is then passed to the RegexParser which is used to get the verb phrases in every email and append them to a list called vp_chunks.

```
# define function to get the phrase counts of each email
def phrase_frequency_features(phrases, phrase_features_list):

    # get the frequency of the prhases
    document_phrase_freq = nltk.FreqDist(phrases)
    features = {}
    for phrase in phrase_features_list:
        if phrase in document_phrase_freq.keys():
            features[phrase] = document_phrase_freq[phrase]
        else:
            features[phrase] = 0
    return features

# use the function to create the featuresets
balanced_sw_vp_featuresets = [(phrase_frequency_features(phrases, balanced_sw_vp_features), label) for (phrases, label) in ta
```

The phrase_frequency_features function is then used with the previously created features to assign frequencies from each email.

Punctuation Featuresets

```
def get_punctuation_frequencies(punctuation_list):

    # get the frequency of the punctuation marks
    punctuation_freq_list = nltk.FreqDist(punctuation_list)

    # print the number of punctuation marks
    print(f'Number of punctuation marks: {str(len(punctuation_freq_list))}')

    # print the most common punctuation marks
    print(f'Most common punctuation marks: \n {punctuation_freq_list.most_common(50)}')

    return punctuation_freq_list


def punctuation_features(tokenized_data_list):

    print("All Punctuation Marks")

    # create a list of punctuation marks
    punctuation_marks = [*string.punctuation]

    # get all the words
    all_punctuation_list = [token for (label,text) in tokenized_data_list for token in text if token in punctuation_marks]

    # get the frequency of the punctuation
    all_punctuation_freq_list = get_punctuation_frequencies(all_punctuation_list)

    # get the 2000 most frequently appearing punctuation marks in the corpus
    most_freq_punctuation_list = all_punctuation_freq_list.most_common(2000)

    # create the list of punctuation to use in the feature sets
    punctuation_features_list = [token for (token, count) in most_freq_punctuation_list]

    return punctuation_features_list
```

Similar preprocessing steps are taken for punctuation featuresets. The punctuation_features function uses the get_punctuation_frequencies to get the list of punctuation marks used in the dataset.

```
# define function to get the phrase counts of each email
def punctuation_frequency_features(punctuation_marks, punctuation_features_list):

    # get the frequency of the punctuation marks
    document_punctuation_freq = nltk.FreqDist(punctuation_marks)
    features = {}
    for punctuation_mark in punctuation_features_list:
        if punctuation_mark in document_punctuation_freq.keys():
            features[punctuation_mark] = document_punctuation_freq[punctuation_mark]
        else:
            features[punctuation_mark] = 0
    return features

# use the function to create the featuresets
balanced_sw_punct_featuresets = [(punctuation_frequency_features(tokens, balanced_sw_punct_features), label) for (label, toke
```

The features are then passed into the punctuation_frequency_features function which gets the counts of each punctuation mark in each email. The featuresets are then created for the balanced dataset with stopwords removed.

Classification Functions

```
## cross-validation ##
# this function takes the number of folds, the feature sets
# it iterates over the folds, using different sections for training and testing in turn
#   it prints the accuracy for each fold and the average accuracy at the end
def cross_validation_accuracy(num_folds, featuresets):
    subset_size = int(len(featuresets)/num_folds)
    print('Each fold size:', subset_size)
    accuracy_list = []
    # iterate over the folds
    for i in range(num_folds):
        test_this_round = featuresets[(i*subset_size):][:subset_size]
        train_this_round = featuresets[:(i*subset_size)] + featuresets[((i+1)*subset_size):]
        # train using train_this_round
        classifier = nltk.NaiveBayesClassifier.train(train_this_round)

        # evaluate against test_this_round and save accuracy
        accuracy_this_round = nltk.classify.accuracy(classifier, test_this_round)
        print (i, accuracy_this_round)
        # add the accuracy to the accuracy list
        accuracy_list.append(accuracy_this_round)

    # find mean accuracy over all rounds
    print ('Mean Accuracy', sum(accuracy_list) / num_folds)
```

For this experiment, we utilized a Cross Validation process. Starting off, we created a function called **cross_validation_accuracy**. In order to do so we created a subset size by taking the length of the featureset and dividing it by the number of folds. Then, we made an empty list called **accuracy_list**. This is so we can append the accuracy of the model in the list. Using a loop again, we created a test set and a training set based on the featuresets that would be utilized for the equation. The classifier was then created using the NLTK Naive Bayes Classifier on the training set. For an accuracy score, we then classified the accuracy using the nltk classify accuracy method on both the classifier and the test set. Finally, the result would be calculated as an output with the Mean Accuracy.

```
# define a function to show the confusion matrix
def create_confusion_matrix(goldlist, predictedlist, classes):

    counter = 0
    for count, label in enumerate(goldlist):
        if label == 'ham' and predictedlist[count] == 'spam':
            counter+=1
    print(f'{counter} times when ham was labeled as spam')


    # print the confusion matrix
    cm = nltk.ConfusionMatrix(goldlist, predictedlist)
    print(cm.pretty_format(sort_by_count=True, truncate=9))

    # show the results as percentages
    print(cm.pretty_format(sort_by_count=True, show_percents=True, truncate=9))

    cm = metrics.confusion_matrix(goldlist, predictedlist, labels=classes)
    disp = metrics.ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classes)
    disp.plot()
    plt.show()
```

To follow the trend of having visualizations as well, we then created a function to generate a confusion matrix. This function was called **create_confusion_matrix**. Starting with creating a variable equal to 0, we performed a loop to count the labels between ham and spam in the predicted list. If the conditions were met, it would add +1 to the counter. Then, by using the nltk confusion matrix method, it would print a confusion matrix to visualize.

```
# Function to compute precision, recall and F1 for each label
#  and for any number of labels
# Input: list of gold labels, list of predicted labels (in same order)
# Output:  prints precision, recall and F1 for each label
def eval_measures(gold, predicted):
    # get a list of labels
    labels = list(set(gold))
    # these lists have values for each label
    recall_list = []
    precision_list = []
    F1_list = []
    for lab in labels:
        # for each label, compare gold and predicted lists and compute values
        TP = FP = FN = TN = 0
        for i, val in enumerate(gold):
            if val == lab and predicted[i] == lab:  TP += 1
            if val == lab and predicted[i] != lab:  FN += 1
            if val != lab and predicted[i] == lab:  FP += 1
            if val != lab and predicted[i] != lab:  TN += 1
        # use these to compute recall, precision, F1
        recall = TP / (TP + FP)
        precision = TP / (TP + FN)
        recall_list.append(recall)
        precision_list.append(precision)
        F1_list.append( 2 * (recall * precision) / (recall + precision))

    # the evaluation measures in a table with one row per label
    print('\tPrecision\tRecall\t\tF1')
    # print measures for each label
    for i, lab in enumerate(labels):
        print(lab, '\t', "{:10.3f}".format(precision_list[i]), \
          "{:10.3f}".format(recall_list[i]), "{:10.3f}".format(F1_list[i]))
```

The eval_measures function is also used to extract and print out the precision, recall, and f1 scores. A high precision means the model returns more relevant results and less irrelevant ones for stocks. A high recall means the model returns most of the relevant results whether or not irrelevant ones are also returned. The F1 score is a mix of precision and recall scores.

```
# define function to split the data
def train_test_accuracy(featuresets):

    # set the training set size as approximately 70% of data
    size = int(len(featuresets) * 0.7)
    # create the training and testnig data sets
    train_set, test_set = featuresets[:size], featuresets[size:]

    # train the model
    classifier = nltk.NaiveBayesClassifier.train(train_set)

    # evaluate the accuracy of the classifier
    print(f'Test Accuracy: {nltk.classify.accuracy(classifier, test_set)}')

    # create lists of predictions and actual labels
    goldlist = []
    predictedlist = []
    for (features, label) in test_set:
        goldlist.append(label)
        predictedlist.append(classifier.classify(features))

    classes = classifier.labels()

    # create a confusion matrix
    create_confusion_matrix(goldlist, predictedlist, classes)

    # print evaluation measure
    eval_measures(goldlist, predictedlist)

    # show which features of classifier are most informative
    print(classifier.show_most_informative_features(30))
```

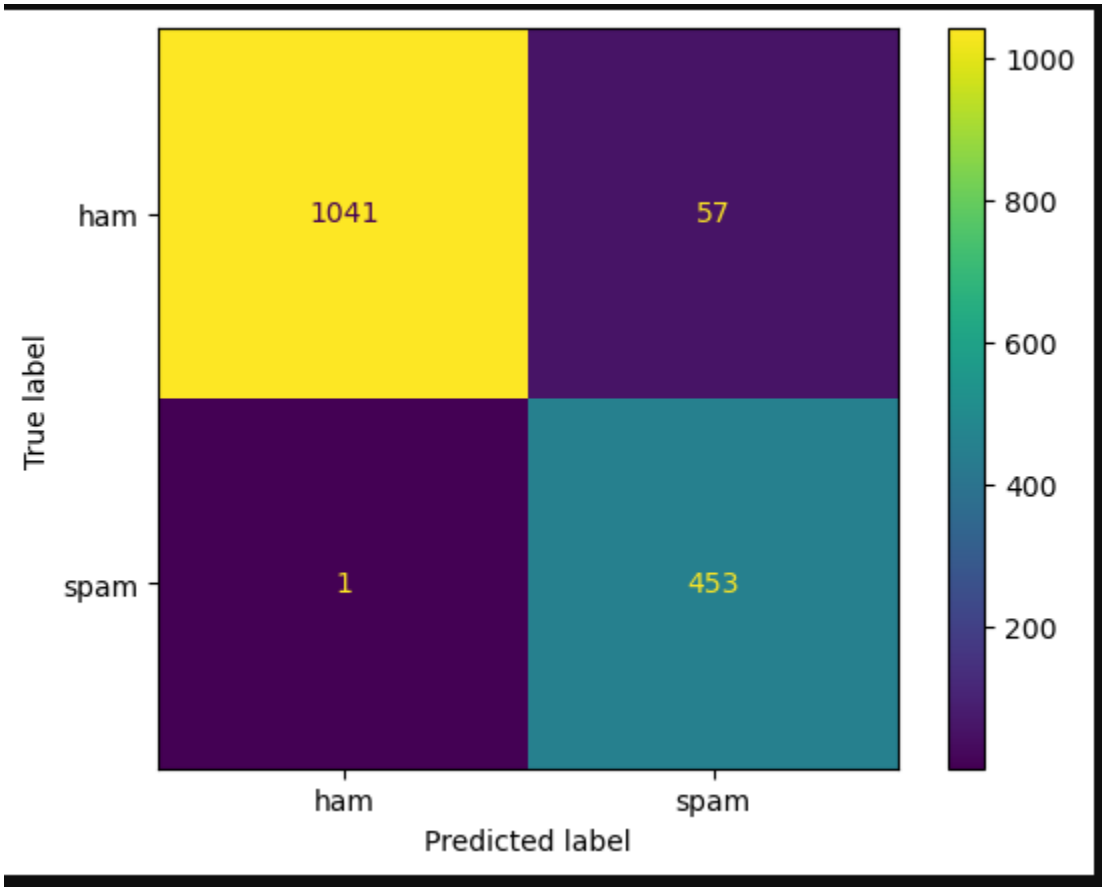
In this code, we created a function to split the data accordingly. Naming the function **train_test_accuracy**, it considers any feature sets created as an input. For this experiment, we utilized 70% of the data within a set called **size**. From here we divided the data under **train_set** and **test_set**. The classifier used in this calculation is the Naive Bayes Classifier that will be implemented on the **train_set**. Printing the test accuracy, we created a list for predictions and actual labels. These were labeled as **goldlist** and **predictedlist**. To compliment our results, we used the confusion matrix method for a visual representation of our results while using the **eval_measures** function to calculate our measures.

Classification Results

```
# perform the cross-validation on the featuresets with word features and generat
num_folds = 5
cross_validation_accuracy(num_folds, raw_base_featuresets)
```

```
Each fold size: 1034
0 0.9468085106382979
1 0.9400386847195358
2 0.9642166344294004
3 0.9584139264990329
4 0.9613152804642167
Mean Accuracy 0.9541586073500966
```

With everything set up, we began our first calculation of accuracy. using the **cross_validation_accuracy** function we created. Setting the number of folds to 5, we utilized the function on the **raw_base_featuresets**. This was the base feature set without anything removed it. The accuracy of prediction was roughly a mean accuracy of 95.42%. This means, by utilizing all the “bag of words” in the feature set, the model was able to predict if the email was spam or an actual email about 95% of the time. Here is a confusion matrix to represent the results.



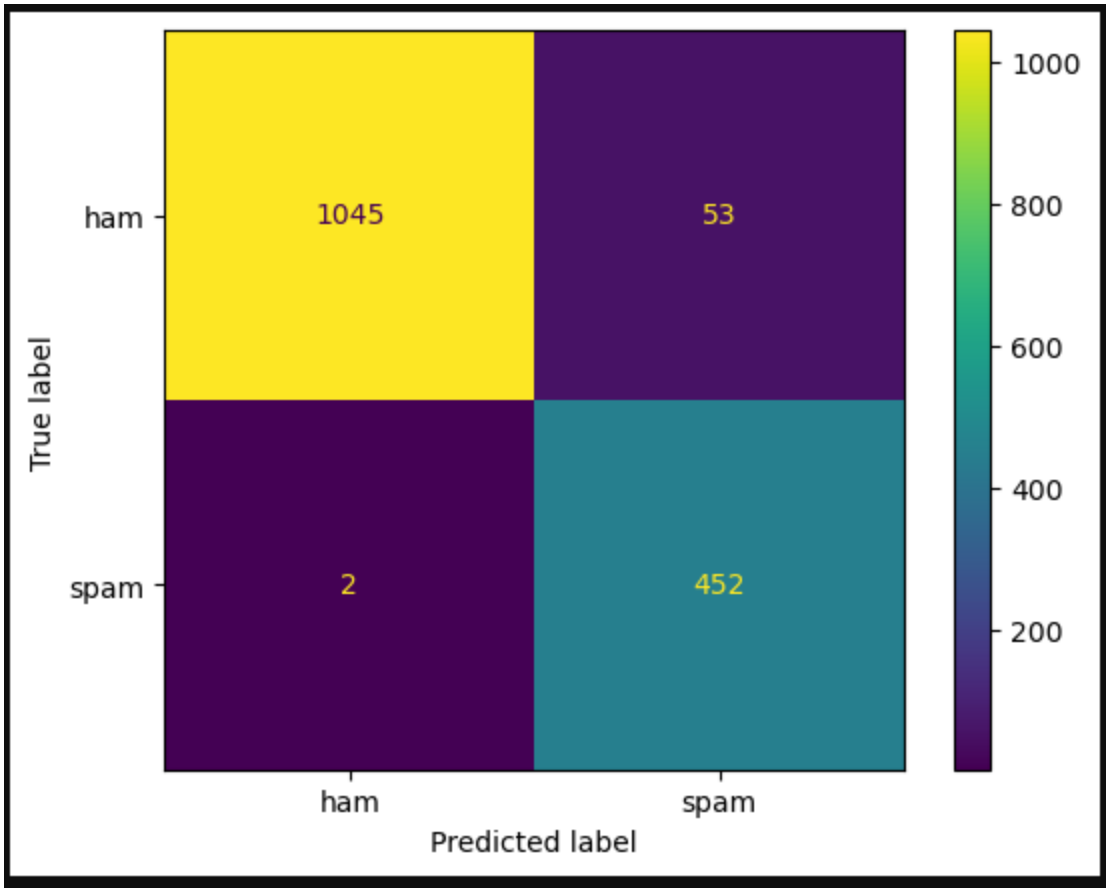
Based on the most informative features, “forwarded” was the top feature for this prediction. When the word “forwarded” occurred once, then there was a ratio of 132:1 in favor of a “ham” email.

```
Most Informative Features
                forwarded = 1                ham : spam   =    132.1 : 1.0
                     2004 = 1               spam : ham    =     94.1 : 1.0
                     pain = 1               spam : ham    =     74.5 : 1.0
                 thousand = 1               spam : ham    =     66.3 : 1.0
                 creative = 1               spam : ham    =     59.8 : 1.0
                       ex = 1               spam : ham    =     59.8 : 1.0
                      ibm = 1               spam : ham    =     56.6 : 1.0
                      sex = 1               spam : ham    =     56.5 : 1.0
                clearance = 1               spam : ham    =     54.9 : 1.0
                        u = 3               spam : ham    =     53.2 : 1.0
                      971 = 2               spam : ham    =     50.0 : 1.0
```

```
# perform the cross-validation on the featuresets with word features and generat
num_folds = 5
cross_validation_accuracy(num_folds, raw_sw_featuresets)
```

```
Each fold size: 1034
0 0.9477756286266924
1 0.937137330754352
2 0.9642166344294004
3 0.9593810444874274
4 0.9632495164410058
Mean Accuracy 0.9543520309477757
```

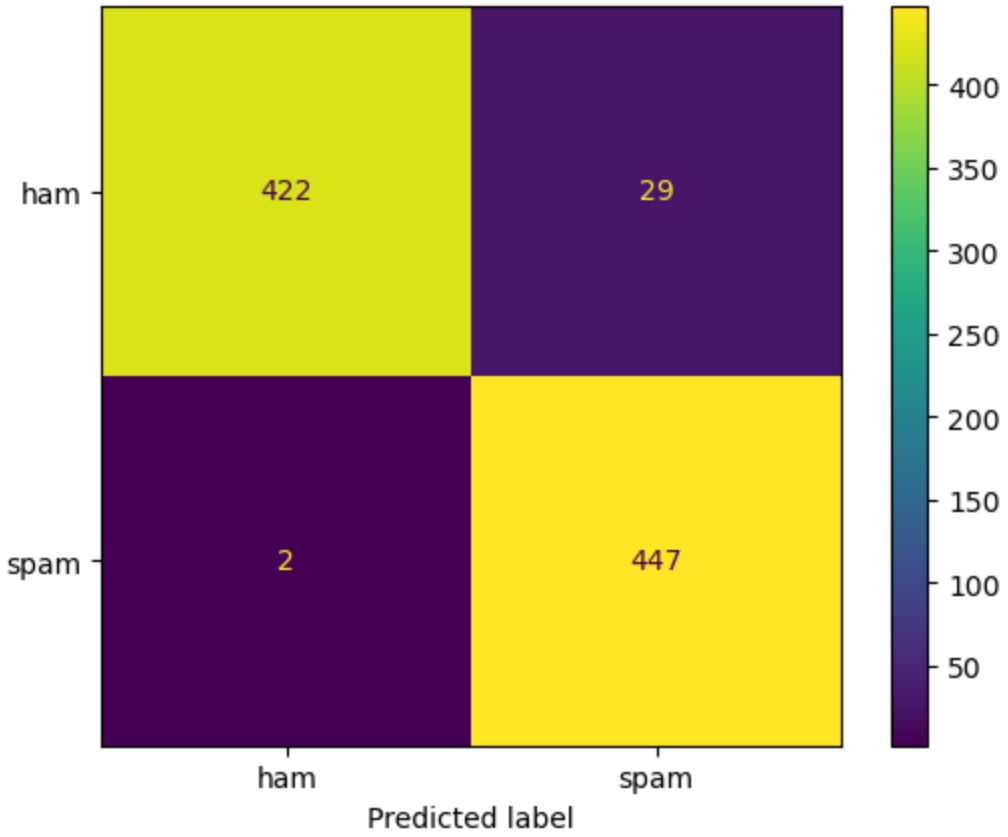
Using the same process, the raw list with stop words removed calculated to a mean accuracy of about 95.44%. This is slightly more accurate than the base list.



Though the accuracy was only marginally improved, the number one distinction for the feature was the word “forwarded’ yet again. With the raw feature sets calculated, we moved to the balanced datasets now. Here are the results below.

```
# perform the cross-validation on the featuresets with word features and generat
num_folds = 5
cross_validation_accuracy(num_folds, balanced_base_featuresets)
```

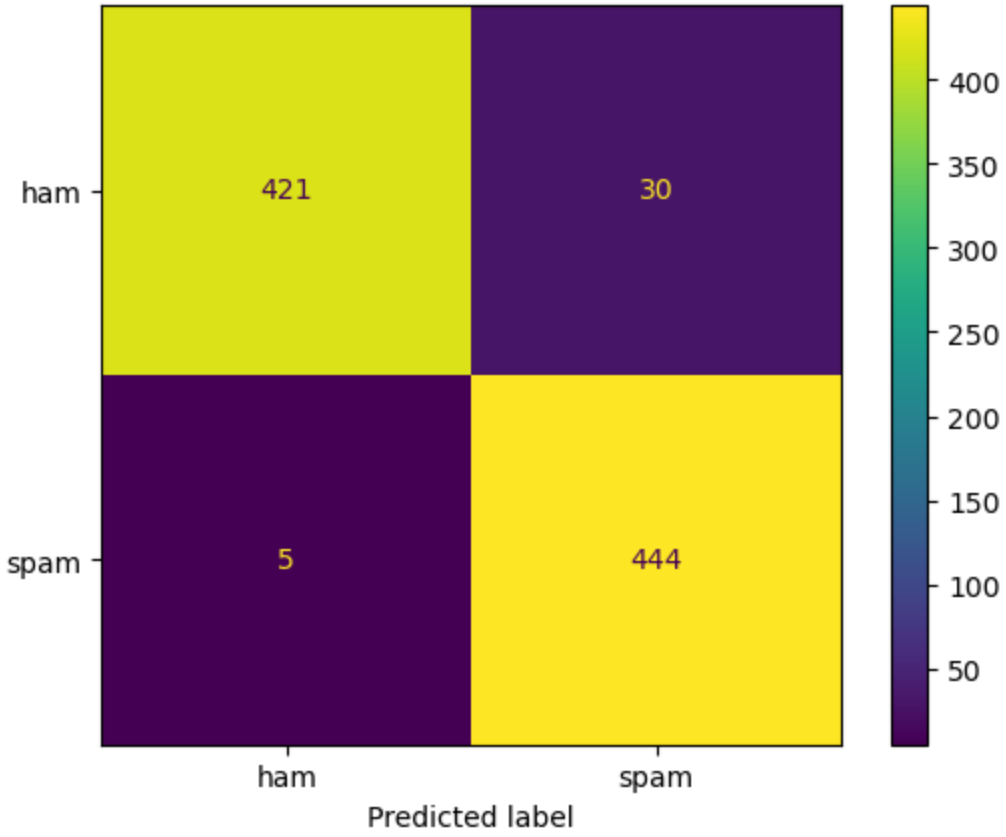
```
Each fold size: 600
0 0.9566666666666667
1 0.9633333333333334
2 0.9583333333333334
3 0.97
4 0.9683333333333334
Mean Accuracy 0.9633333333333335
```



Both images show the code for the balanced data set without stop words removed. The calculation provided a mean accuracy of 96.33%. This means that the balanced data set provided a better accuracy than the two raw data sets.

```
# perform the cross-validation on the featuresets with word features and generat
num_folds = 5
cross_validation_accuracy(num_folds, balanced_sw_featuresets)
```

```
Each fold size: 600
0 0.955
1 0.9666666666666667
2 0.9616666666666667
3 0.9716666666666667
4 0.9616666666666667
Mean Accuracy 0.9633333333333333
```



On the other hand, the accuracy of the balanced dataset with stop words removed provided the exact same accuracy as its counterpart, the balanced data set. This equated to an accuracy of 96.33% as well. In both of these calculations, the most informative features were about equal to one another. This means that there was no significant difference between the two data sets with or without stop words removed.

Conclusion

As clarified before, spam emails have quite the impact on individuals and organizations. These emails can just be incredibly dangerous for any type of facility in various ways. From having information compromised to having data held at ransom for a specific price. In order to try and combat all these various issues, companies had to proceed in taking different steps to identify these malicious emails. Thus, natural language processing is a method to alleviate the blow from either financial loss and/or damage from the organization's reputation.

In this experiment, we sought out four different methods to evaluate emails in the future. The purpose was to create the most accurate model to be able to spot a “spam” email with the purpose of avoiding further tragedies. The four methods utilized were the frequency distribution of unigrams, evaluation of verbs and verb phrases within emails, the punctuation used, and a combination of both verb phrases and punctuation.

Based on our results, we were able to get the most accurate results from the frequency distributions of unigrams. Secondly, using a model based on punctuation was the second most accurate model. Next, the combination of verb phrases and punctuation provided the third most effective model. Lastly, verb and verb phrases had the least accurate score amongst the four methods.

Figuratively speaking, the first process of frequency distribution of unigrams was a broad net that encompassed all attributes of the data provided. Any word or character used in the emails were considered except for the word “Subject’ as each email had that incorporated. We believe that this broad net was the reason why it was the most effective out of all the methods because the model recognized the specific usage of unigrams used between the ham and spam emails. As noted before, ham emails had fewer words and characters than spam emails. By focusing on that detail, the model was able to perform so effectively within this data set.

On the other hand, the other processes were more spearheaded and individualized for a specific purpose. This aspect brought about a less accurate result because it only encompassed a single identity of each email. For example, verb phrases and verbs do not equate to the full usage of a sentence or all the words involved. Same for punctuation as well. Punctuation only focuses on a certain point in the structure of a sentence. By combining the two together, a higher accuracy was calculated.

In conclusion, natural language processing is an incredible tool to try and detect malicious emails. We theorize that a more advanced model can definitely interpret spam emails more effectively. Some ways to enhance this model is by including more areas of grammar into one larger model. For example, taking noun phrases and adjective phrases and meshing them into the feature.

Final Project Split

Coding:

Devyn Hughes - 50%

Sean Deery - 50%

Report:

Devyn Hughes - 50%

Sean Deery - 50%