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**

A screenshot of a computer program

Description automatically generated

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**

***A screen shot of a computer code

Description automatically generated***

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.

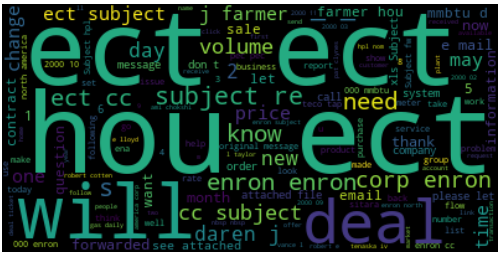
**Example Email**

A screenshot of a computer

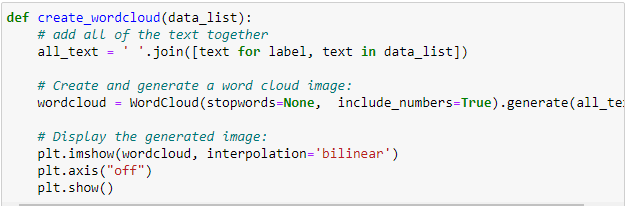
Description automatically generated

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**

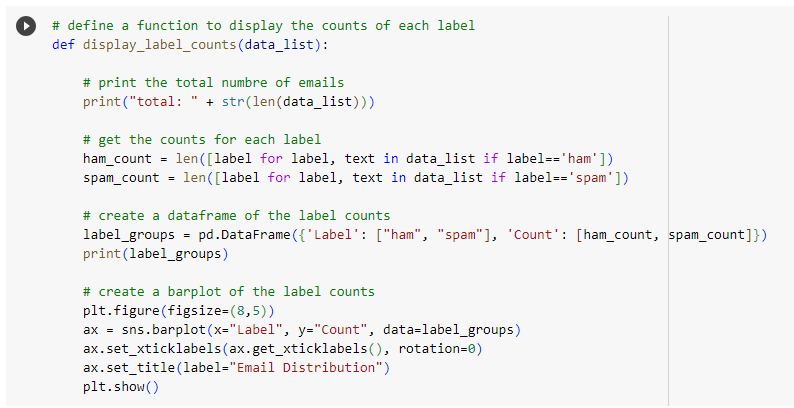
****

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.

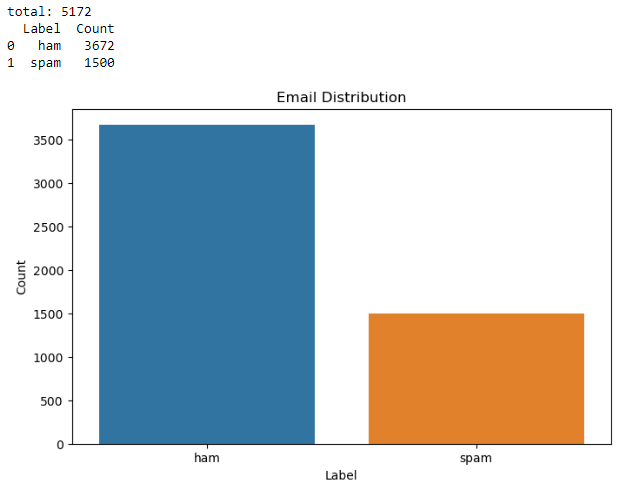


The word cloud is created with a function called **create\_wordcloud**. This function creates a new string of all the words in each of the emails. The string is then passed to the WordCloud generator and then displayed with a bilinear interpolation for a smoother image.

**Counts of Ham and Spam Emails**

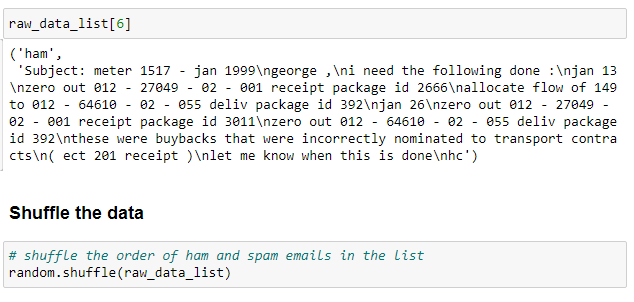
****

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.

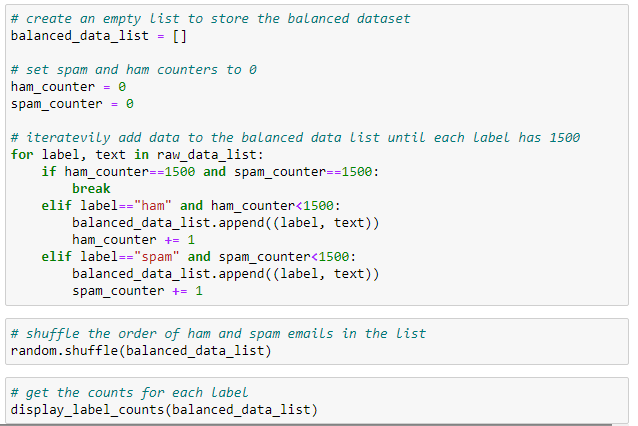


The total number of emails in the data equals 5172. There were 3672 spam emails and 1500 ham emails. The counts and bar plot show the data is unbalanced and could cause an issue with the predictions being biased towards the ham dataset.

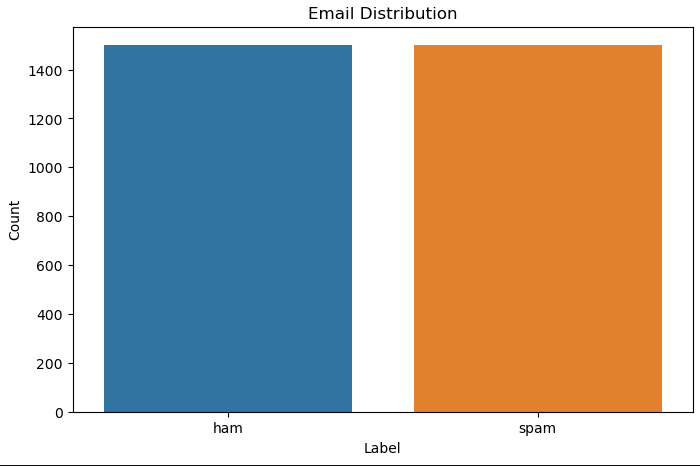
**Data Pre-processing**



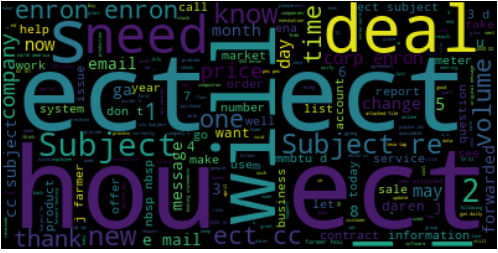
Going into pre-processing our data set, we started by confirming our information was properly listed. Then, we shuffled the information in the **raw\_data\_list**. This was to make sure that the training set and the test set was truly selected at random since the emails between the two labels were listed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



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:

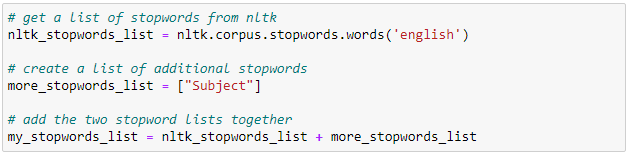


After verifying our balanced data set was complete, we used the same wordcloud function to obtain a different wordcloud.

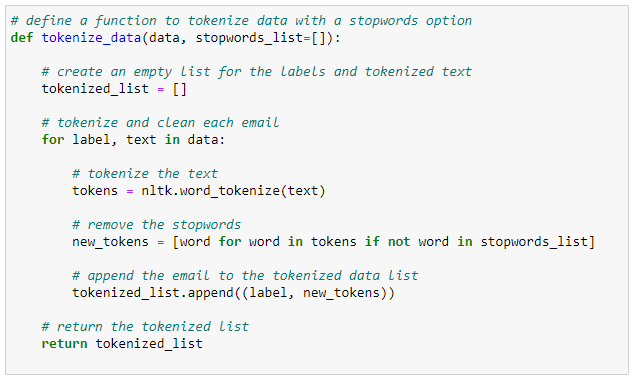


Based on the word cloud created for the raw data list and the balanced list, we can see that some words still occur in high frequency between the two sets. Some noticeable differences is the word “deal” has a higher frequency than how it was used in the raw data set. Also, need was shown to occur more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

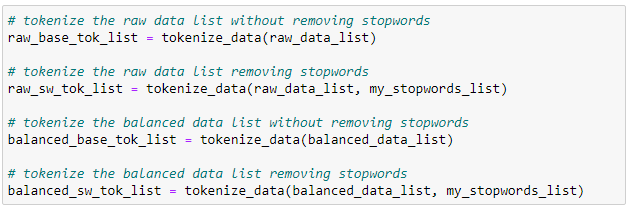
**Tokenizing the Data**



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.



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.



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**



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.



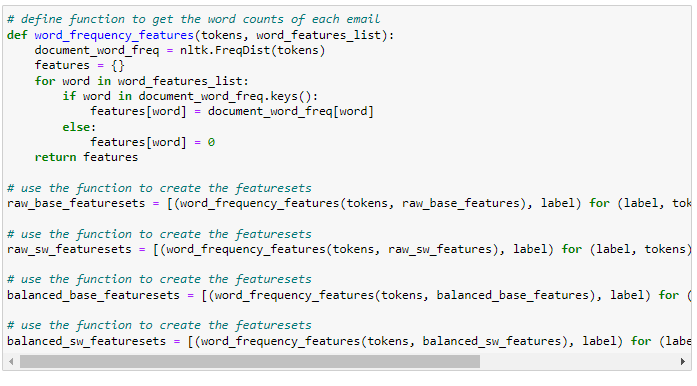
Using the function on the unbalanced tokenized data, we determined a total of 50,566 total words. The five most common words used were actually punctuation marks. We didn’t exclude these from the model because we wanted to test if punctuation helped the model or not. Ham had a total of 20,249 total words and spam had 38,7899. This could mean that ham emails have a more distinct usage of words while spam emails vary greatly.







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.

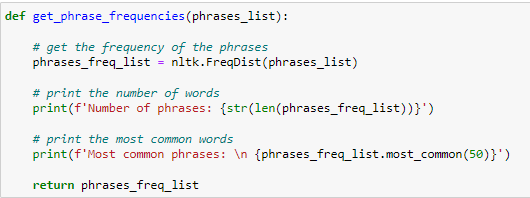


The word\_frequency\_features function is then used to add the counts of words from each email to the features created in the unigram\_features function. The counts of each word in the email is retrieved with FreqDist. The frequencies are then combined with the feature list to create a dictionary of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

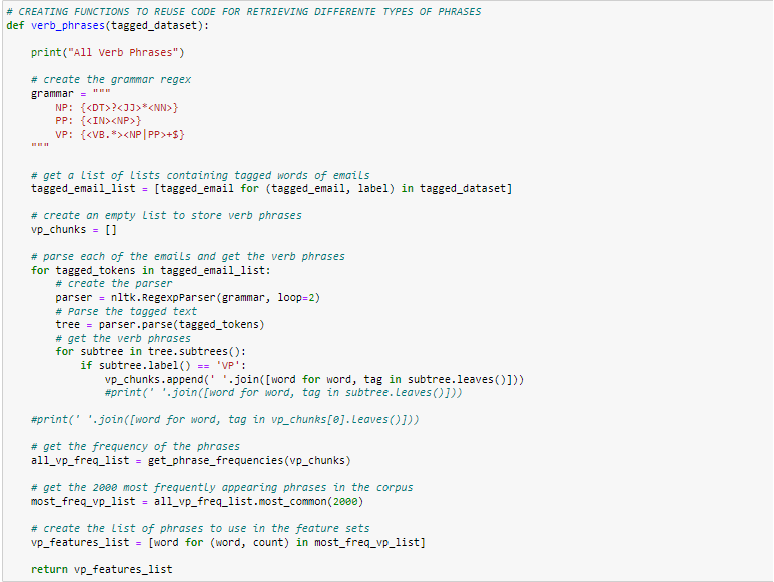
**Verb Phrase Featuresets**



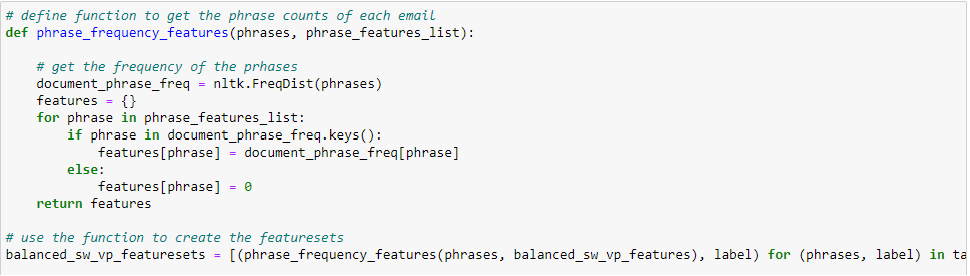
Verb phrase featuresets were then created to capture the predictive quality of calls to action that many spam emails contain. To acquire the verb phrases, the tokens were first tagged with the part of speech tag, resulting in the tagged\_dataset.



The get\_phrase\_frequencies function was then created to assist in getting the verb phrase frequencies just like the unigrams.



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.

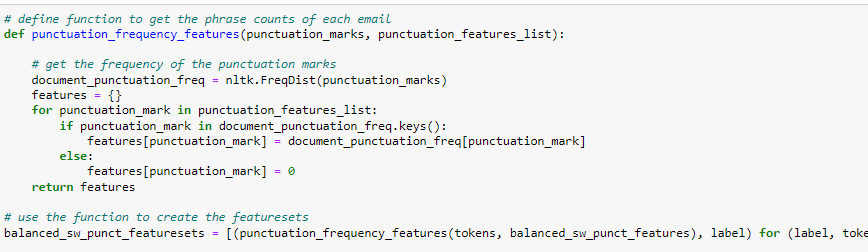


The phrase\_frequency\_features function is then used with the previously created features to assign frequencies from each email.

**Punctuation Featuresets**

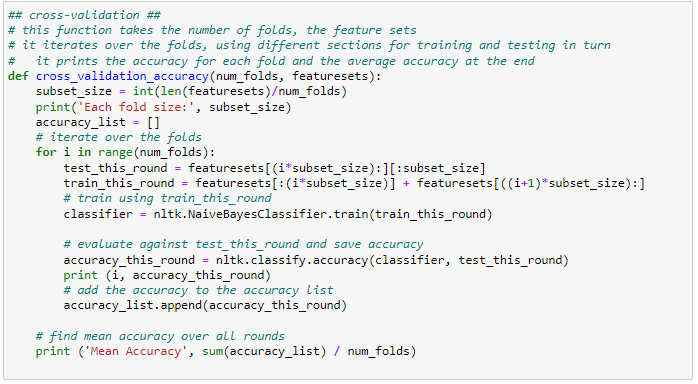


Similar preprocessing steps are taken for punctuation featuresets. The punctuation\_features function uses the get\_punctuation\_frequencies to get the list of punctuation marks used in the dataset.



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.

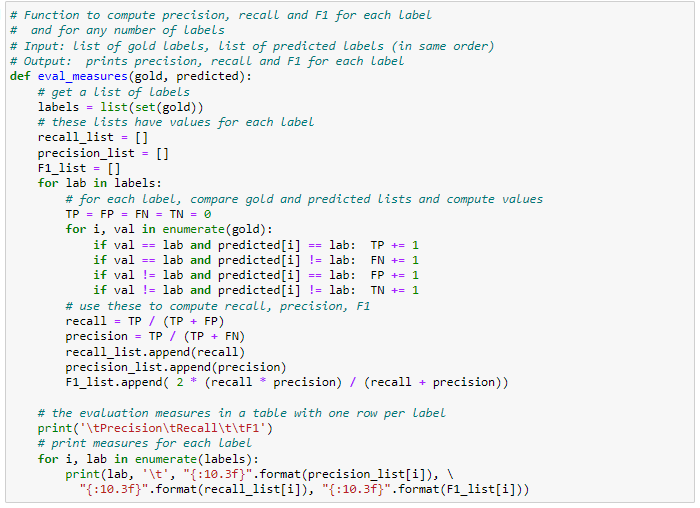
**Classification Functions**



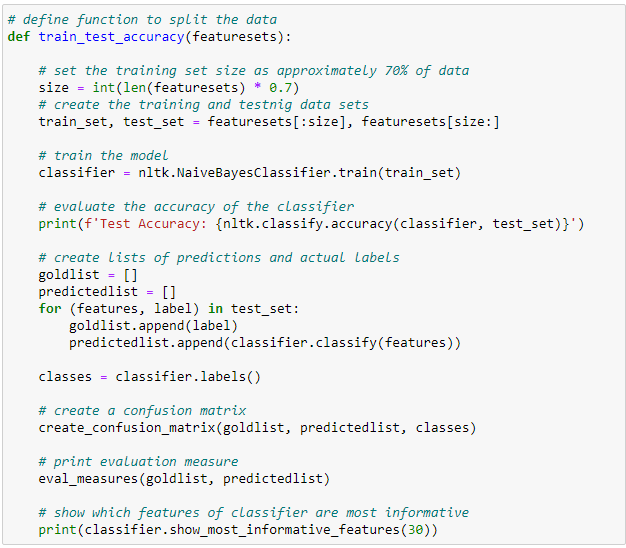
For this experiment, we utilized a Cross Validation process. Starting off, we created a function called **cross\_validation\_accuracy**. In order to do so we created a subset size by taking the length of the featureset and dividing it by the number of folds. Then, we made an empty list called **accuracy\_list**. 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.



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.

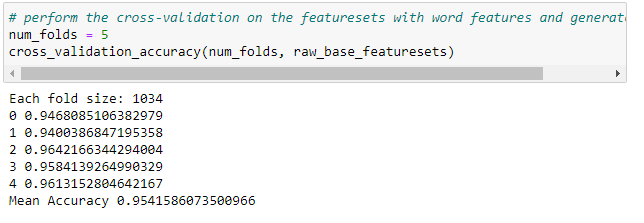


The eval\_measures function is also used to extract and print out the precision, recall, and f1 scores. A high precision means the model returns more relevant results and less irrelevant ones for stocks. A high recall means the model returns most of the relevant results whether or not irrelevant ones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

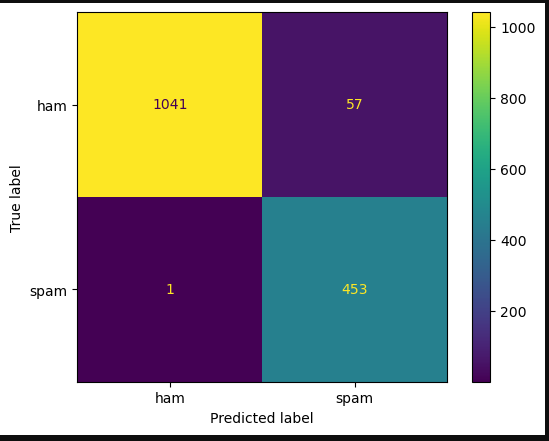


In this code, we created a function to split the data accordingly. Naming the function **train\_test\_accuracy**, it considers any feature sets created as an input. For this experiment, we utilized 70% of the data within a set called **size**. From here we divided the data under **train\_set** and **test\_set**. 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.

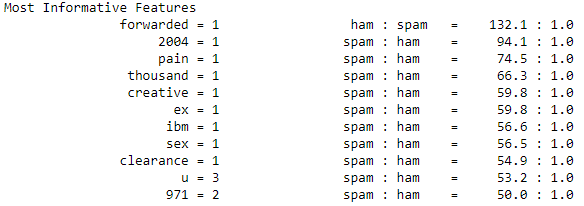
**Classification Results**

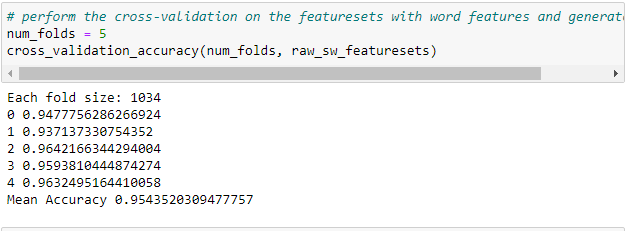


With everything set up, we began our first calculation of accuracy. using the **cross\_validation\_accuracy** function we created. Setting the number of folds to 5, we utilized the function on the **raw\_base\_featuresets**. This was the base feature set without anything removed it. The accuracy of prediction 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.

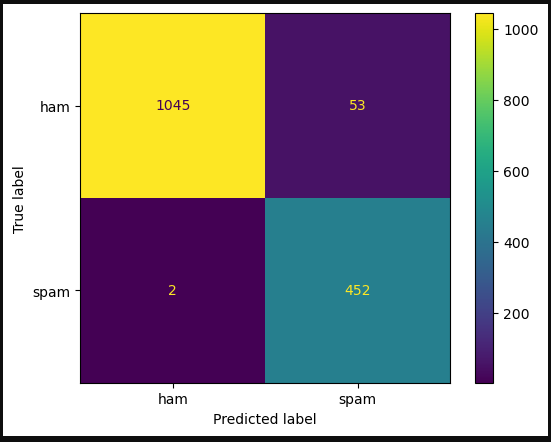


Based on the most informative features, “forwarded” was the top feature for this prediction. When the word “forwarded” occurred once, then there was a ratio of 132:1 in favor of a “ham” email.

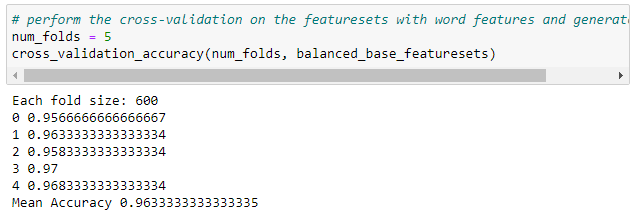


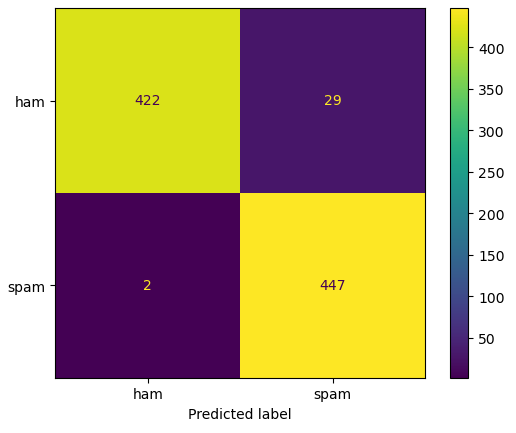


Using the same process, the raw list with stop words removed calculated to a mean accuracy of about 95.44%. This is slightly more accurate than the base list.

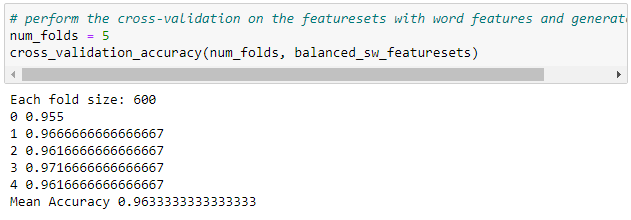


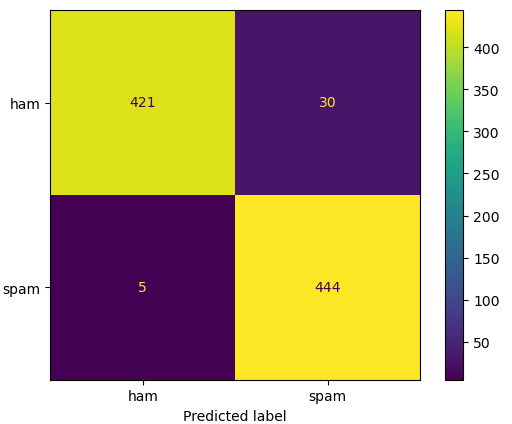
Though the accuracy was only marginally improved, the number one distinction for the feature was the word “forwarded’ yet again. With the raw feature sets calculated, we moved to the balanced datasets now. Here are the results below.





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.





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%**