Automated Spam Email Detection Using Natural Language Processing Techniques

Max McFadden

Overview

For this project, I used a dataset containing the contents of a total of 5172 emails, with 1500 spam emails, and 3672 non-spam emails (ham). This dataset was obtained from the Enron public email corpus, available on their website at https://pages.aueb.gr/users/ion/data/enron-spam/. Each email is in a designated folder, indicating whether it is spam or "ham". The objective of my research is to create a classifier to detect spam emails, using only the string-contents of the mail.

There was a program provided with this dataset which does a simple reading and labeling of the data. When running this program, the user must input a number which will determine how many spam and ham emails will processed – this will be referred to as the limit number. In this preliminary program, the limit number of ham and spam emails are read, converted into strings, tokenized, and placed into a tuple list called "emaildocs". Each tuple consists of two parts – the first being the list of word-tokens obtained from the email, and the second being a label indicating the true classification of the email (spam or ham).

Data Pre-Processing and Filtering

In an attempt to better prepare the data for classification, I added a few pre-processing methods and filters. Firstly, I converted all characters to lowercase. This was to ensure that the classifier does not treat the same words as differently based on case. This was most likely not necessary, as it appears as though the contents of the emails are all in lowercase already, with the only uppercase characters being found in the email metadata (i.e. "Subject:"), but I converted all contents to lowercase just to ensure all emails were uniformly lower. I did so using the lower() function for all tokens, in an enumerated for loop where I would modify the contents of the token list, then update the corresponding emaildocs item with the updated data.

I also filtered out a list of common stop words (like 'the', 'is', 'and') since they occur frequently in both spam and ham emails and typically don't carry significant distinguishing information. I did this by using the NLTK English list of stop words, removing all tokens which matched a word in that list. This had a major effect on the sample emails that I observed, and made it arguably more difficult to distinguish ham from spam from a brief observation.

My final addition to the pre-processing of the text was to remove email-specific jargon such as 'Subject' and 'cc'. These tokens are essentially the metadata of the raw email-text being extracted, which likely would not be useful for classification as it is roughly equally present in both spam and ham emails. I did this by simply adding these to my list of stop words which were already being removed.

I considered removed tokens which were exclusively special characters. I thought the noise added by having many special-character tokens could potentially outweigh the distinguishing

information added by these characters. I initially this by iterating through the tokens, only keeping the ones which matched the regular expression "^[^A-Za-z]+$ ", which will match any string that does not contain any letter from A to Z (in both uppercase and lowercase), ensuring that the string consists exclusively of non-alphabetical characters. However, after adding this feature, I looked back on some of the sample emails that I had generated when running my program, and noticed that many spam emails had extremely heavy use of non-numerical tokens. Because these tokens were seldom used heavily in ham, I decided that the presence of many special characters was likely to provide distinguishing information, which will help our classification model identify spam emails.

Text Feature Production using NLTK

Next, I used the NLTK (Natural Language Tool Kit) package in python to produce the basic text features of the collected data. My first objective was to obtain a list of the most commonly used tokens for spam and ham emails, respectively, normalized by the length of the documents. In order to obtain these using the frequency distribution feature in the NLTK package, I first had to split the emaildocs list into respective ham and spam lists, so that I could obtain the word frequency for each type of email. Since each of these two token lists would contain exclusively email token lists of a single label (i.e. all tuples in the spam tokens list would have a label value of 'spam'), I also removed the label portion of the tuple. This left me with a doubly nested list, where each element in the list contained a sublist of all of the string tokens contained in the email referred to by that element. In order to easily obtain total document length, and to make the tokens compatible with the FreqDist features in the NLTK package, I would have to convert this doubly nested list into a standard, un-nested list. This list would consist of a stream of elements, where each element is a string token. The list would be unbroken by the end of one email and the beginning of another (i.e. [["email", "one"] , ["email", "two"]] would now be stored as ["email", "one", "email", "two"]). In order to do so, I used a nested list comprehension statement.

Next, I would use the features of the FreqDist portion of the NLTK package to obtain and display the most commonly used tokens in spam and ham emails, respectively. I obtained full frequency distributions using the FreqDist() function on my newly constructed un-nested lists. I then obtained the n most commonly used terms from each distribution, using the '.most_common(n)' method on my frequency distribution objects. In order to normalize the most commonly used terms by the total length of each list (i.e. the total number of tokens in each of the un-nested token lists), I made variables containing the length value for each of the two lists. I then printed the 50 most commonly used terms for each – ham and spam. I noticed that the majority of the top 10-15 terms for both spam and ham emails were special characters.

Next, I wanted to find the most frequently occurring bigrams in either type of email. To do this, I used the Bigram Associative Measures Collocation from the NLTK package. I first made a bigram measures variable using the '.BigramAssocMeasures()' method. I then used the '.from_words()' method from the BigramCollocationFinder object to obtain my list of bigrams for either of the two sets of unigrams. I then scored these by frequency using the '.score_ngrams' method on my lists of bigrams, printing the results for observation.

Almost all of the most common bigrams found were either involving, or exclusively, punctuation/special characters. Because of this, I thought it may be useful to obtain the list of bigrams with the highest mutual information score for either type of email. To obtain the bigrams with highest mutual information score for either set of tokens, I made another list of bigrams for each set, and applied a frequency filter to each of these sets so that it only contained bigrams which appear at least a specified number of times. A good value for this filter will depend on the number of emails being read which is specified when running the function – *this is an area that may be useful to experiment with*. I then used the '.score_ngrams()' method again, this time with the argument 'bigram_measures.pmi', to obtain the list of bigrams sorted my mutual information score. A number of these bigrams are then printed below for observation.

Naïve Bayes Classification

I then used the Naïve Bayes classifier included in the NLTK package to train and test a classifier on my feature sets. First, I had to create a feature extraction function, which would take a document (i.e. list of tokens), and a set of word features as input. It would iterate through the document to determine which of the words in the feature list were present in that document, and return a list containing true or false values indicating whether a feature word was present in that document. In order to maintain maximum distinguishability between features, I kept all spam unigrams, ham unigrams, spam bigrams, and ham bigrams in separate lists from each other, and wrote my feature extraction function such that each of these lists were separately tested for presence in the document.

The feature extraction function is named find_features(), and takes 5 arguments as input - document, spam_unigrams, ham_unigrams, spam_bigrams, ham_bigrams. The function converts the document (email) into a set, such that it can be searched more efficiently, and each term only appears once – this will be iterated through for the unigram features. Another list is made by converting the list of bigrams in the document into a set – this will be iterated through for the bigram features. As each feature is iterated through, it is assigned either a true or false value, based on whether or not that feature is present in the document. The output is a list of Boolean values associated with each of the features in the feature set, corresponding to the presence of features in that particular document.

Using list comprehension, I created the input lists to fill in the other 4 arguments in the find_features function - spam_unigrams, ham_unigrams, spam_bigrams, ham_bigrams. The spam_unigrams and ham_unigrams lists were made by simply extracting the first part of each tuple in the list of most common unigrams, thus excluding the part of the tuple which contains the number of occurences. The spam_bigrams and ham_bigrams lists were made similarly, by extracting the first part of each tuple in the list of most common bigrams, and concatenating the first part of each tuple in the list of bigrams sorted by mutual information score. I obtained a list of all feature sets by using a list comprehension that produced a tuple containing the feature set as the first element by calling the find_features function, and the label as the second element.

For cross validation, I made a for loop which ran once for each fold – in my case I used 5 folds. Essentially, the data was split into 5 equal parts, and each part is used one time as the testing data while the other 4 are used as training. The accuracy, precision, Recall, and F-measures are

recorded each time the classifier is run on a fold, and averaged at the end to give a more accurate and reliable set of values.

Feature Experimentation

For all of my feature experiments, I used 1000 emails of each, spam and ham.

My initial feature set was the top 100 most frequent unigrams for spam and ham, respectively, the top 50 most frequent bigrams for spam and ham, respectively, and the top 100 highest mutual information scored bigrams with a frequency minimum of 10. The results were as such:

Average Accuracy: 0.9195
Average Precision: 0.8620039004677068
Average Recall: 0.9979213907785336
Average F-measure: 0.9246974047882537

The accuracy seems fairly high from my perspective, especially with a feature set that isn't overly large. To get a better understanding of the significance of the frequent bigrams, versus the bigrams with high mutual information, I tried out my classifier omitting the high mutual information bigrams, keeping everything else equal. The results were as such:

Average Accuracy: 0.9145
Average Precision: 0.8559796819818761
Average Recall: 0.9970022458744264
Average F-measure: 0.9209722316761996

While the accuracy, precision, recall, F-Measure were all slightly lower, the change was minimal. Because of this result, I decided to add back in the bigrams with high mutual information, and omit the ones with high frequency. These were the results:

Average Accuracy: 0.9339999999999999
Average Precision: 0.8894016516428673
Average Recall: 0.9912349317163267
Average F-measure: 0.9374295913425421

Interestingly, the accuracy, precision, and F-Measure actually increased with the omission of the most frequent bigrams. This indicates that the highly frequent bigrams may have been adding significant noise to the classifier, and not providing enough distinguishing information to make up for that noise. As such they decreased accuracy. In order to see if any significant gain can be made from having frequently used bigrams, I will re-implement the frequent bigrams, reducing the number of frequent bigrams to the top 10. These are the results:

Average Accuracy: 0.9269999999999999
Average Precision: 0.874899486750069
Average Recall: 0.9958600960759971

Average F-measure: 0.9314030612003551

As you can see, the accuracy, precision, and F-measures were again lower than they were without the frequent bigrams altogether. Because of this I have concluded that the classifier will perform better without using these most frequently found bigrams altogether. Next, I will omit the frequently found bigrams, and experiment with the frequency limit (i.e. minimum number of occurrences) of the list of bigrams sorted by mutual information score. I will first attempt to increase the frequency limit from 10 to 20. These are the results:

Average Accuracy: 0.9359999999999999
Average Precision: 0.8912431613309824
Average Recall: 0.9930484712211148
Average F-measure: 0.9392228853715215

These are the highest accuracy, precision, and F-measure results obtained yet. Thus, by using only high mutual information bigrams which occur more frequently than before, our classifier is better able to distinguish spam mail from ham mail. Finally, I wanted to experiment with the number of unigram features. Leaving all else the same from the previous iteration, I will increase the number of terms from 100 to 200. These are the results:

Average Accuracy: 0.938
Average Precision: 0.8923998387173416
Average Recall: 0.9948955668414431
Average F-measure: 0.9407612467091925

By increasing the number of frequently occurring unigrams from 100 to 200, the accuracy, precision, Recall, and F-measure all increased slightly. In order to test whether this will keep enhancing the accuracy of the model, I will try drastically changing the number of frequently occurring unigrams from 200 to 1000. These are the results:

Average Accuracy: 0.9595
Average Precision: 0.926941437782698
Average Recall: 0.9969376229456184
Average F-measure: 0.9606249038263245

This yielded significantly higher accuracy, precision, recall, and F-measure. This leads me to believe that the unigram frequency is doing the vast majority of the distinguishing between types of email in the classifier. With a drastic increase in number of unigrams used in the feature list leading to a dramatic increase in performance of the classifier, it seems as though the limiting factor in adding more unigrams to the feature list is simply the computational efficiency of the classifier, as this did take significantly longer to run than any of the previous iterations.