**NLP Final Project: Detection of SPAM in email**

I chose to use the Enron SPAM dataset for the final project. I will move through each step as the professor outlined in the assignment pdf; however, given the nature of the dataset, I will have to revisit the previous steps in preprocessing, sample sizing, etc. in order for some of the NLTK features to work appropriately. This project will reveal how extensive preprocessing can eliminate key clues that are needed for detection of SPAM vs HAM emails. I was able to achieve 98.1% accuracy using the NLTK Naïve Bayes classifier! I have saved somewhat of a “grand finale” for the end using multiple classifiers in the sklearn package that also yielded some impressive results. I will do my best to present the findings as clearly and concisely as possible using tables, screenshots, and descriptions of code used and the results that followed. The key points in the description of the code will be highlighted in the tables for easy identification.

**Step 1: Get Data / Select Size / Preprocess / Investigate Data**

Step 1a: Read in file/Select document size

I began the project by dissecting the code given in class for uploading, processing, and classifying the email text. I found that braking up the large code block into small chunks allowed for making easy “minor” adjustments. I started the project by loading in all of the documents by setting the limit at 3672 (the maximum):

|  |
| --- |
| Grab the Documents / Select Document Amount Code |
| processspamham(dirPath, 1500)  # start lists for spam and ham email texts  limit = 3672  #allfiles = 3672  hamtexts = []  spamtexts = []  os.chdir(dirPath)  # process all files in directory that end in .txt up to the limit  # assuming that the emails are sufficiently randomized  for file in os.listdir("./spam"):  if (file.endswith(".txt")) and (len(spamtexts) < limit):  # open file for reading and read entire file into a string  f = open("./spam/"+file, 'r', encoding="latin-1")  spamtexts.append (f.read())  f.close()  for file in os.listdir("./ham"):  if (file.endswith(".txt")) and (len(hamtexts) < limit):  # open file for reading and read entire file into a string  f = open("./ham/"+file, 'r', encoding="latin-1")  hamtexts.append (f.read())  f.close() |
| Result |
| A screenshot of a cell phone  Description automatically generated |

Step 1b: Preprocessing / Filtering / Data Investigation

After compiling the documents, I did a basic word tokenize of spam and ham and appended them both into “emaildocs”. Next, I set a seed of 123 for result reproducibility and executed a random shuffle of the documents for testing. Lastly, I printed an email to investigate what type of data/emails I am working with:

|  |
| --- |
| Tokenize / Shuffle / Document Creation Code |
| # create list of mixed spam and ham email documents as (list of words, label)  emaildocs = []  # add all the spam  for spam in spamtexts:  tokens = nltk.word\_tokenize(spam)  emaildocs.append((tokens, 'spam'))  # add all the regular emails  for ham in hamtexts:  tokens = nltk.word\_tokenize(ham)  emaildocs.append((tokens, 'ham'))  # randomize the list  random.Random(123).shuffle(emaildocs) |
| Result |
| A screenshot of a cell phone  Description automatically generated |

An additional step -- I created an “all words list”, placed the words into lowercasing, and made a frequency distribution:

|  |
| --- |
| Frequency Distribution Code |
| # get all words from all emails and put into a frequency distribution  # note lowercase, but no stemming or stopwords  all\_words\_list = [word for (sent,cat) in emaildocs for word in sent]  all\_words = nltk.FreqDist(all\_words\_list)  print((all\_words)) |
| Result |
|  |

**Step 2: Bag-of-Words / Produce Features / Create a baseline experiment / Evaluation Measures**

For Step 2 of the project, I will create a bag of words and find an appropriate number of vocabularies for optimal results. I believe that experimenting with different vocabulary sizes will also satisfy some of the requirements of “Step 3: Experiments”, and for the sake of avoiding redundancy and confusion later, I will include it here.

Step2a: Bag-of-Words

Here I will show the code used to create a bag-of-words for feature creation and future classification. I started with the class code that used 1500 most frequently used keywords. Later, I will prove how choosing vocabulary size is much like Goldilocks picking hot and cold porridge – there is a number in the middle that is just right.

|  |
| --- |
| Feature Amount Code |
| # get the 1500 most frequently appearing keywords in the corpus  word\_items = all\_words.most\_common(1500) #1500 words  word\_features = [word for (word,count) in word\_items] |
| Result |
| A screenshot of a cell phone  Description automatically generated |

We see that for the baseline corpus the most common word features are punctuation and symbols. It can be first instinct to remove these with stop words, but these are actually beneficial to predicting types of emails. Don’t worry, this will be proven later!

Step2b: Produce Features

This is the code used for creating a baseline feature that uses Boolean logic to identify if a word is present or not in an email. The “featuresets” code is also included that is used to train and test later:

|  |
| --- |
| Feature Function Code |
| A screenshot of a cell phone  Description automatically generated |

Step2c: Create a Baseline Experiment

Finally, some fun! I used code to divide the feature set into training and testing with roughly 80% of the data falling into the training. The rest of the data will be used to test accuracy. To test, the following nltk.classify code was used with the classifier based on the training set. This produced a 94.4% accuracy on the data – not a bad start.

|  |
| --- |
| Training/Testing/Classifier Code |
| # training using naive Baysian classifier, training set is roughly 80% of the data  train\_set, test\_set = featuresets[1000:], featuresets[:1000]  classifier = nltk.NaiveBayesClassifier.train(train\_set) |
| Code (80/20 split) |
| print(len(featuresets))  print(len(train\_set))  print(len(test\_set)) |
| Result |
| A close up of a logo  Description automatically generated |
| Code |
| # evaluate the accuracy of the classifier  nltk.classify.accuracy(classifier, test\_set) |
| Result |
| A screenshot of a cell phone  Description automatically generated |

Step2d: Evaluation Meausures

The evaluation measures below utilize the code from class. I did a cross-validation using five folds, a confusion matrix to better understand how the classifier was performing, and a function to provide precision, recall, and f1 values. Finally, I called the classifier to reveal the most informative features to fully understand what matters most when classifying emails using bag-of-words.

|  |
| --- |
| Cross-validation & Folds Code |
| 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)  accuracy\_list.append(accuracy\_this\_round)  # find mean accuracy over all rounds  print ('mean accuracy', sum(accuracy\_list) / num\_folds)  # perform the cross-validation on the featuresets with word features and generate accuracy  num\_folds = 5  cross\_validation\_accuracy(num\_folds, featuresets) |
| Results |
| A close up of a logo  Description automatically generated |

After five folds we can see that the mean accuracy of this particular classifier was 94.1%. Next we will move on to the confusion matrix:

|  |
| --- |
| Confusion Matrix Code |
| goldlist = []  predictedlist = []  for (features, label) in test\_set:  goldlist.append(label)  predictedlist.append(classifier.classify(features))  # look at the first 10 examples  print(goldlist[:15])  print(predictedlist[:15])  cm = nltk.ConfusionMatrix(goldlist, predictedlist)  print(cm.pretty\_format(sort\_by\_count=True, truncate=9)) |
| Results |
| A screenshot of a cell phone  Description automatically generated |

It looks like our classifier is great at distinguishing what is spam but is sometimes lumping ham in with the spam (sounds like I’m talking about dinner and not data). Of the 1000, it only miscategorized 56 hams as spam and predicted spam perfectly. We can easily double-check this by subtracting 1.0 – 0.056 to get back to our original accuracy of 0.944. Let’s move forward to additional evaluation measures:

|  |
| --- |
| Precision/Recall/F1 Code |
| 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]))  # call the function with our data  eval\_measures(goldlist, predictedlist) |
| Results |
| A close up of a logo  Description automatically generated |

We have an inverse between ham and spam in terms of predicting false positives (precision) and false negatives (recall). When the model is predicting spam, it is labeling ham as “spam”, which could lead to non-spam emails ending up in the spam folder – not great (this happens to me in the real world too). Furthermore, we are seeing some False Negatives with spam (recall). In the end, our F1 score gives a slight leg up to ham predictions as opposed to spam. We will see how adding additional features to our bag-of-words feature set will help this.

|  |
| --- |
| Informative Features Code |
| print(classifier.show\_most\_informative\_features(10)) |
| Results |
| A screenshot of a cell phone  Description automatically generated |

Looking at the top 10 most informative features it looks like a lot of spam emails are associated with the body: weight, health, pain, sex. Oddly, some spam emails actually included the word “spam” and the word “forwarded” is a strong feature for distinguishing ham – this is not my personal experience, but times have changed since this email corpus was created.

(Spoiler Alert) Later, we will see how further preprocessing can hinder the performance of the model. But first we will experiment with vocabulary size and total number of documents.

**Step 3: Experiments**

Step3a: Making a better base model by changing document size and feature size (vocabulary)

After reflecting over the baseline model performance, I wondered if the ratio of ham vs spam documents could contribute to incorrectly labeling ham as spam. The original limit set utilized all documents (ham: 3672, spam: 1500), and maybe the extra hams are creating a prediction skew. For this experiment, I changed the limit number to give an equal 1500 documents of ham/spam. I also changed the number of features to 4000 from the original 1500. This took a lot of trial and error and to prevent this paper from becoming excessive in length will only show the final product with explanation.

|  |
| --- |
| Changing the Limit Code |
| processspamham(dirPath, 1500)  # start lists for spam and ham email texts  limit = 1500  #allfiles = 3672  hamtexts = []  spamtexts = []  os.chdir(dirPath)  # process all files in directory that end in .txt up to the limit  # assuming that the emails are sufficiently randomized  for file in os.listdir("./spam"):  if (file.endswith(".txt")) and (len(spamtexts) < limit):  # open file for reading and read entire file into a string  f = open("./spam/"+file, 'r', encoding="latin-1")  spamtexts.append (f.read())  f.close()  for file in os.listdir("./ham"):  if (file.endswith(".txt")) and (len(hamtexts) < limit):  # open file for reading and read entire file into a string  f = open("./ham/"+file, 'r', encoding="latin-1")  hamtexts.append (f.read())  f.close() |
| Results |
| A screenshot of a cell phone  Description automatically generated |
| Explanation |
| Here I set the limit to only grab 1500 ham emails to create an even number between ham and spam |

|  |
| --- |
| Upping the Features Code |
| # get the 1500 most frequently appearing keywords in the corpus  word\_items = all\_words.most\_common(4000)  word\_features = [word for (word,count) in word\_items]  test\_features = [word for (word,count) in emaildocs] |
| Results |
| See Next Table |
| Explanation |
| Upping the word features to 4000 from 1500 gives the model more features to go off of. 4000 seems to be that Goldilocks number I referenced earlier – anything more or less decreases model performance. This can be explained by considering our own minds; too much information can weigh us down the same as not knowing enough can make us ill-informed. |

|  |
| --- |
| New Splitting / New Training / New Classifying Code |
| # training using naive Baysian classifier, training set is roughly 80% of the data  train\_set, test\_set = featuresets[600:], featuresets[:600]  classifier = nltk.NaiveBayesClassifier.train(train\_set) |
| Results |
| A screenshot of a cell phone  Description automatically generated |
| Explanation |
| Here I needed to change the featureset split ratios inorder to get the 80/20 split. When left alone, it is more of a two-thirds/one-thirds split, which wasn’t enough training and resulted in a lower accuracy. Once the 80/20 was reachieved, the accuracy was quite high! |

|  |
| --- |
| Reevaluating the “new” bag-of-words model |
| A screenshot of a cell phone  Description automatically generated |
| Explanation |
| The 5-fold cross-validation was a little disappointing in that the model is likely not as high as 98.1%; however this is still an improvement from the base model. |
| Confusion Matrix |
| A screenshot of a cell phone  Description automatically generated |
| Precision, Recall, F1 |
| A screenshot of a cell phone  Description automatically generated |
| There was some sacrifice in Precision in Ham and Recall in Spam; however, the overall model is performing much better based on the F1 scores!  The precision on ham was my main conscern and the model was able to increase by 3.5 percentage points meaning that less ham emails will end up as spam. |

Step3a continued: Preprocess with Removing Stop Words, Symbols, and Punctuation

This experiment was created to understand if stop words, symbols, and punctuation have an effect on how the model predicts. It does. I employed the stop word function built-in to NLTK and also added additional stop words to the corpus. This new model was trained and tested on the newly improved bag-of-words feature model from Step3a.

|  |
| --- |
| Stop Word Corpus Code |
| A close up of a logo  Description automatically generated |
| Results |
|  |
| Explanation |
| We see that the total number of words have dropped by employing the stop word corpus from 44574 to 44399. This is not a huge drop, but this small drop shows how stop words help with prediction of spam emails |

|  |
| --- |
| Stop Feature Set / Train Test / Classify Code |
| A screenshot of a cell phone  Description automatically generated |
| Explanation |
| We see that the use of stop words dropped the prediction down 3 points.  I believe the reason is that stop words and unique symbols are used more heavily in spam emails.  Below, we can see that the accuracy is even less using cross-validation and other evaluation metrics. |
| Cross-Validation |
| A screenshot of a cell phone  Description automatically generated |
| Precision, Recall, F1 |
| A screenshot of a cell phone  Description automatically generated |

Step 3a continued: Subjectivity

I thought it would be interesting to see how subjectivity plays a role in spam vs ham emails. This required some additional work, but the ending result was insightful (even though the accuracy wasn’t great).

|  |
| --- |
| Subjectivity Code |
| from nltk.corpus import sentence\_polarity  def readSubjectivity(path):  flexicon = open(path, 'r')  # initialize an empty dictionary  sldict = { }  for line in flexicon:  fields = line.split() # default is to split on whitespace  # split each field on the '=' and keep the second part as the value  strength = fields[0].split("=")[1]  word = fields[2].split("=")[1]  posTag = fields[3].split("=")[1]  stemmed = fields[4].split("=")[1]  polarity = fields[5].split("=")[1]  if (stemmed == 'y'):  isStemmed = True  else:  isStemmed = False  # put a dictionary entry with the word as the keyword  # and a list of the other values  sldict[word] = [strength, posTag, isStemmed, polarity]  return sldict  SLpath = "/Users/wa3/Syracuse/Term6/NLP/Week8/subjclueslen1-HLTEMNLP05.tff"  SL = readSubjectivity(SLpath)  def SL\_features(document, SL):  document\_words = set(document)  features = {}    # count variables for the 4 classes of subjectivity  weakPos = 0  strongPos = 0  weakNeg = 0  strongNeg = 0  for word in document\_words:  if word in SL:  strength, posTag, isStemmed, polarity = SL[word]  if strength == 'weaksubj' and polarity == 'positive':  weakPos += 1  if strength == 'strongsubj' and polarity == 'positive':  strongPos += 1  if strength == 'weaksubj' and polarity == 'negative':  weakNeg += 1  if strength == 'strongsubj' and polarity == 'negative':  strongNeg += 1  features['positivecount'] = weakPos + (2 \* strongPos)  features['negativecount'] = weakNeg + (2 \* strongNeg)  return features |
| Results |
| A screenshot of a cell phone  Description automatically generated |
| Explanation |
| The subjectivity classifier did not perform as well as I had originally hoped at 66.1%.  I did not feel the need to go through extensive evaluation measures for this one due to the fact that it does not add much of a solution. It is interesting; however, to see the informative features.  If you dig into the numbers, you will find that most spam is either highly positive or somewhat negative. My personal guess is that spam tries to use excitement and/or fear in the emails to elicit an urgency to click on the links to the ads.  Below are the top 25 features: |
| A screenshot of a cell phone  Description automatically generated |

Step3a continued: Define Your Own Feature Function – Word Counts/Frequency

For this one I had to greatly reduce the number of documents for my computer to run a word count of each email. In the final project it states that we need to define a feature function not given in class. I took it to mean that we needed to create one, but maybe using SciKit Learn counts as well. Either way, I will include both. The word count took so long that I was unable to get anything to run other than the basics. Here are screenshots of the output including the informative features. You can see that this was not a successful classifier:

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

In order to write this new feature function I needed to create “word\_features.count(word)” as seen in the first screenshot. I also had to write the “featuresets = “ on the “all\_words\_list” as opposed to the frequency distribution used for the base bag-of-words model. Being new to this, I hoped to write some beautiful code that created something interesting, but I think this is about as good as I can make it with the computing power in my home.

Step3b Sci Kit Learn:

This was one of the most satisfying aspects of the project! I discovered how to implement the sklearn packages through the nltk and used the base model setup to implement multiple models. I implemented 8 models that I will display below, and I will do the extensive evaluation methods required on the better performing models to compare to my original optimized NaiveBayes classifier that attained 98%. A great deal of packages was required to run all of the models:

|  |
| --- |
| Packages for Sklearn |
| import sys  import pandas  import numpy  from sklearn import preprocessing  from sklearn.svm import LinearSVC  from sklearn.naive\_bayes import MultinomialNB  from sklearn.model\_selection import cross\_val\_predict  from sklearn.naive\_bayes import GaussianNB  from sklearn.metrics import classification\_report  from sklearn.metrics import confusion\_matrix  from sklearn.linear\_model import LogisticRegression  from sklearn.tree import DecisionTreeClassifier  from sklearn.svm import SVC  from nltk.classify.scikitlearn import SklearnClassifier  from sklearn.naive\_bayes import MultinomialNB,BernoulliNB  from sklearn.linear\_model import LogisticRegression,SGDClassifier  from sklearn.svm import SVC, LinearSVC, NuSVC |

Next I brought back the original featuresets for training and testing:

|  |
| --- |
| Training / Testing Code |
| training\_set, testing\_set = featuresets[600:], featuresets[:600] |

Finally, I loaded all of the different classifiers in one big chunk of code. This is the “grand finale” I alluded to earlier! The training/testing ran rather quickly and produced great results.

|  |
| --- |
| Code |
| MNB\_classifier = SklearnClassifier(MultinomialNB())  MNB\_classifier.train(training\_set)  print("MNB\_classifier accuracy percent:", (nltk.classify.accuracy(MNB\_classifier, testing\_set))\*100)  BernoulliNB\_classifier = SklearnClassifier(BernoulliNB())  BernoulliNB\_classifier.train(training\_set)  print("BernoulliNB\_classifier accuracy percent:", (nltk.classify.accuracy(BernoulliNB\_classifier, testing\_set))\*100)  LogisticRegression\_classifier = SklearnClassifier(LogisticRegression())  LogisticRegression\_classifier.train(training\_set)  print("LogisticRegression\_classifier accuracy percent:", (nltk.classify.accuracy(LogisticRegression\_classifier, testing\_set))\*100)  SGDClassifier\_classifier = SklearnClassifier(SGDClassifier())  SGDClassifier\_classifier.train(training\_set)  print("SGDClassifier\_classifier accuracy percent:", (nltk.classify.accuracy(SGDClassifier\_classifier, testing\_set))\*100)  SVC\_classifier = SklearnClassifier(SVC())  SVC\_classifier.train(training\_set)  print("SVC\_classifier accuracy percent:", (nltk.classify.accuracy(SVC\_classifier, testing\_set))\*100)  LinearSVC\_classifier = SklearnClassifier(LinearSVC())  LinearSVC\_classifier.train(training\_set)  print("LinearSVC\_classifier accuracy percent:", (nltk.classify.accuracy(LinearSVC\_classifier, testing\_set))\*100)  NuSVC\_classifier = SklearnClassifier(NuSVC())  NuSVC\_classifier.train(training\_set)  print("NuSVC\_classifier accuracy percent:", (nltk.classify.accuracy(NuSVC\_classifier, testing\_set))\*100)  DecisionTree\_classifier = SklearnClassifier(DecisionTreeClassifier())  DecisionTree\_classifier.train(training\_set)  print("DecisionTree\_classifier accuracy percent:", (nltk.classify.accuracy(DecisionTree\_classifier, testing\_set))\*100) |
| Results |
| A screenshot of a social media post  Description automatically generated |

There are two different Naïve Bayes approaches used in the first set of classifiers notated by MNB (Multinomial Naïve Bayes) and Bernoulli. Both performed very well with Bernoulli edging out MNB just slightly. I researched the difference between the two and found that MNB cares about counts for multiple features that occur whereas Bernoulli NB cares about counts for a single feature that do and do NOT occur. This means that Bernoulli will focus on a keyword and whether or not it occurs. MNB will focus on if multiple keywords are there and doesn’t worry about it if they aren’t. This leads me to believe that there are certain keywords that occur or don’t that give Bernoulli the edge. If we recall, “forward” was highly associated with ham; a keyword like this would be a strong driver for prediction in the BernoulliNB classifier. We can dig into the evaluation measures to see what is going on:

|  |
| --- |
| Measures for Bernoulli |
| A screenshot of a cell phone  Description automatically generated |
| If we recall, this looks very similar to our out-of-the-box model using all documents. Lets take a look to compare: |
| Measures for Starting Model (using all documents) |
| A close up of a logo  Description automatically generated |
| Very cool! This model was able to maintain precision of spam and recall of ham while significantly raising recall of spam.  Not to toot my own horn, but the F1 values for Bernoulli are slightly less than my suped-up base NB model seen below.  Choosing which is better really all depends on what the user values more in predicting emails.  If you can live with a spam email in your inbox every now and then you would pick my model, but if you can’t stand seing them you would go with the Bernoulli approach. |
| Optimized NB bag-of-words |
| A screenshot of a cell phone  Description automatically generated |

**Final Thoughts**

This exploration of email documents through the application of NLTK shows a few interesting insights:

* Using different types of preprocessing can hinder prediction accuracy
* Finding the right document size, vocabulary size, and training/testing size can change how strong or poorly the models operate
* Spam emails tend to either lean heavily positive or somewhat negative
* Computational costs can weigh in on practicality of a model

I will say that it was intimidating being in a class with other students with heavy backgrounds in NLP and machine learning! But, once I found my footing, I really enjoyed the last couple of assignments. The class has been challenging, but also very rewarding. NLP applications can be difficult due to the complexity of language, but I find it one of the more practical aspects of data science and machine learning (especially when considering spam emails). I find it interesting that some of the strong features for predicting ham emails are now techniques used by spammers to break through my current email system and into my inbox. There was a time that most spam emails I received started with “FWD:”. I didn’t think much of it at the time, but I now see that this was an attempt to trick the old models. Language and how we use language are always changing; catching spam email is a good example of the need to constantly update our predictive models in order to catch new “spam techniques”. I tried to create this report in a way that wasn’t overly lengthy, although it still became longer than 20 pages. There was definitely a lot more I could have included, and I will be sure to attach my Jupyter Notebook so to show the other NLP tasks that were executed.