IST-664 - Final Project

Detection on SPAM in email

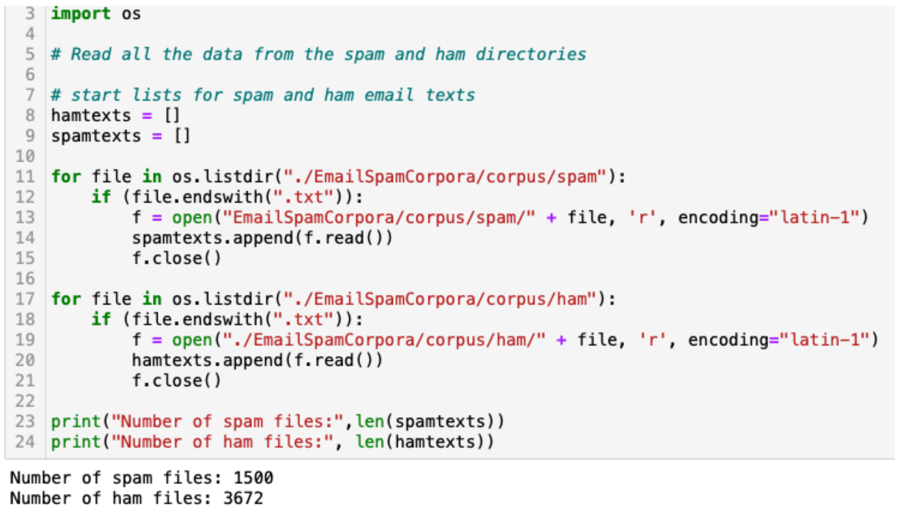
Student: Sharat Sripada (vssripad@syr.edu)

1. Introduction

As part of the final project for IST-664, we will work on building on Spam email filter. The goal is to be able to detect spam emails based on a dataset produced from Enron's public email corpus.

The data is largely structured under two sub-directories namely *ham* and *spam* comprising non-spam emails and spam emails respectively.

First, we download the dataset to a local working directory and using the python os module list all the relevant .txt extension files and read data into lists labelled hamtexts and spamtexts:



```python
import os

# Read all the data from the spam and ham directories

# start lists for spam and ham email texts
hamtexts = []
spamtexts = []

for file in os.listdir("./EmailSpamCorpora/corpus/spam"):
    if (file.endswith(".txt")):
        f = open("EmailSpamCorpora/corpus/spam/" + file, 'r', encoding="latin-1")
        spamtexts.append(f.read())
        f.close()

for file in os.listdir("./EmailSpamCorpora/corpus/ham"):
    if (file.endswith(".txt")):
        f = open("./EmailSpamCorpora/corpus/ham/" + file, 'r', encoding="latin-1")
        hamtexts.append(f.read())
        f.close()

print("Number of spam files:",len(spamtexts))
print("Number of ham files:", len(hamtexts))
```

```
Number of spam files: 1500
Number of ham files: 3672
```

As seen above, *os.listdir()* helps list all files within the directories and using a for loop we iteratively open each file, read its contents and populate a list.

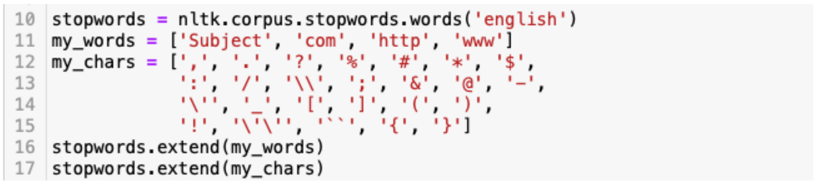
2. Preparing the data

During the first exploration of data, several redundant words and symbols were found in the text. Since, we would be building unigram and bi-gram feature-sets it was important to filter and keep only relevant data to make the modelling faster and efficient.

The first step was to consume stop-words from the *nltk.corpus.stopwords.words()* package and extend it to the following:

- remove common email text words like Subject, com, http, www
- remove characters like :.,$#?*{}()[]\ etc.

Commonly, this is done by populating words and characters to lists and later extend the stopwords as seen below:

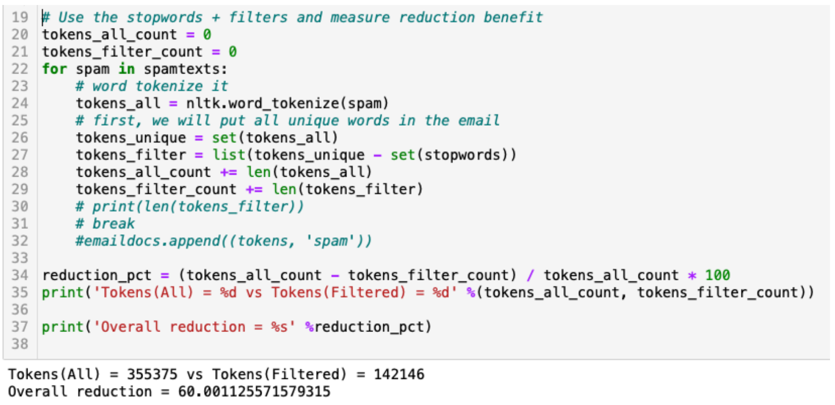


```python
stopwords = nltk.corpus.stopwords.words('english')
my_words = ['Subject', 'com', 'http', 'www']
my_chars = [',', '.', '?', '%', '#', '*', '$',
            ':', '/', '\\', ';', '&', '@', '-',
            '\'', '_', '[', ']', '(', ')',
            '!', '\'\'', '``', '{', '}']
stopwords.extend(my_words)
stopwords.extend(my_chars)
```

Using the python set functionality, we will filter the corpus and remove all stopwords from it. Following is an example demonstrating this ability:

```
>>> first_eight = [1,2,3,4,5,6,7,8]
>>> filter_last_four = [5,6,7,8]
>>> first_four = list(set(first_eight) - set(filter_last_four))
>>> first_four
[1, 2, 3, 4]
```

Essentially, the mathematical operation *minus* helps filter out the contents in the latter set from contents in the former set. Extending this to filter out the stopwords, we iteratively walk each email, word tokenize it and use the set operation to filter stopwords:

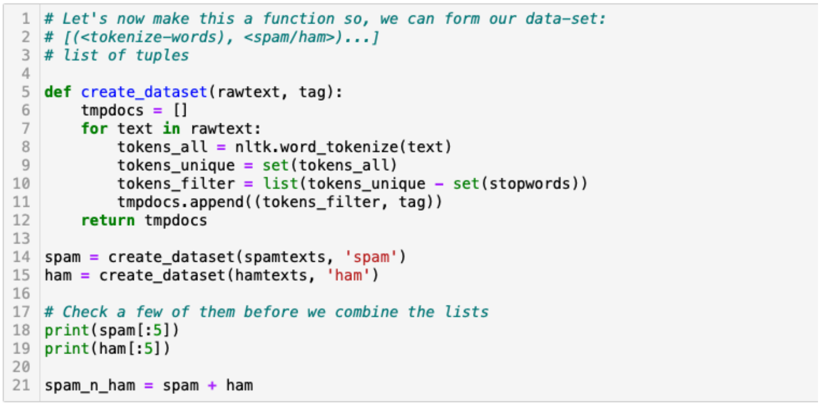


```python
# Use the stopwords + filters and measure reduction benefit
tokens_all_count = 0
tokens_filter_count = 0
for spam in spamtexts:
    # word tokenize it
    tokens_all = nltk.word_tokenize(spam)
    # first, we will put all unique words in the email
    tokens_unique = set(tokens_all)
    tokens_filter = list(tokens_unique - set(stopwords))
    tokens_all_count += len(tokens_all)
    tokens_filter_count += len(tokens_filter)
    # print(len(tokens_filter))
    # break
    #emaildocs.append((tokens, 'spam'))

reduction_pct = (tokens_all_count - tokens_filter_count) / tokens_all_count * 100
print('Tokens(All) = %d vs Tokens(Filtered) = %d' %(tokens_all_count, tokens_filter_count))

print('Overall reduction = %s' %reduction_pct)

```

```
Tokens(All) = 355375 vs Tokens(Filtered) = 142146
Overall reduction = 60.001125571579315
```

We also measured a word reduction count of 60% when taking this filtering approach.

Convinced, this would be the right approach a specific function was written so it could be applied to both the spam and ham lists:



```python
# Let's now make this a function so, we can form our data-set:
# [(<tokenize-words), <spam/ham>)...]
# list of tuples

def create_dataset(rawtext, tag):
    tmpdocs = []
    for text in rawtext:
        tokens_all = nltk.word_tokenize(text)
        tokens_unique = set(tokens_all)
        tokens_filter = list(tokens_unique - set(stopwords))
        tmpdocs.append((tokens_filter, tag))
    return tmpdocs

spam = create_dataset(spamtexts, 'spam')
ham = create_dataset(hamtexts, 'ham')

# Check a few of them before we combine the lists
print(spam[:5])
print(ham[:5])

spam_n_ham = spam + ham
```

Calling the *create_dataset()* function individually on the spam and ham lists, we combine the lists and aggregate a new list called *spam_n_ham* in row-21. The data in this aggregated list is such that each index or node is a tuple comprising words in an email and a tag indicating if it was spam or ham. This categorization will help us with the classification task a later time.

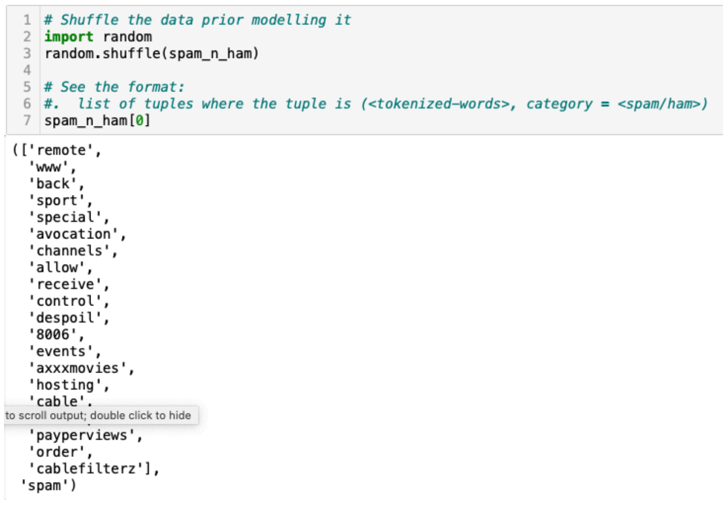
Output of the combined list showing an entry for spam and ham:

```
[(['time', 'road', 'naturalgolden', 'ress', 'companion', 'find', 'terrific', 'youll',
'www', 'turn', 'bio', 'love', 'cam', 'developed', 'meganbang', 'site', 'movie', 'line'
, 'try', 'help', 'amazed', 'date', 'encomia', 'form', 'matter', 'catatonia', 'fashione
d', 'intervenor', 'catfish', 'every', 'see', 'acc', 'brandywine', 'good', 'preemptive'
, 'babe', 'plz', 'war', 'shoehorn', 'word', 'sense', 'browser', 'ole', 'skeleton', 're
tract', 'satisfaction', 'electrocardiograph', 'counterattack', 'biz', 'quick', 'scaup'
, 'evening', 'brand', 'lookup', 'created', 'copy', 'step', 'byrne', 'friendship', 'may
', 'looking', 'pietism', 'anyone', 'pa', 'monster', 'ste', 'come', 'honeycomb', 'new',
'bld', 'add', 'aitken'], 'spam'),
```

```
[(['103', 'month', '3', '11', '29165', 'meters', '6', '1997', '078', 'k', '30100', 'he
lp', 'thanks', '4', '12', '1', 'referenced', '1567', 'issues', 'placed', 'meter', '98'
, '9638', 'note', '089', '6736', 'ends', 'please', 'also', '97', 'deal', '101', '9497'
, 'jackie', 'need', 'cpr', '5', 'activity', 'ua', 'information'], 'ham'),
```

NOTE:
Data that may seem noise – like numbers, alpha-numerics and other non-dictionary words were not filtered fr om the dataset using regular expressions or other string processing methods. Retaining such data would likely help the classification problem.

Before we start to consume the dataset, we un-serialize the serialized data (data is serial since we formed spa m_n_ham using spam + ham) using the random shuffle python function as seen in row-3 below. This would als o help train the model on fairly diverse data:



```
1 # Shuffle the data prior modelling it
2 import random
3 random.shuffle(spam_n_ham)
4
5 # See the format:
6 #.  list of tuples where the tuple is (<tokenized-words>, category = <spam/ham>)
7 spam_n_ham[0]
```

```
(['remote',
  'www',
  'back',
  'sport',
  'special',
  'avocation',
  'channels',
  'allow',
  'receive',
  'control',
  'despoil',
  '8006',
  'events',
  'axxxmovies',
  'hosting',
  'cable',
to scroll output; double click to hide
  'payperviews',
  'order',
  'cablefilterz'],
 'spam')
```

The output of *spam_n_ham* on row-7 above also shows the words different from the first index on the spam list earlier.

3. Classification Task

With the data filtering and aggregation sufficiently complete we can now build a model and make predictions.

For this exercise, we will use the Naïve Bayes classification model. Historically, although other algorithms like Support vector machines and Boosting are known to be top-performing, we will use Naïve Bayes as it is vastly popular in commercial and open-source category spam filters.

NLTK readily offers the Naïve Bayes classification method, and we will see in the following sections how it can be used to solve the problem and study its performance.

To segregate the data for train and test, we will use the K-fold (k=5) cross-validation method rather than a simple 80-20 split.

Why k-fold cross-validation method?

Cross-validation is a powerful tool that helps us better use our data (spread) and therefore provides useful information about the performance of the algorithms we may choose.

Here's a good visualization of how cross-validation works:

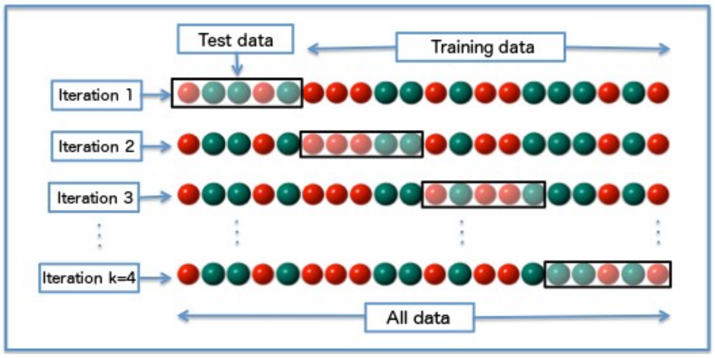
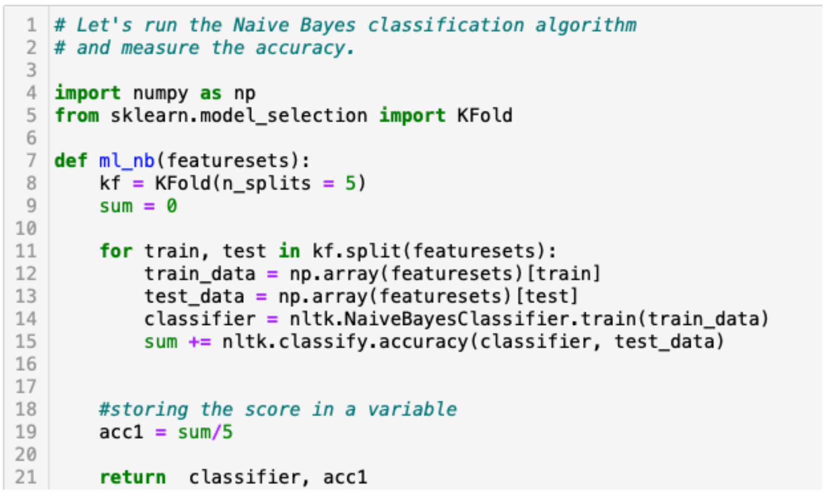


Diagram of k-fold cross-validation with k=4.

Data is split more than once (controlled by folds or commonly known as variable k), than in the classic 1-split 80-20, 90-10 or 70-30 representing train and validation/test data.

Each split or fold is a good representation of the whole data and therefore helps train or develop comprehensive and robust models. Accuracy or performance is derived by averaging the results over several iterations as shown below:



```
 1 # Let's run the Naive Bayes classification algorithm
 2 # and measure the accuracy.
 3 
 4 import numpy as np
 5 from sklearn.model_selection import KFold
 6 
 7 def ml_nb(featuresets):
 8     kf = KFold(n_splits = 5)
 9     sum = 0
10 
11     for train, test in kf.split(featuresets):
12         train_data = np.array(featuresets)[train]
13         test_data = np.array(featuresets)[test]
14         classifier = nltk.NaiveBayesClassifier.train(train_data)
15         sum += nltk.classify.accuracy(classifier, test_data)
16 
17 
18     #storing the score in a variable
19     acc1 = sum/5
20 
21     return  classifier, acc1
```

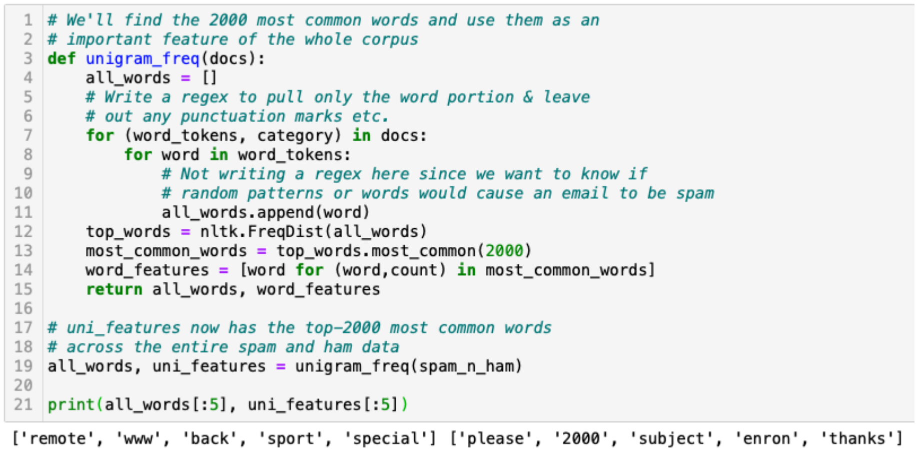
See rows 14-19 above on how accuracy is summed over each k-fold iteration and then averaged to provide an overall accuracy.

Building models

While Naïve Bayes would largely help us with the classification, we take two fundamental NLP approaches:

1. Unigram feature-set

   In this approach, we derive a feature set comprising single words ordered by frequency and pick the top 2000 words.

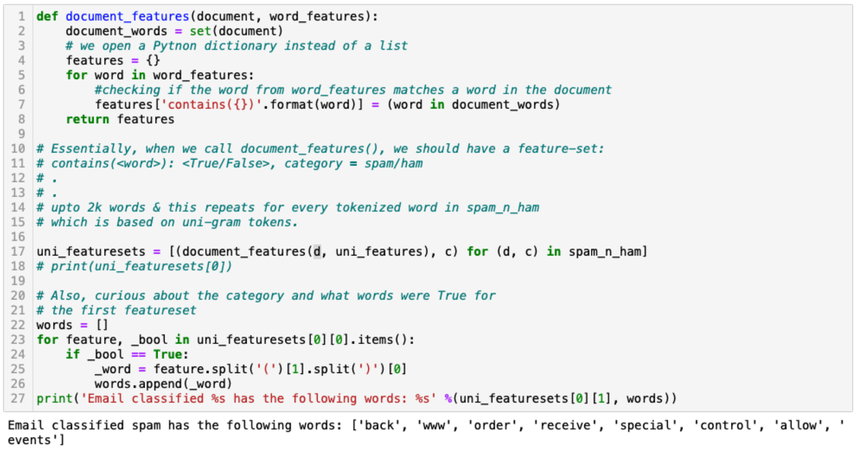


```
 1 # We'll find the 2000 most common words and use them as an
 2 # important feature of the whole corpus
 3 def unigram_freq(docs):
 4     all_words = []
 5     # Write a regex to pull only the word portion & leave
 6     # out any punctuation marks etc.
 7     for (word_tokens, category) in docs:
 8         for word in word_tokens:
 9             # Not writing a regex here since we want to know if
10             # random patterns or words would cause an email to be spam
11             all_words.append(word)
12     top_words = nltk.FreqDist(all_words)
13     most_common_words = top_words.most_common(2000)
14     word_features = [word for (word,count) in most_common_words]
15     return all_words, word_features
16 
17 # uni_features now has the top-2000 most common words
18 # across the entire spam and ham data
19 all_words, uni_features = unigram_freq(spam_n_ham)
20 
21 print(all_words[:5], uni_features[:5])
```

```
['remote', 'www', 'back', 'sport', 'special'] ['please', '2000', 'subject', 'enron', 'thanks']
```

   The code shows use of *nltk.FreqDist()* to sort the words by frequency and we return the top 2000 as seen through rows 13-15.

Next, we form a unigram feature set as shown below:



```
 1 def document_features(document, word_features):
 2     document_words = set(document)
 3     # we open a Pytnon dictionary instead of a list
 4     features = {}
 5     for word in word_features:
 6         #checking if the word from word_features matches a word in the document
 7         features['contains({})'.format(word)] = (word in document_words)
 8     return features
 9 
10 # Essentially, when we call document_features(), we should have a feature-set:
11 # contains(<word>): <True/False>, category = spam/ham
12 # .
13 # .
14 # upto 2k words & this repeats for every tokenized word in spam_n_ham
15 # which is based on uni-gram tokens.
16 
17 uni_featuresets = [(document_features(d, uni_features), c) for (d, c) in spam_n_ham]
18 # print(uni_featuresets[0])
19 
20 # Also, curious about the category and what words were True for
21 # the first featureset
22 words = []
23 for feature, _bool in uni_featuresets[0][0].items():
24     if _bool == True:
25         _word = feature.split('(')[1].split(')')[0]
26         words.append(_word)
27 print('Email classified %s has the following words: %s' %(uni_featuresets[0][1], words))
```

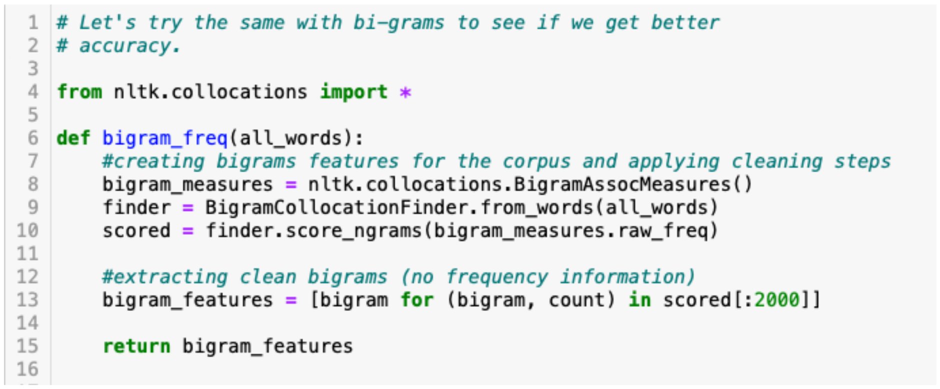
```
Email classified spam has the following words: ['back', 'www', 'order', 'receive', 'special', 'control', 'allow', '
events']
```

The function *document_features()* takes two inputs namely the word corpora (which essentially is word tokens from a sentence) and the set of features we identified in the earlier step. Using this we iterate each word and record 'contains(<word>): <Boolean>' result in a dictionary and return it back. *uni_featuresets* in row17 is thus a list comprehension comprising at each node or index a dictionary of words with a Boolean result of True/False for a word found or not found respectively. Importantly, the category of spam or ham is still retained in the tuple.

A sample output for index 0 of *uni_featuresets* is shown:
`Email classified` **`spam`** `has the following words:` **`['back', 'www', 'order', 'receive', 'special', 'control', 'allow', 'events']`**

2. Bi-gram feature-set

In this approach, we will group two-words at a time using the *NLTK collocations.BigramAssocMeasures()* and score frequencies using the *score_ngrams()* functionality. This is shown in the code below in rows 8-10:



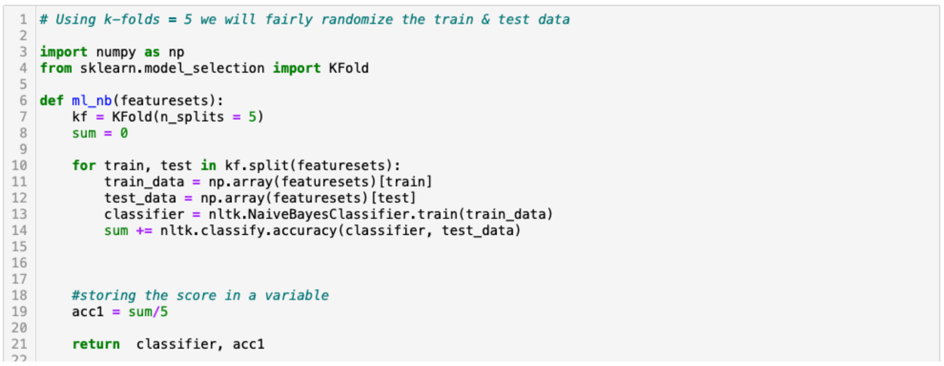
```
 1 # Let's try the same with bi-grams to see if we get better
 2 # accuracy.
 3 
 4 from nltk.collocations import *
 5 
 6 def bigram_freq(all_words):
 7     #creating bigrams features for the corpus and applying cleaning steps
 8     bigram_measures = nltk.collocations.BigramAssocMeasures()
 9     finder = BigramCollocationFinder.from_words(all_words)
10     scored = finder.score_ngrams(bigram_measures.raw_freq)
11 
12     #extracting clean bigrams (no frequency information)
13     bigram_features = [bigram for (bigram, count) in scored[:2000]]
14 
15     return bigram_features
16 
```

Correspondingly, the *bi_document_features()* function would walk the word pairs (collocations) in each email, and attempt to find a match with the bi-gram features. Like the unigram document features functionality, the function would return a dictionary comprising 'contains(<word-pairs>): <Boolean>' which would be then placed alongside a category spam or ham at each node or index in the list.

Below is the category and word-pairs at index 0 for the list comprehension associated with bi-grams:

```
Email classified spam has the following words: ["contains(('receive', 'control'))", "c
ontains(('allow', 'receive'))"]
```

The classification and prediction using Naïve Bayes itself was written in a general manner so it return a classifier and accuracy measure given a feature-set, and is represented by the function *ml_nb():*



```
 1 # Using k-folds = 5 we will fairly randomize the train & test data
 2 
 3 import numpy as np
 4 from sklearn.model_selection import KFold
 5 
 6 def ml_nb(featuresets):
 7     kf = KFold(n_splits = 5)
 8     sum = 0
 9 
10     for train, test in kf.split(featuresets):
11         train_data = np.array(featuresets)[train]
12         test_data = np.array(featuresets)[test]
13         classifier = nltk.NaiveBayesClassifier.train(train_data)
14         sum += nltk.classify.accuracy(classifier, test_data)
15 
16 
17 
18     #storing the score in a variable
19     acc1 = sum/5
20 
21     return  classifier, acc1
```

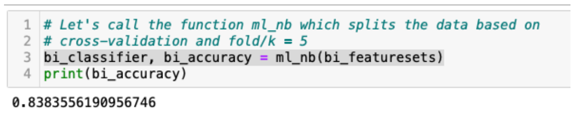
Within this function, the data is split as train and test (using *kfold* method) before running the NLTK NaiveBayesClassifier() on train-data (seen on row-13). The prediction or classification and accuracy is recorded and averaged over the k-folds.

Results

Following are Unigram vs bi-gram results tabulated:

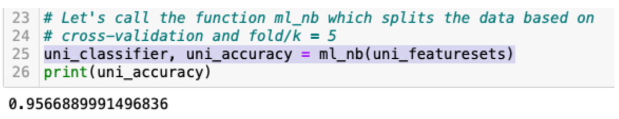
Naïve Bayes classifier k-fold = 5	Spam and Ham Data
Unigram frequency	96.0%
Bi-gram frequency	83.8%

The outputs showing the function call and results are captured below:



```
# Let's call the function ml_nb which splits the data based on
# cross-validation and fold/k = 5
bi_classifier, bi_accuracy = ml_nb(bi_featuresets)
print(bi_accuracy)
```

```
0.8383556190956746
```



```
# Let's call the function ml_nb which splits the data based on
# cross-validation and fold/k = 5
uni_classifier, uni_accuracy = ml_nb(uni_featuresets)
print(uni_accuracy)
```

```
0.9566889991496836
```

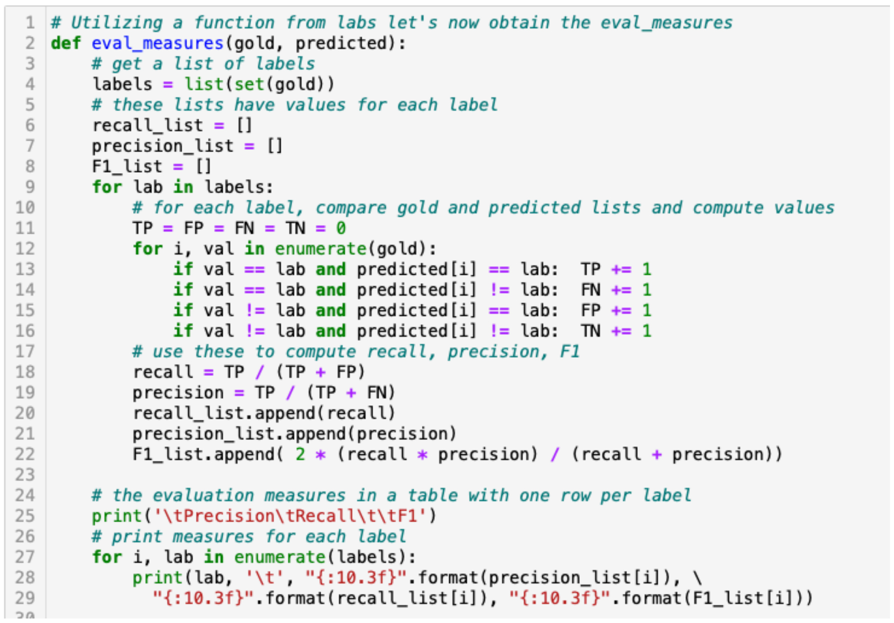
Evaluation Measures

In this section we will derive evaluation measures to further understand the behavior and performance of the model. Following are some of KPIs:

- RECALL
  Definition: The percentage of actual yes answers that are right
  Or
  Recall = TP / (TP + FP)
- PRECISION
  Definition: The percentage of predicted yes answers that are right
  Or
  Precision = TP / (TP + FN)
- F-measure
  Recall and precision combined into an average or harmonic mean
  Or
  F-measure = 2 * (Recall * Precision) / (Recall + Precision)
- ACCURACY
  Percentage of correct Yes and No out all the text examples
  Or
  Accuracy = TP + TN / (TP + FP + FN + TN)

Where TP = True Positive, TN = True Negative, FP = False Positive, FN = False Negative

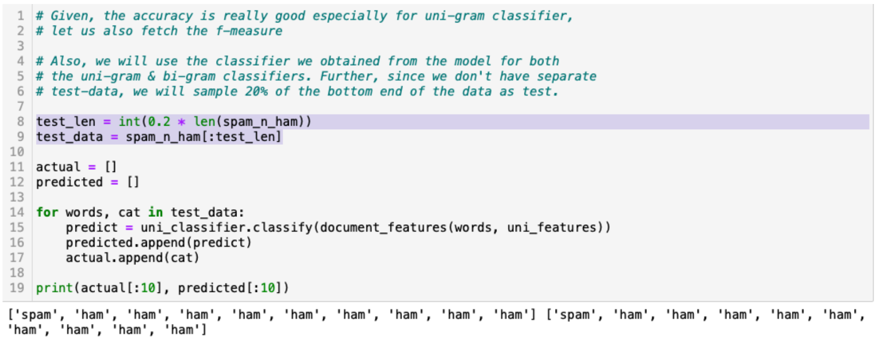
These definitions are formulated into Python code in rows 18-22 and encompassed with the *eval_measures()* function as shown below:



```
 1 # Utilizing a function from labs let's now obtain the eval_measures
 2 def eval_measures(gold, predicted):
 3     # get a list of labels
 4     labels = list(set(gold))
 5     # these lists have values for each label
 6     recall_list = []
 7     precision_list = []
 8     F1_list = []
 9     for lab in labels:
10         # for each label, compare gold and predicted lists and compute values
11         TP = FP = FN = TN = 0
12         for i, val in enumerate(gold):
13             if val == lab and predicted[i] == lab:  TP += 1
14             if val == lab and predicted[i] != lab:  FN += 1
15             if val != lab and predicted[i] == lab:  FP += 1
16             if val != lab and predicted[i] != lab:  TN += 1
17         # use these to compute recall, precision, F1
18         recall = TP / (TP + FP)
19         precision = TP / (TP + FN)
20         recall_list.append(recall)
21         precision_list.append(precision)
22         F1_list.append( 2 * (recall * precision) / (recall + precision))
23 
24     # the evaluation measures in a table with one row per label
25     print('\tPrecision\tRecall\t\tF1')
26     # print measures for each label
27     for i, lab in enumerate(labels):
28         print(lab, '\t', "{:10.3f}".format(precision_list[i]), \
29           "{:10.3f}".format(recall_list[i]), "{:10.3f}".format(F1_list[i]))
```

The function takes two lists viz. actual and predicted classification results. Then compares them and captures KPIs in a table-like format. Rows 13-16 mathematically evaluate TP, FN, FP, and TN and subsequently calculate the Recall, Precision and F-measure.

Since, we do not have test data or validation data we split the bottom 20% for test data to run some measures:

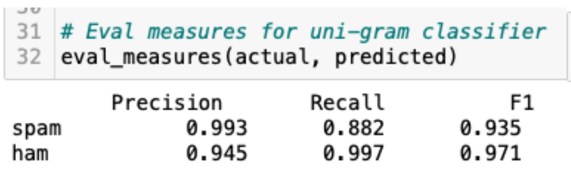


```
# Given, the accuracy is really good especially for uni-gram classifier,
# let us also fetch the f-measure

# Also, we will use the classifier we obtained from the model for both
# the uni-gram & bi-gram classifiers. Further, since we don't have separate
# test-data, we will sample 20% of the bottom end of the data as test.

test_len = int(0.2 * len(spam_n_ham))
test_data = spam_n_ham[:test_len]

actual = []
predicted = []

for words, cat in test_data:
    predict = uni_classifier.classify(document_features(words, uni_features))
    predicted.append(predict)
    actual.append(cat)

print(actual[:10], predicted[:10])
```

```
['spam', 'ham', 'ham', 'ham', 'ham', 'ham', 'ham', 'ham', 'ham', 'ham'] ['spam', 'ham', 'ham', 'ham', 'ham', 'ham',
'ham', 'ham', 'ham', 'ham']
```

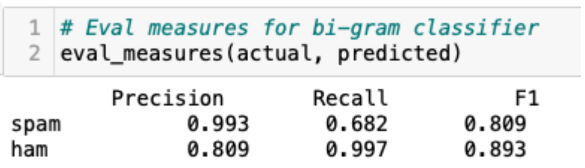
Using the classifier from the unigram and bi-gram classification task earlier we will put together two lists – actu al and predicted. We will then pass the lists to the *eval_measures()* function to get a summary of the results.

Results



```
# Eval measures for uni-gram classifier
eval_measures(actual, predicted)
```

```
          Precision    Recall       F1
spam          0.993     0.882    0.935
ham           0.945     0.997    0.971
```



```
# Eval measures for bi-gram classifier
eval_measures(actual, predicted)
```

```
          Precision    Recall       F1
spam          0.993     0.682    0.809
ham           0.809     0.997    0.893
```

Interpreting evaluation measures

- Precision for bi-gram classification shows poor-performance for ham classification emails.
    - Precision for spam emails is comparable at 99.3%.
- Recall for bi-gram classification also shows poor-performance for spam (that is, the percent of predicti on spam vs actual is only at 68%).
- F-measure is showing an overall improved performance with unigrams.

4. Conclusion

In this case-study, spam and ham emails were classified using a Naïve Bayes classification algorithm. Feature-sets were mostly based on unigrams and bi-grams derived from the email corpus.

The model was trained and tested across datasets derived from cross-validation (k-folds/k=5) and yielded moderately poorer performance or accuracy for the bi-gram feature-set at 83.8% (compared to 96% with unigrams).

This is strangely an outlier in comparison with some of the experiments with books from Glutenberg corpus in other home-work assignments where we always recorded bi-grams to show improved performance in comparison with unigrams. Further analysis with SPAM email classifier using evaluation measurement techniques also showed particularly poor predictions to classify ham (non-spam) emails.

To improve the accuracy of the model, whitepaper '*Spam Filtering with Naïve Bayes – Which Naïve Bayes?*' seems to provide some answers. While the paper broadly acknowledges the popularity of the Naïve Bayes algorithms for SPAM classification it brings to light some possible nuances with the algorithm – and choices between the following:

- Multi-variate Bernoulli NB
- Multinominal NB, TF attributes
- Multinominal NB, Boolean attributes
- Multi-variate Gauss NB
- Flexible Bayes

Per the results in the whitepaper the Multinominal NB, Boolean spam and ham recall is at >97%. As part of some of the future work I intend to explore some of these variants.