Introduction

This dataset is developed for detecting spam emails from Enron public email corpus. The spam emails are labelled as spam and non-spam emails are labelled as ham. For our analysis, we have decided to use 3000 files.

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

The task at hand is to develop features which would detect spam mails using the dataset provided.

Text Processing

The program shell initially used reads email data for the spam classification problem. The input to the program is the path to the Email directory "corpus" and a limit number. The program reads the first limit number of ham emails and the first limit number of spam. It creates an "emaildocs" variable with a list of emails consisting of a pair with the list of tokenized words from the email and the label either spam or ham.

- As part of our first work tokenizing, we note that the word tokenization produces tokens that have special characters in them. We removed all the tokens that have only special characters. We then applied filters to remove non-alphabetical characters from these tokens.

```
# possibly filter tokens
  def alpha_filter(w): # function that takes a word and returns true if it consists only of non-alphabetic characters
      pattern = re.compile('^[^a-z]+$')
      if (pattern.match(w)):
          return True
      else:
          return False
```

- In the next step, we have created a list of stopwords and combined it with the nltk in-built stopwords list. This stopwords list created was based on inspecting the most frequent words that are irrelevant to our analysis. For example, we have added ‘s’, ‘b’, ‘e’, etc and words like ‘subject’ which appear often in an email but add no value.
- We created three word sets, one containing an all words list, second containing alpha-numeric characters, and third word list with stopwords in it.

```
# Creating a stopwords list to filter stopwords from email documents data
nltkstopwords = nltk.corpus.stopwords.words('english')
morestopwords =  ["Subject","subject","com","http",'could','would','might','must','need','sha','wo',"ll","t","m","re","ve","th","pm","e",'y',"s","e","d"]
stopwords = nltkstopwords + morestopwords


# Creating word features

all_words_list = [(word,tag) for (email,tag) in emaildocs for word in email]
words_alpha = [(word,tag) for (word,tag) in all_words_list if not alpha_filter(word)]
words_stopped = [(word,tag) for (word,tag) in words_alpha if not word in stopwords]
```

- We created word features with **2500** most common words as we do not need to consider all the words for our analysis. Feature Sets were created for both lists with processing and without processing so we could compare the classification results between them.
- Next, we define a feature definition function for the unigram features separated with filtering and without filtering.
- Below we are showing screenshots of **30** most frequent words in both the feature sets.

```
FreqDist of 30 most common without filters
('-', 59589)
('.', 37289)
('/', 30180)
(',', 26172)
(':', 18960)
('the', 17217)
('to', 13325)
('ect', 10658)
('and', 9034)
('@', 8101)
('of', 7502)
('a', 7031)
('for', 6841)
('you', 5752)
('?', 5650)
('in', 5613)
('hou', 5515)
('is', 5059)
('this', 5006)
('on', 4719)
('enron', 4053)
('i', 4035)
(')', 3724)
("'", 3675)
('2000', 3619)
('(', 3551)
('Subject', 3500)
('be', 3440)
('that', 3243)
('=', 3209)
```

```
FreqDist of 30 most common with filters
('ect', 10658)
('hou', 5515)
('enron', 4053)
('please', 2113)
('meter', 1671)
('gas', 1633)
('cc', 1478)
('deal', 1414)
('hpl', 1329)
('corp', 1305)
('thanks', 1093)
('new', 1049)
('daren', 1029)
('know', 994)
('forwarded', 931)
('company', 925)
('get', 905)
('information', 866)
('may', 843)
('price', 798)
('j', 785)
('mmbtu', 721)
('let', 715)
('time', 714)
('l', 711)
('us', 708)
('one', 704)
('see', 686)
('p', 677)
('www', 657)
```

Naïve Bayes Classifier and Cross Validation Evaluation Method

- Testing data with Naïve Bayes classifier on random samples by dividing it into a train and test set will often give us skewed results. The remedy for this is to use different chunks of data as a test set to repeatedly train. We do this by using a cross validation method.
- In this method we choose several folds like 5 or 10 and randomly partition the data into **k** subsets of equal size. Then we train the classifier **k** times.
- NLTK does not have a built-in function for cross validation. Here we have programmed the process in a function that takes in several folds and feature sets and iterates over training and testing a classifier.
- The function reports accuracy for each fold and for the overall average.

```python
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)
```

- Below we show the most important features data used by the classifiers for classification with the ratio scores. These features must be excluded from the stopwords list if they were added.

```
Most Informative Features
             V_forwarded = True              ham : spam   =    217.4 : 1.0
                   V_hou = True              ham : spam   =    201.2 : 1.0
                   V_ect = True              ham : spam   =    125.7 : 1.0
            V_nomination = True              ham : spam   =     71.7 : 1.0
          V_professional = True             spam : ham    =     53.7 : 1.0
                  V_2005 = True             spam : ham    =     49.2 : 1.0
                 V_steve = True              ham : spam   =     48.5 : 1.0
                   V_ami = True              ham : spam   =     46.2 : 1.0
                 V_vance = True              ham : spam   =     46.0 : 1.0
                  V_lisa = True              ham : spam   =     43.1 : 1.0
                V_health = True             spam : ham    =     42.9 : 1.0
                    V_cc = True              ham : spam   =     42.0 : 1.0
                V_farmer = True              ham : spam   =     41.3 : 1.0
                   V_713 = True              ham : spam   =     40.6 : 1.0
                 V_susan = True              ham : spam   =     39.6 : 1.0
                    V_pg = True              ham : spam   =     37.7 : 1.0
               V_generic = True             spam : ham    =     37.5 : 1.0
                 V_brand = True             spam : ham    =     35.6 : 1.0
                    V_pm = True              ham : spam   =     35.6 : 1.0
                   V_tel = True             spam : ham    =     34.7 : 1.0
                V_jackie = True              ham : spam   =     34.2 : 1.0
                 V_lloyd = True              ham : spam   =     34.2 : 1.0
              V_pipeline = True              ham : spam   =     34.2 : 1.0
                V_shares = True             spam : ham    =     33.8 : 1.0
              V_thousand = True             spam : ham    =     33.8 : 1.0
                V_meters = True              ham : spam   =     33.7 : 1.0
                V_differ = True             spam : ham    =     32.9 : 1.0
              V_featured = True             spam : ham    =     32.9 : 1.0
             V_investing = True             spam : ham    =     32.9 : 1.0
              V_property = True             spam : ham    =     32.9 : 1.0
```

- Cross-validation output of features with all words included

```
Each fold size: 600
0 0.9716666666666667
         Precision       Recall         F1
spam           1.000      0.946       0.972
ham            0.944      1.000       0.971
1 0.9716666666666667
         Precision       Recall         F1
spam           0.997      0.947       0.971
ham            0.949      0.997       0.972
2 0.975
         Precision       Recall         F1
spam           0.997      0.957       0.976
ham            0.952      0.996       0.974
3 0.9616666666666667
         Precision       Recall         F1
spam           1.000      0.927       0.962
ham            0.925      1.000       0.961
4 0.97
         Precision       Recall         F1
spam           1.000      0.945       0.972
ham            0.938      1.000       0.968
mean accuracy 0.97
```

Here we observe that the mean accuracy from the above cross validation result is **97%** with features like 'forwarded', 'hou', and 'ect' as the most informative features.

Experiments

We have conducted several experiments with different feature sets to compare the classification results between them.

Filter by removing special character words and stop words.

- Here we have used the classifier on the feature set after removing the above-mentioned stop words and all special character words. The cross-validation output is shown below.

```
Each fold size: 600
0 0.9583333333333334
          Precision      Recall        F1
spam          0.953       0.963       0.958
ham           0.963       0.954       0.959
1 0.9383333333333334
          Precision      Recall        F1
spam          0.927       0.944       0.935
ham           0.949       0.934       0.941
2 0.925
          Precision      Recall        F1
spam          0.890       0.962       0.925
ham           0.962       0.891       0.925
3 0.9416666666666667
          Precision      Recall        F1
spam          0.942       0.939       0.940
ham           0.942       0.945       0.943
4 0.9383333333333334
          Precision      Recall        F1
spam          0.926       0.953       0.939
ham           0.952       0.923       0.937
mean accuracy 0.9403333333333335
```

The mean accuracy achieved from the cross-validation results is 94.03% which has reduced from 97% after adding the filter for stop words and character words.

Creating bigram features

- We have created some bigram features. We decided to filter out special characters which were very frequent in the bigrams to use highly frequent bigrams and also filter them by frequency. Another frequently used alternative is to just use frequency, which is the bigram measure raw_freq. But there is another bigram association measure that is more often used to filter bigrams for classification features. This is the chi-squared measure, which is another measure of information gain, but which does its own frequency filtering.
- We created a feature extraction function that has all the word features as before but also has bigram features.

```
def bigram_document_features(document, word_features, bigram_features):
    document_words = set(document)
    document_bigrams = nltk.bigrams(document)
    features = {}
    for word in word_features:
        features['V_{}'.format(word)] = (word in document_words)
    for bigram in bigram_features:
        features['B_{}_{}'.format(bigram[0], bigram[1])] = (bigram in document_bigrams)
    return features

#Now we create feature sets as before, but using this feature extraction function.

bigram_featuresets = [(bigram_document_features(d,words_features,bigram_features), c) for (d,c) in emaildocs]
```

- The cross-validation results after adding the bigram features have been added below.

```
Each fold size:  600
0 0.9716666666666667
        Precision       Recall         F1
spam          1.000      0.946      0.972
ham           0.944      1.000      0.971
1 0.9716666666666667
        Precision       Recall         F1
spam          0.997      0.947      0.971
ham           0.949      0.997      0.972
2 0.975
        Precision       Recall         F1
spam          0.997      0.957      0.976
ham           0.952      0.996      0.974
3 0.9616666666666667
        Precision       Recall         F1
spam          1.000      0.927      0.962
ham           0.925      1.000      0.961
4 0.97
        Precision       Recall         F1
spam          1.000      0.945      0.972
ham           0.938      1.000      0.968
mean accuracy 0.97
```

Mean accuracy from cross-validation reported here is **97%** after adding the bigram features.

Representing Negation

We added a feature to remove negation words. This includes words like 'no', 'not', 'never', etc. The presence of a negative word does not necessarily mean there is a negative sentiment. That's why we did this because negation words can be misleading and ruin the analysis. So, we run the classification again and get the accuracy result using cross-validation.

Description of the feature set used is attached below.

```
negationwords = ['nowhere', 'nothing', 'noone','no', 'not', 'never', 'none','nobody',
                 'rather', 'hardly', 'scarcely', 'rarely', 'seldom', 'neither', 'nor']

def Neg_features(document, word_features, negationwords):
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = False
        features['contains(NOT{})'.format(word)] = False
    # go through document words in order
    for ctr in range(0, len(document)):
        word = document[ctr]
        if ((ctr + 1) < len(document)) and ((word in negationwords) or (word.endswith("n't"))):
            ctr += 1
            features['contains(NOT{})'.format(document[ctr])] = (document[ctr] in word_features)
        else:
            features['contains({})'.format(word)] = (word in word_features)
    return features

Neg_featuresets = [(Neg_features(d, fil_words_features, negationwords), c) for (d, c) in emaildocs]
```

- The cross-validation output after considering the negation word features is added below

```
Each fold size: 600
0 0.9783333333333334
        Precision       Recall          F1
spam          0.983      0.974       0.978
ham           0.973      0.983       0.978
1 0.965
        Precision       Recall          F1
spam          0.969      0.959       0.964
ham           0.961      0.971       0.966
2 0.9533333333333334
        Precision       Recall          F1
spam          0.945      0.964       0.954
ham           0.962      0.943       0.952
3 0.9633333333333334
        Precision       Recall          F1
spam          0.973      0.953       0.963
ham           0.955      0.974       0.964
4 0.9683333333333334
        Precision       Recall          F1
spam          0.974      0.965       0.970
ham           0.962      0.972       0.967
mean accuracy 0.9656666666666667
```

Mean accuracy from cross-validation reported here is **96.5%** after adding the negation word features.

Using the LIWC sentiment lexicon

We used a LIWC (Linguistic Inquiry and Word Count) text analysis program. It calculates the degree to which various categories of words are used in a text, and can process texts ranging from emails to speeches, poems and transcribed natural language in either plain text or Word formats. In this experiment we want to see if the emotion words of the document help us better predict if an email is spam or not.

```
os.chdir("C:/Users/HP/Documents/SU - MSBA/MSBA Lectures/Spring 21/IST 664 NLP/Project/FinalProjectData/kagglemoviereviews/SentimentLexicons")
def read_words():

    poslist = []
    neglist = []
    flexicon = open('liwcdic2007.dic', encoding='latin1')
    # read all LIWC words from file
    wordlines = [line.strip() for line in flexicon]
    # each line has a word or a stem followed by * and numbers of the word classes it is in
    # word class 126 is positive emotion and 127 is negative emotion
    for line in wordlines:
        if not line == '':
            items = line.split()
            word = items[0]
            classes = items[1:]
            for c in classes:
                if c == '126':
                    poslist.append( word )
                if c == '127':
                    neglist.append( word )
    return (poslist, neglist)

# Getting positive and negative words
(poslist, neglist) = read_words()

# Creating new feature function
def LIWC_features(document,poslist,neglist):
    doc_words = set(document)
    features = {}
    for word in poslist:
        features['P_{}'.format(word)] = (word in doc_words)
    for word in neglist:
        features['N_{}'.format(word)] = (word in doc_words)
    return features

LIWC_featuresets = [(LIWC_features(d,poslist,neglist), c) for (d,c) in emaildocs]
```

- The cross-validation output is added below

```
Each fold size: 600
0 0.7783333333333333
        Precision       Recall          F1
spam          0.625      0.899       0.738
ham           0.930      0.714       0.808
1 0.745
        Precision       Recall          F1
spam          0.585      0.837       0.688
ham           0.894      0.698       0.784
2 0.69
        Precision       Recall          F1
spam          0.497      0.837       0.623
ham           0.897      0.625       0.737
3 0.7783333333333333
        Precision       Recall          F1
spam          0.654      0.857       0.742
ham           0.896      0.732       0.806
4 0.7616666666666667
        Precision       Recall          F1
spam          0.619      0.885       0.729
ham           0.914      0.692       0.788
mean accuracy 0.7506666666666667
```

After adding the Linguistic Count and Word Frequency sentiment, the accuracy from cross-validation reported is **75%** which is a big downfall from the previously reported accuracy of 96.5%

Representing new feature functions combining word frequency with sentiment lexicons

Here with the LIWC features function, we have also added word features

```
def WF_LIWC_features(document,poslist,neglist,word_features):
    dw = set(document)
    features = {}
    for word in poslist:
        features['PW_{}'.format(word)] = (word in dw)
    for word in neglist:
        features['NW_{}'.format(word)] = (word in dw)
    FD_dw = nltk.FreqDist(dw)
    for word in word_features:
        features['WF_{}'.format(word)] = FD_dw[dw]
    return features

    WF_LIWC_featuresets = [(WF_LIWC_features(d,fil_words_features), c) for (d,c) in emaildocs]
```

- The cross-validation output after considering the negation word features is added below

  Mean accuracy:

```
LIWC Sentiment Classifier accuracy
Each fold size:  600
0 0.7833333333333333
            Precision       Recall          F1
spam             0.602       0.942       0.735
ham              0.963       0.709       0.817
1 0.7533333333333333
            Precision       Recall          F1
spam             0.529       0.927       0.674
ham              0.961       0.687       0.802
2 0.6616666666666666
            Precision       Recall          F1
spam             0.377       0.921       0.535
ham              0.966       0.592       0.734
3 0.7083333333333334
            Precision       Recall          F1
spam             0.425       0.947       0.586
ham              0.977       0.642       0.775
4 0.715
            Precision       Recall          F1
spam             0.481       0.937       0.635
ham              0.966       0.635       0.766
mean accuracy 0.7243333333333333
```

After combining the word features with sentiment lexicons, the mean accuracy reported is **72.4%** which is again low compared to our other analysis.

TFIDF Scores

In the dataset we have used tfidf scores as the value of the word features instead of Boolean values. TF-IDF enables us to give us a way to associate each word in a document with a number that represents how relevant each word is in that document. It reflects how important the word is to a document in a corpus.

```
# IDF(t) = log_e(Total number of documents / Number of documents with term t in it)
doc_w = [word for (email,tag) in emaildocs for word in email]
dw = set(doc_w)
FD_idf = nltk.FreqDist(dw)
len_ed = len(emaildocs)
def IDF(word):
    if not FD_idf[word] == 0:
        IDF = math.log(len_ed/FD_idf[word])
        return IDF

def TFIDF_features(document, word_features):
    dw = set(document)
    FD_dw = nltk.FreqDist(dw)
    features = {}
    for word in word_features:
        # TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document)
        TF_word = FD_dw[word]/len(dw)
        idf_word = IDF(word)
        features['TFIDF_{}'.format(word)] = TF_word*idf_word
    return features

TFIDF_featuresets = [(TFIDF_features(d,fil_words_features), c) for (d,c) in emaildocs]
```

- The cross-validation output after considering the negation word features is added below

```
\TFIDF Classification accuracy
Each fold size: 600
0 0.7833333333333333
        Precision       Recall          F1
ham             0.963       0.709       0.817
spam            0.602       0.942       0.735
1 0.7533333333333333
        Precision       Recall          F1
ham             0.961       0.687       0.802
spam            0.529       0.927       0.674
2 0.6616666666666666
        Precision       Recall          F1
ham             0.966       0.592       0.734
spam            0.377       0.921       0.535
3 0.7083333333333334
        Precision       Recall          F1
ham             0.977       0.642       0.775
spam            0.425       0.947       0.586
4 0.715
        Precision       Recall          F1
ham             0.966       0.635       0.766
spam            0.481       0.937       0.635
mean accuracy 0.7243333333333333
```

Mean accuracy for TFIDG scores is also pretty low, coming up to **72.4%**

Representing classification using Sci Kit Learn classifier with features produced in NLTK

Naïve based classifier has good performance on smaller classification problems. For a large problem, nltk might run out of memory. For this, we prepare nltk to prepare features and then use an outside classifier. Here we choose Sci Kit Learn classifier. The features and their tags is written to a csv file using the function 'writeFeatureSets'

```
def writeFeatureSets(featuresets, outpath):
    # open outpath for writing
    f = open(outpath, 'w')
    # get the feature names from the feature dictionary in the first featureset
    featurenames = featuresets[0][0].keys()
    # create the first line of the file as comma separated feature names
    #    with the word class as the last feature name
    featurenameline = ''
    for featurename in featurenames:
        # replace forbidden characters with text abbreviations
        featurename = featurename.replace(',','CM')
        featurename = featurename.replace("'","DQ")
        featurename = featurename.replace('"','QU')
        featurenameline += featurename + ','
    featurenameline += 'class'
    # write this as the first line in the csv file
    f.write(featurenameline)
    f.write('\n')
    # convert each feature set to a line in the file with comma separated feature values,
    # each feature value is converted to a string
    #   for booleans this is the words true and false
    #   for numbers, this is the string with the number
    for featureset in featuresets:
        featureline = ''
        for key in featurenames:
            featureline += str(featureset[0][key]) + ','
        featureline += featureset[1]
        # write each feature set values to the file
        f.write(featureline)
        f.write('\n')
    f.close()

# Using the filtered feature sets
outpath = "SK_Learn_features.csv"
writeFeatureSets(fil_featuresets, outpath)
```

- The cross-validation output after considering the negation word features is added below.

```
              precision    recall  f1-score   support

         ham       1.00      0.96      0.98      1500
        spam       0.96      1.00      0.98      1500

    accuracy                           0.98      3000
   macro avg       0.98      0.98      0.98      3000
weighted avg       0.98      0.98      0.98      3000



Predicted   ham  spam   All
Actual
ham        1444    56  1500
spam          7  1493  1500
All        1451  1549  3000
Shape of feature data - num instances with num features + class label
(3000, 2501)
** Results from Logistic Regression with liblinear
              precision    recall  f1-score   support

         ham       1.00      0.96      0.98      1500
        spam       0.96      1.00      0.98      1500

    accuracy                           0.98      3000
   macro avg       0.98      0.98      0.98      3000
weighted avg       0.98      0.98      0.98      3000



Predicted   ham  spam   All
Actual
ham        1444    56  1500
spam          7  1493  1500
All        1451  1549  3000
```

Using the SciKit classifier on features, the accuracy reported is **97.9%** which is a vast improvement to our previous analysis.