NLP - IST 664 Winterlude
Final Project
Kabir Thakur - 427482939

Introduction

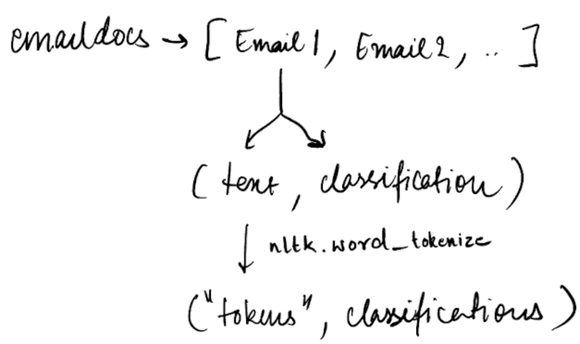
In this project we use data from the Enron public email corpus to build several classifiers to identify spam emails. Spam email detection is a common problem in natural language processing. Thoughout this project we will be using the NLTK or the Natural Language Tool Kit library available on python. Our goal is to develop featuresets for the classifier to be trained on. Once trained, we used test our classifier for accuracy, precision, recall etc.

We start with reading the data from the email corpus provided to us with the code provided. The program is designed in such a way that we can choose the number of spam and ham emails we want to use each time.

The function processspamham(dirPath,limitStr) is the main function that has most of the control throughout the program. It processes the words by performing tokenzation.

```
# 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'))
```

The NLTK has inbuilt functions for text tokenization one of which word_tokenize is being used here. The program creates a list called emaildocs and stores all the spam emails and ham emails as word-tokens inside a tuple in the list. The tuple has two objects the tokens and the classification.



We use random.shuffle to shuffle the data.

We get a list of all words initially that it’s easier to remove the stopwords from each email. To filter the tokens I have removed the stopwords from the emails. The stopwords list used is from nltk.

```
# continue as usual to get all words

all_words=[]
for email in emaildocs:
  for words in email:
    for word in words:
      all_words.append(word)
all_original = all_words
```

We do some processing before we create featuresets and build the following features

Featuresets

Unigram

The unigram featureset also called the Bag of Words (BOW) gets the words and their frequency using the FreqDist function from NLTK. From this list we pick the 2000 most common words and prepare a list of word features. This list is then sent to the function word_freq(email, word_features) for each email in emaildocs. The function first converts the email to a set data type which removes all duplicate word tokens and then returns a dictionary with the words in contains({}) tag as keys and True or False as value. Note that only those words are returned which are present in the list of the top 2000 words by frequency. These features are clubbed with the classification and saved as tuples in a list.

```
#Unigram processed
distribution = nltk.FreqDist(all_words)
word_items = distribution.most_common(2000)
word_features = [word for (word, freq) in word_items]
```

```
featureset_freq = [(word_freq(email, word_features),classification) for (email,classification) in emaildocs]
```

```
def word_freq(email,word_features):
    email_words=set(email)
    features={}
    for word in word_features:
        features['contains({})'.format(word)]=(word in email_words)
    return features
```

This featureset is ready for being used to train a model. It also acts a baseline for all other experimental featuresets.

Unigram Unprocessed

To see the effects of not processing the tokens we also get the unigrams from the unprocessed emails. We use the same function word_freq to build the featureset

```
#Unigram unprocessed
distribution_o = nltk.FreqDist(all_original)
word_items_o = distribution.most_common(2000)
word_features_o = [word for (word, freq) in word_items_o]
```

```
featureset_freq_original = [(word_freq(email, word_features),classification) for (email,classification) in emaildocs]
```

Bigram
Bigrams are two words that appear together in the text. We use nltk.collocations to create an instance of a BigramAssocMeasures class.

```
bigram_measures = nltk.collocations.BigramAssocMeasures()
finder = BigramCollocationFinder.from_words(tokens,window_size=3)
bigram_features_all = finder.nbest(bigram_measures.chi_sq, 3000)
```

The second line creates an instance of the Bigram Collocation Finder class and initializes it with the list of tokens and a window_size of 3 which means it will only consider bigrams which appear within 3 words of each other. The last line uses the nbest method to extract 3000 most likely bigrams based on chi-sq measure of association.

```
featureset_bigram = [(bigram_features(email,word_features,bigram_features_all),classification) for (email,classification) in emaildocs]
```

```
def bigram_features(email, word_features, bigram_features):
  email_words = set(email)
  email_bigrams = nltk.bigrams(email)
  features = {}
  for word in word_features:
    features['contains({})'.format(word)] = (word in email_words)
  for bigram in bigram_features:
    features['bigram({} {})'.format(bigram[0], bigram[1])] = (bigram in email_bigrams)
  return features
```

The bigram_features function does the same thing as the unigram function word_freq except it also includes the bigrams using nltk.bigrams(email) and it returns the bigram features keeping the unigram features as a baseline.

Trigram
As a new feature I included the trigram featureset. This works same as Bigram features but for trigrams

```
#Trigram processed
trigram_measures = nltk.collocations.TrigramAssocMeasures()
trigram_finder = TrigramCollocationFinder.from_words(tokens)
trigram_scores = trigram_finder.score_ngrams(TrigramAssocMeasures.chi_sq)
trigrams = [trigram for trigram,score in trigram_scores]
```

```
featureset_trigram = [(trigram_features(email,word_features,trigram_scores),classification) for (email, classification) in emaildocs]
```

```
def trigram_features(email, word_features, trigrams):
  email_words = set(email)
  trigram_finder_email = TrigramCollocationFinder.from_words(email)
  trigram_scores_email = trigram_finder_email.score_ngrams(TrigramAssocMeasures.chi_sq)
  email_trigrams = [trigram for trigram,score in trigram_scores_email]
  features = {}
  for word in word_features:
    features['contains({})'.format(word)] = (word in email_words)
  for trigram in trigrams:
    features['trigram({} {} {})'.format(trigram[0], trigram[1], trigram[2])] = (trigram in email_trigrams)
  return features
```

POS tags
Part of speech tags give significance to words in sentence structures. Though I have implemented them in my program, I have not used sent_tokenize to tokenize the sentences in the POS tags

```
featureset_POS = [(POS_features(email, word_features), classification) for (email, classification) in emaildocs]
```

```
def POS_features(email, word_features):
    email_words = set(email)
    tagged_words = nltk.pos_tag(email)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in email_words)
    numNoun = 0
    numVerb = 0
    numAdj = 0
    numAdverb = 0
    for (word, tag) in tagged_words:
        if tag.startswith('N'): numNoun += 1
        if tag.startswith('V'): numVerb += 1
        if tag.startswith('J'): numAdj += 1
        if tag.startswith('R'): numAdverb += 1
    features['nouns'] = numNoun
    features['verbs'] = numVerb
    features['adjectives'] = numAdj
    features['adverbs'] = numAdverb
    return features
```

This function uses the nltk.pos_tag to tag words. We then get our base unigram features. The next features to be added are the number of nouns, verbs , adjectives and adverbs.

Subjectivity featureset
The subjectivity featureset uses a predefined set of words. This featureset is quite useless for this dataset as it was given for movie reviews. However, it is still a list of words we can experiment with thus I've included it as a feature.

```
featureset_SL = [(SL_features(email, word_features), classification) for (email, classification) in emaildocs]
```

```
def SL_features(email, word_features):

  SL = readSubjectivity()
  email_words = set(email)
  features = {}
  for word in word_features:
    features['contains({})'.format(word)] = (word in email_words)
  # count variables for the 4 classes of subjectivity
  weakPos = 0
  strongPos = 0
  weakNeg = 0
  strongNeg = 0
  for word in email_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)

  if 'positivecount' not in features:
    features['positivecount']=0
  if 'negativecount' not in features:
    features['negativecount']=0
  return features
```

```
def readSubjectivity():
    path = "sub.tff"
    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
```

It uses the function SL_features to generate the features. This function in turn calls the readSubjectivity function which reads the file which for each word defines the strength of subject, part of speech, if it has a stem word or not and it's polarity. We use this list to construct a count of positive and negative words keeping the bag of words as baseline. The count of positive and negative words are the features.

Combined
The combined featureset is a combination of SL, unigram, bigram and trigram features.

```
featureset_combined = [(combined_features(email, word_features, bigram_features_all, trigram_scores),classification) for email,
    classification in emaildocs]
```

```
def combined_features(email, word_features, bigram_features_all, trigrams):
    email_words = set(email)
    SL = readSubjectivity()
    email_bigrams=nltk.bigrams(email)
    trigram_finder_email = TrigramCollocationFinder.from_words(email)
    trigram_scores_email = trigram_finder_email.score_ngrams(TrigramAssocMeasures.chi_sq)
    email_trigrams = [trigram for trigram,score in trigram_scores_email]
    features={}
    weakPos = 0
    strongPos = 0
    weakNeg = 0
    strongNeg = 0
    for word in email_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)
    if 'positivecount' not in features:
      features['positivecount']=0
    if 'negativecount' not in features:
      features['negativecount']=0
    for word in word_features:
      features['contains({})'.format(word)] = (word in email_words)
    for bigram in bigram_features_all:
      features['bigram({} {})'.format(bigram[0], bigram[1])] = (bigram in email_bigrams)
    for trigram in trigrams:
      features['trigram({} {} {})'.format(trigram[0], trigram[1], trigram[2])] = (trigram in email_trigrams)
      return features
```

Model Training
For all the featuresets we use both NaiveBayes Classifier from NLTK and SVM from SciKit Learn.
The function model_training takes in the featureset and splits it into a 70:30 train test split. We train the Naive Bayes classifier here and use the nltk ConfusionMatrix function to get the confusion matrix and classify.accuracy for accuracy. We use the cross_validation function to run a 10 fold cross validation and give us a mean accuracy. The Eval measures function calculates the f1 score, precision and recall values. We can also use this function to get the most informative features.

```python
def model_training(featuresets):
  training_size = int(0.7*len(featuresets))
  test_set = featuresets[:training_size]
  training_set = featuresets[training_size:]
  classifier = nltk.NaiveBayesClassifier.train(training_set)
  goldlist = []
  predictedlist = []
  for (features, label) in test_set:
      goldlist.append(label)
      predictedlist.append(classifier.classify(features))
  cm = nltk.ConfusionMatrix(goldlist, predictedlist)
  print ("Confusion Matrix : ")
  print(cm)
  print ("Model Accuracy : ")
  print (nltk.classify.accuracy(classifier, test_set))
  print ("k-fold cross validation mean accuracy : ")
  print (cross_validation(10,featuresets))
  print ("Other Evaluation Measures")
  eval_measures(goldlist,predictedlist)
```

```python
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))
```

```python
def cross_validation(num_folds, featuresets):
      subset_size = int(len(featuresets)/num_folds)
      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
      mean_accuracy = sum(accuracy_list) / num_folds
      return(mean_accuracy)
```

The writeFeatureSet function is used to create a text file in proper format for the SVM classifier and the program run_sklearn_model_performance.py is used to run the scikit learn svm.

Results

We can see the results on the next page. The left pane represents the Naive Bayes classifier and the right pane represents the SVM classifier. Most of the accuracy is attained by the Bag of words feature which is common for all. We can also see that between the scores only from Naive Bayes classifier everytime we use the Subjectivity lexicon (SL and Combined) our accuracy goes down but a very small amount. For the SVM classifier while using SL and POS we get a convergence warning as we have too many iterations.

Naive Bayes Classifier

```
-------------------------------
Feature : Unprocessed BOW
Confusion Matrix :
     |       s |
     |   h   p |
     |   a   a |
     |   m   m |
-----+---------+
 ham |<322> 34 |
spam |   2<342>|
-----+---------+
(row = reference; col = test)

Model Accuracy :
0.9485714285714286
k-fold cross validation mean accuracy :
0.9550000000000001
Other Evaluation Measures
        Precision       Recall          F1
ham             0.904       0.994       0.947
spam            0.994       0.910       0.950
-------------------------------
```

```
-------------------------------
Feature : BOW
Confusion Matrix :
     |       s |
     |   h   p |
     |   a   a |
     |   m   m |
-----+---------+
 ham |<322> 34 |
spam |   2<342>|
-----+---------+
(row = reference; col = test)

Model Accuracy :
0.9485714285714286
k-fold cross validation mean accuracy :
0.9550000000000001
Other Evaluation Measures
        Precision       Recall          F1
ham             0.904       0.994       0.947
spam            0.994       0.910       0.950
-------------------------------
```

```
-------------------------------
Feature : Bigram
Confusion Matrix :
     |       s |
     |   h   p |
     |   a   a |
     |   m   m |
-----+---------+
 ham |<322> 34 |
spam |   2<342>|
-----+---------+
(row = reference; col = test)

Model Accuracy :
0.9485714285714286
k-fold cross validation mean accuracy :
0.9550000000000001
Other Evaluation Measures
        Precision       Recall          F1
ham             0.904       0.994       0.947
spam            0.994       0.910       0.950
-------------------------------
```

SVM Classifier

```
kabirthakur@Kabirs-MBP EmailSpamCorpora % python3 sk.py corpus/BOW_unprocessed
Shape of feature data - num instances with num features + class label
(1000, 2001)
** Results from Logistic Regression with liblinear
              precision    recall  f1-score   support

         ham       0.99      0.94      0.96       500
        spam       0.94      0.99      0.96       500

    accuracy                           0.96      1000
   macro avg       0.96      0.96      0.96      1000
weighted avg       0.96      0.96      0.96      1000



Predicted  ham  spam   All
Actual
ham        470    30   500
spam         7   493   500
All        477   523  1000
kabirthakur@Kabirs-MBP EmailSpamCorpora %
```

```
kabirthakur@Kabirs-MBP EmailSpamCorpora % python3 sk.py corpus/BOW
Shape of feature data - num instances with num features + class label
(1000, 2001)
** Results from Logistic Regression with liblinear
              precision    recall  f1-score   support

         ham       0.99      0.94      0.96       500
        spam       0.94      0.99      0.96       500

    accuracy                           0.96      1000
   macro avg       0.96      0.96      0.96      1000
weighted avg       0.96      0.96      0.96      1000



Predicted  ham  spam   All
Actual
ham        470    30   500
spam         7   493   500
All        477   523  1000
```

```
kabirthakur@Kabirs-MBP EmailSpamCorpora % python3 sk.py corpus/Bigram
Shape of feature data - num instances with num features + class label
(1000, 2097)
** Results from Logistic Regression with liblinear
              precision    recall  f1-score   support

         ham       0.99      0.94      0.96       500
        spam       0.94      0.99      0.96       500

    accuracy                           0.96      1000
   macro avg       0.96      0.96      0.96      1000
weighted avg       0.96      0.96      0.96      1000



Predicted  ham  spam   All
Actual
ham        470    30   500
spam         7   493   500
All        477   523  1000
```

Naive Bayes Classifier

```
-------------------------------
Feature : POS
Confusion Matrix :
     |       s |
     |   h   p |
     |   a   a |
     |   m   m |
-----+---------+
 ham |<322> 34 |
spam |   2<342>|
-----+---------+
(row = reference; col = test)

Model Accuracy :
0.9485714285714286
k-fold cross validation mean accuracy :
0.9570000000000001
Other Evaluation Measures
        Precision       Recall          F1
ham          0.904       0.994       0.947
spam         0.994       0.910       0.950
-------------------------------
```

```
-------------------------------
Feature : SL
Confusion Matrix :
     |       s |
     |   h   p |
     |   a   a |
     |   m   m |
-----+---------+
 ham |<321> 35 |
spam |   2<342>|
-----+---------+
(row = reference; col = test)

Model Accuracy :
0.9471428571428572
k-fold cross validation mean accuracy :
0.958
Other Evaluation Measures
        Precision       Recall          F1
ham          0.902       0.994       0.946
spam         0.994       0.907       0.949
-------------------------------
```

```
-------------------------------
Feature : Trigram
Confusion Matrix :
     |       s |
     |   h   p |
     |   a   a |
     |   m   m |
-----+---------+
 ham |<322> 34 |
spam |   2<342>|
-----+---------+
(row = reference; col = test)

Model Accuracy :
0.9485714285714286
k-fold cross validation mean accuracy :
0.9540000000000001
Other Evaluation Measures
        Precision       Recall          F1
ham          0.904       0.994       0.947
spam         0.994       0.910       0.950
-------------------------------
```

SVM Classifier

```
              precision    recall  f1-score   support

         ham       0.98      0.94      0.96       500
        spam       0.94      0.98      0.96       500

    accuracy                           0.96      1000
   macro avg       0.96      0.96      0.96      1000
weighted avg       0.96      0.96      0.96      1000


Predicted  ham  spam   All
Actual
ham        469    31   500
spam        10   490   500
All        479   521  1000
```

```
              precision    recall  f1-score   support

         ham       0.99      0.94      0.97       500
        spam       0.94      0.99      0.97       500

    accuracy                           0.97      1000
   macro avg       0.97      0.97      0.97      1000
weighted avg       0.97      0.97      0.97      1000


Predicted  ham  spam   All
Actual
ham        470    30   500
spam         3   497   500
All        473   527  1000
```

```
kabirthakur@Kabirs-MBP EmailSpamCorpora % python3 sk.py corpus/Trigram
Shape of feature data - num instances with num features + class label
(1000, 2052)
** Results from Logistic Regression with liblinear
              precision    recall  f1-score   support

         ham       0.99      0.94      0.96       500
        spam       0.94      0.99      0.96       500

    accuracy                           0.96      1000
   macro avg       0.96      0.96      0.96      1000
weighted avg       0.96      0.96      0.96      1000


Predicted  ham  spam   All
Actual
ham        471    29   500
spam         7   493   500
All        478   522  1000
```

Naive Bayes Classifier

```
------------------------------
Feature : Combined
Confusion Matrix :
     |       s |
     |   h   p |
     |   a   a |
     |   m   m |
-----+---------+
 ham |<321> 35 |
spam |    2<342>|
-----+---------+
(row = reference; col = test)

Model Accuracy :
0.9471428571428572
k-fold cross validation mean accuracy :
0.958
Other Evaluation Measures
        Precision       Recall          F1
ham             0.902       0.994       0.946
spam            0.994       0.907       0.949
```

SVM Classifier

```
kabirthakur@Kabirs-MBP EmailSpamCorpora % python3 sk.py corpus/Combined
Shape of feature data - num instances with num features + class label
(1000, 2001)
** Results from Logistic Regression with liblinear
              precision    recall  f1-score   support

         ham       0.99      0.94      0.96       500
        spam       0.94      0.99      0.96       500

    accuracy                           0.96      1000
   macro avg       0.96      0.96      0.96      1000
weighted avg       0.96      0.96      0.96      1000



Predicted  ham  spam   All
Actual
ham        470    30   500
spam         7   493   500
All        477   523  1000
```

Conclusion

The spam classifier can further be deployed onto email servers to curb spam emails. There are quite a few other features that can be added as a future scope of development. One featureset can be a list of subjectivity lexicon specifically made for spam emails.