N'Dea Jackson & Michael Armesto
IST 664 – Natural Language Processing
Assignment: Final Project: Kaggle Movie Reviews
March 25, 2021

Pre-Processing

Before adding any new code to the shell that was provided for this project, the code was run as-is to get a better understanding of exactly what it would do. After using the "cd" command to navigate to the appropriate folder, the code that was entered into the command line interface was "python classifyKaggle.py corpus 10". This code was designed to read the 156,060 phrases in the corpus and then use 10 of those phrases randomly. The output of this code can be seen in the screenshot below.

```
(base) NDeasPesonalMBP:kagglemoviereviews ndeajackson$ python classifyKaggle.py corpus 10
Read 156060 phrases, using 10 random phrases
['Heidegger', '2']
['Music episode', '2']
['a quarter', '2']
['appears to have given up on in favor of sentimental war movies in the vein of ` We Were Soldiers', '2']
['bathing', '2']
["flawless film , -LRB- Wang -RRB- emerges in the front ranks of China 's now numerous , world-renowned filmmakers .", '3']
['the occasional bursts of sharp writing alternating with lots of sloppiness', '2']
['in which the hero might have an opportunity to triumphantly sermonize', '3']
['the rolling of a stray barrel or', '2']
['SO De Palma', '2']
(['Heidegger'], 2)
(['Music', 'episode'], 2)
(['a', 'quarter'], 2)
(['appears', 'to', 'have', 'given', 'up', 'on', 'in', 'favor', 'of', 'sentimental', 'war', 'movies', 'in', 'the', 'vein', 'of', '`', 'We',
'Were', 'Soldiers'], 2)
(['bathing'], 2)
(['flawless', 'film', ',', '-LRB-', 'Wang', '-RRB-', 'emerges', 'in', 'the', 'front', 'ranks', 'of', 'China', "'s", 'now', 'numerous', ',',
 'world-renowned', 'filmmakers', '.'], 3)
(['the', 'occasional', 'bursts', 'of', 'sharp', 'writing', 'alternating', 'with', 'lots', 'of', 'sloppiness'], 2)
(['in', 'which', 'the', 'hero', 'might', 'have', 'an', 'opportunity', 'to', 'triumphantly', 'sermonize'], 3)
(['the', 'rolling', 'of', 'a', 'stray', 'barrel', 'or'], 2)
(['SO', 'De', 'Palma'], 2)
```

```
        ####################
        ## pre processing ##
        ####################

        ## some stopwords
        stopwords = nltk.corpus.stopwords.words('english')
        newstopwords = [word for word in stopwords if word not in ['not', 'no', 'can','don', 't']]

        def preprocessing(doc):
          # make all the words lowercase
          word_list = re.split('\s+', doc.lower())
          # remove puncutation and numbers
          punctuation = re.compile(r'[-.?!/\%@,":;()|0-9]')
          word_list = [punctuation.sub("", word) for word in word_list]
          final_word_list = []
          for word in word_list:
            if word not in newstopwords:
              final_word_list.append(word)
          line = " ".join(final_word_list)
          return line
```

Next, some lines of code needed to be added to the *kaggleClassify.py* file. To help structure some of the code for this project, Week 9 Lab was referenced. The first modification that was made to the shell code was made to the random sample line. In order for us to truly get an understanding of how the changes that were being made to the code were effecting the overall outcome, we decided to set a seed. By setting a seed, we are ensuring that every time the data is sampled, we are getting the exact same sample. This would also allow us to truly monitor and compare the overall accuracy scores that we would receive. The code that was modified can be seen in the screenshot below.

```
# pick a random sample of length limit because of phrase overlapping sequences
random.Random(715).shuffle(phrasedata)
phraselist = phrasedata[:limit]
```

After setting a seed for our random sampling, some feature sets needed to be created. The first two features were a *get_words* and a *get_word_features*. *Get_words*

Get words and features
Get_words

```
def get_words(docs):
  all_words = []
  for (words, sentiment) in docs:
    # more than 3 length
    # possible_words = [x for x in words if len(x) >= 3]
    # all_words.extend(possible_words)
    all_words.extend(words)
  return all_words
```

Get_word_features
With this line of code, we defined a function that gets the 200 most frequently appearing words within the corpus.

```
def get_word_features(wordlist):
  wordlist = nltk.FreqDist(wordlist)
  word_features = [w for (w, c) in wordlist.most_common(200)]
  return word_features
```

Baseline feature set
Document_features
With this line of code, we defined a function that defines the features (keywords) of a document for a BOW/unigram baseline. Each of these features is 'contains(keyword)' and is true or false depending on whether that keyword is in the document.

```
def document_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['V_{}'.format(word)] = (word in document_words)
    return features
```

Preprocessing

After the baseline unigram document features were obtained, we decided to implement some preprocessing of the documents. Stop words from the English NLTK corpus were removed. This list was later modified to exclude negator words such as "not", which would be used later in a different featureset. We also used this opportunity to make all words lowercase, so that capitalization would

not play a role. Another preprocessing step was to exclude phrases with fewer than four words. While it is a preprocessing step, this filter is carried out by a different function, a modified version of the aforementioned get_words function. One can see the filter code commented out in the above screenshot of get_words.

```
####################
## pre processing ##
####################

## some stopwords
stopwords = nltk.corpus.stopwords.words('english')
newstopwords = [word for word in stopwords if word not in ['not', 'no', 'can','don', 't']]

def preprocessing(doc):
  # make all the words lowercase
  word_list = re.split('\s+', doc.lower())
  # remove puncutation and numbers
  punctuation = re.compile(r'[-.?!/\%@,":;()|0-9]')
  word_list = [punctuation.sub("", word) for word in word_list]
  final_word_list = []
  for word in word_list:
    if word not in newstopwords:
      final_word_list.append(word)
  line = " ".join(final_word_list)
  return line
```

Subjectivity feature set

With this chunk of code, we defined a function that defined features that included word counts of subjectivity words. First, the function reads in the subject clues lexicon that was provided in a previous class. The function then looks at each word in the document, matches it to a word in the lexicon (if available), and assigns the corresponding polarity and subjectivity values. These values include weakly subjective, strongly subjective, positive, and negative. In our interpretation, words that are weakly subjective and positive are deemed weakly positive. Words that are weakly subjective and negative are deemed weakly negative, and so on. Based on these word features, a phrase is assigned a total value of positive and negative word counts. The negative count includes the number of weakly negative words plus 2 times the number of strongly negative words. This is done in an effort to scale the scores of each word. The positive feature also has a similar definition to that of the negative feature. This definition does not count the neutral words.

```
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 = "./SentimentLexicons/subjclueslen1-HLTEMNLP05.tff"
SL = readSubjectivity(SLpath)
def SL_features(document, word_features, SL):
  document_words = set(document)
  features = {}
  for word in word_features:
    features['contains({})'.format(word)] = (word in document_words)
  # 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)

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

Negation words feature set

Negation words include words such as “not”, “never”, and “no”. The word not is also included in contraction words such as “doesn’t”, “can’t” and “won’t”. One of the strategies that is often used with negation words includes negating the word following the negation word. Other strategies involve negating all words up to the next punctuation. The strategy that we used for this particular project was to go through all of the document words in order, adding all of the word features, and when we get to a word that follows a negation word, the feature would be changed to negated word. All of the word features and the not word features were set to false to begin. Looping through the words in each phrase, the function looks for negation words, and applies a NOT feature to the word that follows. Words not preceded by negation words are treated as simple unigrams, as in the baseline featureset.

```
negationwords = ['no', 'not', 'never', 'none', 'nowhere', 'nothing', 'noone', 'rather',
                 'hardly', 'scarcely', 'rarely', 'seldom', 'neither', 'nor']

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

Bigram feature set

get_bigram_features
With this line of code, we defined a function that creates a bigram finder on all of the words in a sequence.

bigram_document_features
With this line of code, we defined a function that defines features that include words as before and adds the most frequent significant bigrams. This function takes the list of words in a document as an argument and returns a feature dictionary. It depends on the variables *word_features* and *bigram_features*.

```
def get_bigram_features(tokens):
  bigram_measures = nltk.collocations.BigramAssocMeasures()
  # create the bigram finder on all the words in sequence
  finder = BigramCollocationFinder.from_words(tokens,window_size=3)
  ## optionally filter for frequency
  #finder.apply_freq_filter(6)
  # define the top 1000 bigrams using the chi squared measure
  bigram_features = finder.nbest(bigram_measures.chi_sq, 1000)
  return bigram_features[:500]

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['contains({})'.format(word)] = (word in document_words)
  for bigram in bigram_features:
    features['bigram({} {})'.format(bigram[0], bigram[1])] = (bigram in document_bigrams)
  return features
```

Part of Speech (POS) Feature Set

With these lines of code, we defined a function that takes a document list of words and then returns a feature dictionary. It works by running the default POS tagger on the document and counting four types of POS tags: nouns, verbs, adjectives and adverbs. Every phrase is then assigned a count of each part of speech, and this count is used as a feature.

```
######################
##  POS featureset  ###
######################

# this function takes a document list of words and returns a feature dictionay
#   it depends on the variable word_features
# it runs the default pos tagger (the Stanford tagger) on the document
#   and counts 4 types of pos tags to use as features

def POS_features(document, word_features):
    document_words = set(document)
    tagged_words = nltk.pos_tag(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_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

####################
```

Linguistic and Inquiry Word Count (LIWC) Sentiment

With this chunk of code, both negative and positive word prefixes were initialized from the Linguistic and Inquiry Word Count, a dictionary that assigns properties to over 5,000 English words. Included in the LIWC are lists of positive and negative words. These negative and positive prefixes were taken from *poslist* and *neglist.* Positive counts were identified as *poscount* and

negative counts were identified as *negcount*. At the end of this defintion, the features themselves were returned.

```
####################
## LIWC Sentiment ##
####################

# initialize positve and negative word prefix lists from LIWC
#   note there is another function isPresent to test if a word's prefix is in the list
(poslist, neglist) = sentiment_read_LIWC_pos_neg_words.read_words()

def LIWC_features(doc, word_features,poslist,neglist):
  doc_words = set(doc)
  features = {}
  for word in word_features:
    features['contains({})'.format(word)] = (word in doc_words)
  poscount = 0
  negcount = 0
  for word in doc_words:
    if sentiment_read_LIWC_pos_neg_words.isPresent(word,poslist):
      poscount += 1
    if sentiment_read_LIWC_pos_neg_words.isPresent(word,neglist):
      negcount += 1
    features['positivecount'] = poscount
    features['negativecount'] = negcount
  if 'positivecount' not in features:
    features['positivecount'] = 0
  if 'negativecount' not in features:
    features['negativecount'] = 0
  return features
```

Hybrid Sentiment Feature Set

The two sentiment lexicons used in these experiments, the subject clues lexicon and the LIWC were both found to be fairly accurate and very interesting predictors. We decided to attempt another featureset using a hybrid of both lexicons. This function first runs through each word and checks it against the subject clues lexicon, which was found to be the more accurate predictor of the two. If the word is included there, a polarity score is assigned. If not, the function then checks to see if the LIWC can score the word. Because the subject clues lexicon creates four word categories while the LIWC only creates two, we decided to scale the scores so that a positive from the LIWC would be equivalent to a strong positive from the subject clues lexicon, and a weak positive would again be worth half of that when added to the positivity count. Of course, the same logic applies to negative words.

```
################################
## Hybrid Sentiment Featureset ##
################################

def hybrid_features(document, word_features, SL, poslist, neglist):
  document_words = set(document)
  features = {}
  for word in word_features:
    features['contains({})'.format(word)] = (word in document_words)
  # count variables for the 4 classes of subjectivity
  weakPos = 0
  strongPos = 0
  weakNeg = 0
  strongNeg = 0
  poscount = 0
  negcount = 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

    elif sentiment_read_LIWC_pos_neg_words.isPresent(word,poslist):
      poscount += 2
    elif sentiment_read_LIWC_pos_neg_words.isPresent(word,neglist):
      negcount += 2

    features['positivecount'] = weakPos + (2 * strongPos) + (2 * poscount)
    features['negativecount'] = weakNeg + (2 * strongNeg) + (2 * negcount)

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

Accuracy and confusion matrix

In order to provide a standard accuracy assessment, the calculate_accuracy function was implemented to simply test if phrases were being placed in the right bins, so to speak. Here a training set size is determined, and the featureset is split into a training and testing set. The below code shows a small training size of only 10% of the featureset, but in later runs of the code this was increased to 50% allow the model to read more words before attempting to classify the testing set. This was found to increase overall accuracy, but only to a slight degree. After NLTK's built-in Naïve Bayes classifier is run, the function prints the most informative individual features. Then, a confusion matrix is printed. This displays the count of each category in a cross-table, allowing us to quickly see how many were correctly and incorrectly categorized, and to which bin they were placed.

```
def calculate_accuracy(featuresets):
  print("Training and testing a classifier ")

  training_size = int(0.1*len(featuresets))
  test_set = featuresets[:training_size]
  training_set = featuresets[training_size:]
  classifier = nltk.NaiveBayesClassifier.train(training_set)

  print("Accuracy of classifier : ")
  print(nltk.classify.accuracy(classifier, test_set))
  print("--------------------------------------------------")
  print("Showing most informative features")
  print(classifier.show_most_informative_features(30))
  print("--------------------------------------------------")
  print("precision, recall and F-measure scores")

  # use NLTK to compute evaluation measures from a reflist of gold labels
  #    and a testlist of predicted labels for all labels in a list
  # returns lists of precision and recall for each label

  goldlist = []
  predictedlist = []

  for (features, label) in test_set:
    goldlist.append(label)
    predictedlist.append(classifier.classify(features))

  cm = nltk.ConfusionMatrix(goldlist, predictedlist)
  print(cm.pretty_format(sort_by_count=True, show_percents=False, truncate = 9))
```

Cross Validation

In order to increase the training data available, cross validation was implemented. We used classic k-fold cross validation, a technique that divides the available data into groups or "folds". One at a time, each group is held out as test data, while the other groups are used as training data. The process is repeated until each group has had it's "turn" as the test data once. Using this method, you can maximize the amount of training and testing data available, rather than simply splitting the data into two parts. The procedure also gives a more robust look at accuracy. This cross validation function takes the number of folds, the feature sets, and the labels and iterates over the folds using different sections for training and testing.

This analysis also allows us to look further than accuracy, and calculate recall and precision. Recall is the number of true positives divided by the number of true positives plus false negatives. Precision, on the other hand, is the number of true positives divided by the number of true positives plus false positives. Essentially recall examines the model's ability to find any negative documents, for example, while precision examines the model's ability to correctly identify negative documents. The harmonic mean of the two measures, also called the F1 score, finds the ideal blend of precision and recall.

```
#####################
## Cross Validation ##
#####################

# this function takes the number of folds, the feature sets and the labels
# it iterates over the folds, using different sections for training and testing in turn
#   it prints the performance for each fold and the average performance at the end
def cross_validation_PRF(num_folds, featuresets, labels):
    subset_size = int(len(featuresets)/num_folds)
    print('Each fold size:', subset_size)
    # for the number of labels - start the totals lists with zeroes
    num_labels = len(labels)
    total_precision_list = [0] * num_labels
    total_recall_list = [0] * num_labels
    total_F1_list = [0] * num_labels

    # 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 to produce the gold and predicted labels
        goldlist = []
        predictedlist = []
        for (features, label) in test_this_round:
            goldlist.append(label)
            predictedlist.append(classifier.classify(features))

        # computes evaluation measures for this fold and
        #   returns list of measures for each label
        print('Fold', i)
        (precision_list, recall_list, F1_list) \
                 = eval_measures(goldlist, predictedlist, labels)
        # take off triple string to print precision, recall and F1 for each fold
        '''
        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]))
        '''
        # for each label add to the sums in the total lists
        for i in range(num_labels):
            # for each label, add the 3 measures to the 3 lists of totals
            total_precision_list[i] += precision_list[i]
            total_recall_list[i] += recall_list[i]
            total_F1_list[i] += F1_list[i]

    # find precision, recall and F measure averaged over all rounds for all labels
    # compute averages from the totals lists
    precision_list = [tot/num_folds for tot in total_precision_list]
    recall_list = [tot/num_folds for tot in total_recall_list]
    F1_list = [tot/num_folds for tot in total_F1_list]
    # the evaluation measures in a table with one row per label
    print('\nAverage Precision\tRecall\t\tF1 \tPer Label')
    # 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]))
```

```
# print macro average over all labels - treats each label equally
print('\nMacro Average Precision\tRecall\t\tF1 \tOver All Labels')
print('\t', "{:10.3f}".format(sum(precision_list)/num_labels), \
      "{:10.3f}".format(sum(recall_list)/num_labels), \
      "{:10.3f}".format(sum(F1_list)/num_labels))

# for micro averaging, weight the scores for each label by the number of items
#    this is better for labels with imbalance
# first intialize a dictionary for label counts and then count them
label_counts = {}
for lab in labels:
  label_counts[lab] = 0
# count the labels
for (doc, lab) in featuresets:
  label_counts[lab] += 1
# make weights compared to the number of documents in featuresets
num_docs = len(featuresets)
label_weights = [(label_counts[lab] / num_docs) for lab in labels]
print('\nLabel Counts', label_counts)
#print('Label weights', label_weights)
# print macro average over all labels
print('Micro Average Precision\tRecall\t\tF1 \tOver All Labels')
precision = sum([a * b for a,b in zip(precision_list, label_weights)])
recall = sum([a * b for a,b in zip(recall_list, label_weights)])
F1 = sum([a * b for a,b in zip(F1_list, label_weights)])
print( '\t', "{:10.3f}".format(precision), \
  "{:10.3f}".format(recall), "{:10.3f}".format(F1))
```

Evaluate Measures

This function is used to compute precision, recall, and F1 for eac of the labels and for any particular number of labels. The inputs for this function were the fold labels and the list of predictions. The outcomes that were to be expected from this function were the prints precision, recall, and F1 for each label.

```
# Function to compute precision, recall and F1 for each label
#  and for any number of labels
# Input: list of gold labels, list of predicted labels (in same order)
# Output: returns lists of precision, recall and F1 for each label
#      (for computing averages across folds and labels)
def eval_measures(gold, predicted, labels):

    # 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
        # for small numbers, guard against dividing by zero in computing measures
        if (TP == 0) or (FP == 0) or (FN == 0):
          recall_list.append (0)
          precision_list.append (0)
          F1_list.append(0)
        else:
          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
    return (precision_list, recall_list, F1_list)
```

Test File

This chunk of code creates a test file using the inputs of *featuresets, test_featuresets,* and *fileName*. It then sets the *test_set* to *test_featuresets* and sets the *training_set* to *featuresets*.

```
######################
##  create test file ##
######################

def create_test_submission(featuresets,test_featuresets,fileName):
  print("-------------------------------------------------------")
  print("Training and testing a classifier ")
  test_set = test_featuresets
  training_set = featuresets
  classifier = nltk.NaiveBayesClassifier.train(training_set)

  fw = open(fileName,"w")
  fw.write("PhraseId"+','+"Sentiment"+'\n')
  for test,id in test_featuresets:
    fw.write(str(id)+','+str(classifier.classify(test))+'\n')
  fw.close()
```

The exact code that was modified within the *kaggleClassify.py* file can be seen in the screenshot below including a description of exactly what some of the code does.

```
## repeat the setup of the movie review sentences for classification
# for each sentence(document), get its words and category (positive/negative)
documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]

# get all words from all movie_reviews and put into a frequency distribution
#   note lowercase, but no stemming or stopwords
all_words_list = [word for (sent,cat) in documents for word in sent]
all_words = nltk.FreqDist(all_words_list)
print(len(all_words))

# get the 1500 most frequently appearing keywords in the corpus
word_items = all_words.most_common(1500)
word_features = [word for (word,count) in word_items]

# define features (keywords) of a document for a BOW/unigram baseline
# each feature is 'contains(keyword)' and is true or false depending
# on whether that keyword is in the document
def document_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['V_{}'.format(word)] = (word in document_words)
    return features

# define a feature definition function here
# get features sets for a document, including keyword features and category feature
featuresets = [(document_features(d, word_features), c) for (d, c) in documents]

# training using naive Baysian classifier, training set is 90% of data
train_set, test_set = featuresets[1000:], featuresets[:1000]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print('Overall Accuracy', nltk.classify.accuracy(classifier, test_set))

# use NLTK to compute evaluation measures from a reflist of gold labels
#    and a testlist of predicted labels for all labels in a list
# returns lists of precision and recall for each label
goldlist = []
predictedlist = []
for (features, label) in test_set:
  goldlist.append(label)
  predictedlist.append(classifier.classify(features))

cm = nltk.ConfusionMatrix(goldlist, predictedlist)

print(cm.pretty_format(sort_by_count=True, show_percents=False, truncate = 9))
```

Results

Once updated, this code was saved and the entire file was run using the same code that was entered into the command line before and the output was the result of the frequency distribution of word counts, the overall accuracy of the random sampling seed that we set, a confusion matrix of the

negative and positive words (rows being reference and columns being test), and then again reading 156,060 phrases and using 10,000 random phrases. The output code can be seen in the screenshot below. Our baseline overall accuracy was ***0.56,*** with a precision of 0.56, a recall of 0.54, and an F1 of 0.53.

```
Baseline accuracy:
Training and testing a classifier
Accuracy of classifier :
0.5606666666666666
--------------------------------------------------
Showing most informative features
Most Informative Features
                 V_both = True           negati : neutra =     13.5 : 1.0
                V_great = True           negati : neutra =     13.5 : 1.0
                V_funny = True           negati : neutra =     12.4 : 1.0
          V_interesting = True           negati : neutra =     12.2 : 1.0
                  V_An = True           negati : neutra =     11.2 : 1.0
None
--------------------------------------------------
         |         n    p |
         |    n    e    o |
         |    e    g    s |
         |    u    a    i |
         |    t    t    t |
         |    r    i    i |
         |    a    v    v |
         |    l    e    e |
---------+----------------+
 neutral |<2485> 267  297 |
negative |  910 <448> 236 |
positive |  702  224 <431>|
---------+----------------+
(row = reference; col = test)

Each fold size: 2000
Fold 0
Fold 1
Fold 2
Fold 3
Fold 4

Average Precision       Recall          F1      Per Label
neutral               0.812      0.608      0.695
positive              0.343      0.447      0.388
negative              0.275      0.500      0.355

Macro Average Precision Recall          F1      Over All Labels
              0.477      0.518      0.479

Label Counts {'neutral': 5079, 'positive': 2277, 'negative': 2644}
Micro Average Precision Recall          F1      Over All Labels
              0.563      0.543      0.535
```

Although it had performed well in initial runs with smaller sample sizes, the preprocessing functions implemented actually decreased the model's overall accuracy, as well as precision and recall. Some adjustments such as reducing the stop word list, and deleting the phrase length filter were attempted with slightly improved results, however the initial preprocessing function results are attached below:

```
Preprocessed accuracy:
Training and testing a classifier
Accuracy of classifier :
0.5438333333333333
--------------------------------------------------
Showing most informative features
Most Informative Features
                 V_dull = True           positi : neutra =     14.0 : 1.0
                V_great = True           negati : neutra =     13.5 : 1.0
                V_funny = True           negati : neutra =     13.0 : 1.0
           V_interesting = True           negati : neutra =     12.2 : 1.0
                 V_worth = True           negati : neutra =     11.0 : 1.0
None
--------------------------------------------------
         |         n    p |
         |    n    e    o |
         |    e    g    s |
         |    u    a    i |
         |    t    t    t |
         |    r    i    i |
         |    a    v    v |
         |    l    e    e |
---------+----------------+
 neutral |<2603> 246  200 |
negative |  1064 <416> 114 |
positive |   939  174 <244>|
---------+----------------+
(row = reference; col = test)

Each fold size: 2000
Fold 0
Fold 1
Fold 2
Fold 3
Fold 4

Average Precision       Recall          F1      Per Label
neutral               0.856      0.571      0.685
positive              0.205      0.453      0.282
negative              0.269      0.524      0.355

Macro Average Precision Recall          F1      Over All Labels
              0.443      0.516      0.441

Label Counts {'neutral': 5079, 'positive': 2277, 'negative': 2644}
Micro Average Precision Recall          F1      Over All Labels
              0.553      0.532      0.506
```

The subjectivity clues lexicon provided the greatest accuracy improvement of any featureset tested, at 0.591. It's standing as the best featureset held consistent through various runs with different phrase limit sizes, even before the randomization was controlled. This featureset also boosted precision and recall.

```
Subjectivity Lexicon accuracy:
Training and testing a classifier
Accuracy of classifier :
0.591
--------------------------------------------------
Showing most informative features
Most Informative Features
          contains(dull) = True           positi : neutra =     14.0 : 1.0
         contains(great) = True           negati : neutra =     13.5 : 1.0
         contains(funny) = True           negati : neutra =     13.0 : 1.0
   contains(interesting) = True           negati : neutra =     12.2 : 1.0
        contains(moving) = True           negati : neutra =     11.0 : 1.0
None
--------------------------------------------------
         |         n    p |
         |    n    e    o |
         |    e    g    s |
         |    u    a    i |
         |    t    t    t |
         |    r    i    i |
         |    a    v    v |
         |    l    e    e |
---------+----------------+
 neutral |<2490> 283  276 |
negative |  794 <670> 130 |
positive |  761  210 <386>|
---------+----------------+
(row = reference; col = test)

Each fold size: 2000
Fold 0
Fold 1
Fold 2
Fold 3
Fold 4

Average Precision       Recall          F1      Per Label
neutral               0.810      0.619      0.702
positive              0.293      0.501      0.369
negative              0.443      0.579      0.502

Macro Average Precision Recall          F1      Over All Labels
              0.515      0.566      0.524

Label Counts {'neutral': 5079, 'positive': 2277, 'negative': 2644}
Micro Average Precision Recall          F1      Over All Labels
              0.595      0.582      0.573
```

The negation word featureset performed slightly worse than the base set, though it is an improvement on the preprocessed featureset, which actually feeds into this one. Comparatively, it provides an improvement of about 1%, and similar improvements to precision and recall.

```
Negation word accuracy:
Training and testing a classifier
Accuracy of classifier :
0.5541666666666667
--------------------------------------------------
Showing most informative features
Most Informative Features
                V_worst = False          positi : neutra =     18.4 : 1.0
                 V_dull = True           positi : neutra =     14.0 : 1.0
                V_great = True           negati : neutra =     13.5 : 1.0
                V_funny = True           negati : neutra =     13.0 : 1.0
          V_interesting = True           negati : neutra =     12.2 : 1.0
None
--------------------------------------------------
         |         n    p |
         |    n    e    o |
         |    e    g    s |
         |    u    a    i |
         |    t    t    t |
         |    r    i    i |
         |    a    v    v |
         |    l    e    e |
---------+----------------+
 neutral |<1653> 671  725 |
negative |  427 <901> 266 |
positive |  364  222 <771>|
---------+----------------+
(row = reference; col = test)

Each fold size: 2000
Fold 0
Fold 1
Fold 2
Fold 3
Fold 4

Average Precision       Recall          F1      Per Label
neutral               0.499      0.708      0.585
positive              0.640      0.446      0.526
negative              0.613      0.514      0.559

Macro Average Precision Recall          F1      Over All Labels
              0.584      0.556      0.557

Label Counts {'neutral': 5079, 'positive': 2277, 'negative': 2644}
Micro Average Precision Recall          F1      Over All Labels
              0.561      0.597      0.565
```

The bigram featureset performs only as well as the preprocessed featureset that feeds into it. One can see below that none of the five most important features are bigrams. This list would have to be expanded very far before a bigram feature was actually included.

```
Bigram features accuracy:
Training and testing a classifier
Accuracy of classifier :
0.5438333333333333
--------------------------------------------------
Showing most informative features
Most Informative Features
          contains(dull) = True           positi : neutra =     14.0 : 1.0
         contains(great) = True           negati : neutra =     13.5 : 1.0
         contains(funny) = True           negati : neutra =     13.0 : 1.0
   contains(interesting) = True           negati : neutra =     12.2 : 1.0
        contains(moving) = True           negati : neutra =     11.0 : 1.0
None
--------------------------------------------------
         |         n    p |
         |    n    e    o |
         |    e    g    s |
         |    u    a    i |
         |    t    t    t |
         |    r    i    i |
         |    a    v    v |
         |    l    e    e |
---------+----------------+
 neutral |<2603> 246  200 |
negative | 1064 <416> 114 |
positive |  939  174 <244>|
---------+----------------+
(row = reference; col = test)

Each fold size: 2000
Fold 0
Fold 1
Fold 2
Fold 3
Fold 4

Average Precision       Recall          F1      Per Label
neutral               0.856      0.571      0.685
positive              0.205      0.453      0.282
negative              0.269      0.524      0.355

Macro Average Precision Recall          F1      Over All Labels
              0.443      0.516      0.441

Label Counts {'neutral': 5079, 'positive': 2277, 'negative': 2644}
Micro Average Precision Recall          F1      Over All Labels
              0.553      0.532      0.506
```

The part of speech featureset is a slight improvement on the preprocessed featureset, with an accuracy and precision of 0.56 each. Recall barely improves, implying that false negatives are still rampant in each category.

```
POS features accuracy:
Training and testing a classifier
Accuracy of classifier :
0.557
--------------------------------------------------
Showing most informative features
Most Informative Features
          contains(dull) = True           positi : neutra =     14.0 : 1.0
         contains(great) = True           negati : neutra =     13.5 : 1.0
         contains(funny) = True           negati : neutra =     13.0 : 1.0
   contains(interesting) = True           negati : neutra =     12.2 : 1.0
        contains(moving) = True           negati : neutra =     11.0 : 1.0
None
--------------------------------------------------
         |         n    p |
         |    n    e    o |
         |    e    g    s |
         |    u    a    i |
         |    t    t    t |
         |    r    i    i |
         |    a    v    v |
         |    l    e    e |
---------+----------------+
 neutral |<2459> 292  298 |
negative |  879 <493> 222 |
positive |  720  247 <390>|
---------+----------------+
(row = reference; col = test)

Each fold size: 2000
Fold 0
Fold 1
Fold 2
Fold 3
Fold 4

Average Precision       Recall          F1      Per Label
neutral               0.811      0.608      0.695
positive              0.309      0.428      0.359
negative              0.295      0.495      0.369

Macro Average Precision Recall          F1      Over All Labels
              0.472      0.510      0.474

Label Counts {'neutral': 5079, 'positive': 2277, 'negative': 2644}
Micro Average Precision Recall          F1      Over All Labels
              0.560      0.537      0.532
```

The LIWC featureset gave the second most improved results, after the subjectivity clues. It improved accuracy and precision to 0.58, and recall to 0.57. One can still see that the most important features are simply word features, but clearly the sentiment scores were a useful factor.

```
LIWC features accuracy:
Training and testing a classifier
Accuracy of classifier :
0.5766666666666667
--------------------------------------------------
Showing most informative features
Most Informative Features
          contains(dull) = True           positi : neutra =     14.0 : 1.0
         contains(great) = True           negati : neutra =     13.5 : 1.0
         contains(funny) = True           negati : neutra =     13.0 : 1.0
   contains(interesting) = True           negati : neutra =     12.2 : 1.0
        contains(moving) = True           negati : neutra =     11.0 : 1.0
None
--------------------------------------------------
         |         n    p |
         |    n    e    o |
         |    e    g    s |
         |    u    a    i |
         |    t    t    t |
         |    r    i    i |
         |    a    v    v |
         |    l    e    e |
---------+----------------+
 neutral |<2524> 313  212 |
negative |  868 <615> 111 |
positive |  822  214 <321>|
---------+----------------+
(row = reference; col = test)

Each fold size: 2000
Fold 0
Fold 1
Fold 2
Fold 3
Fold 4

Average Precision       Recall          F1      Per Label
neutral               0.827      0.607      0.700
positive              0.259      0.501      0.341
negative              0.394      0.549      0.459

Macro Average Precision Recall          F1      Over All Labels
              0.494      0.552      0.500

Label Counts {'neutral': 5079, 'positive': 2277, 'negative': 2644}
Micro Average Precision Recall          F1      Over All Labels
              0.583      0.568      0.555
```

Because the subjectivity lexicon and LIWC featuresets provided the best improvement, it seemed fitting to create a hybrid featureset that would combine the two lexicons to cover a broader span of words. Unfortunately, the LIWC features seem to actually drag the subjectivity lexicon down, as the accuracy of 0.589 is just slightly worse than the subjectivity lexicon featureset obtained on its own. There also very slight decreases to precision and recall.

```
hybrid sentiment features accuracy:
Training and testing a classifier
Accuracy of classifier :
0.5886666666666667
--------------------------------------------------
Showing most informative features
Most Informative Features
          contains(dull) = True           positi : neutra =     14.0 : 1.0
         contains(great) = True           negati : neutra =     13.5 : 1.0
         contains(funny) = True           negati : neutra =     13.0 : 1.0
   contains(interesting) = True           negati : neutra =     12.2 : 1.0
        contains(moving) = True           negati : neutra =     11.0 : 1.0
None
--------------------------------------------------
         |         n    p |
         |    n    e    o |
         |    e    g    s |
         |    u    a    i |
         |    t    t    t |
         |    r    i    i |
         |    a    v    v |
         |    l    e    e |
---------+----------------+
 neutral |<2448> 322  279 |
negative |  768 <683> 143 |
positive |  739  217 <401>|
---------+----------------+
(row = reference; col = test)

Each fold size: 2000
Fold 0
Fold 1
Fold 2
Fold 3
Fold 4

Average Precision       Recall          F1      Per Label
neutral              0.797      0.623      0.699
positive             0.311      0.503      0.384
negative             0.445      0.565      0.498

Macro Average Precision Recall          F1      Over All Labels
              0.518      0.563      0.527

Label Counts {'neutral': 5079, 'positive': 2277, 'negative': 2644}
Micro Average Precision Recall          F1      Over All Labels
              0.593      0.580      0.574
```

Responsibilities
Joint: Pre-processing,

N'Dea: Structuring the report, took on the brunt of report writing, Cross-validation, Confusion matrices, random sampling seeds,

Michael: Most of the featuresets, structuring the code, generating the test output file, most of the experimenting with feature variations.