NATURAL LANGUAGE PROCESSING FINAL PROJECT

Kaggle Movie Review

Achal Velani
Rohit Sharma

Contents

1. The Dataset:

The dataset which we are using was produced for the Kaggle competition, as described here in the link:

https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews.

The data was taken from the original Pang and Lee movie review corpus based on reviews from

the Rotten Tomatoes web site. Socher's group used crowd-sourcing to manually annotate all the

subphrases of sentences with a sentiment label ranging over:

0 - "negative",

1 - "somewhat negative",

2 - "neutral",

3 - "somewhat positive",

4 - "positive".

There are two data files provided in the data set which are of relevance to us:

train.tsv – This data contains phrases with sentence ids and their associated sentiment labels. Can be divided into train and test splits and can be used for training and testing purpose both.

test.tsv – This data contains only phrases which are to be classified i.e. we need to identify and assign sentiments to each of the phrases in the dataset.

2. Steps taken for the sentiment classification:

a. Read the data from train.tsv file :

First, we read the data from train.tsv and ignore "PhraseID" and "SentenceID" because they have no significance in the sentiment analysis. We loop over the lines in the file one by one ignoring the first line which starts with 'Phrase'. Each line has four fields which are separated with a 'tab'. So, at this step, we ignore the PhraseId and SentenceId and keep phrase and sentiment.

Moreover, as we have 150K data lines we are not using the whole dataset every time, we have a variable called limit populated from limitStr passed to the function from a command line parameter. We use this parameter to limit the used data lines.

Code:

```
if __name__ == '__main__':
  if (len(sys.argv) != 3):
    print ('usage: classifyKaggle.py <corpus-dir> <limit>')
    sys.exit(0)
  processkaggle(sys.argv[1], sys.argv[2])


def processkaggle(dirPath,limitStr):
  # convert the limit argument from a string to an int
  limit = int(limitStr)
  os.chdir(dirPath)
  f = open('./train.tsv', 'r')
  # loop over lines in the file and use the first limit of them
  phrasedata = []

  for line in f:
    # ignore the first line starting with Phrase and read all lines
    if (not line.startswith('Phrase')):
      # remove final end of line character
      line = line.strip()
      # each line has 4 items separated by tabs
      # ignore the phrase and sentence ids, and keep the phrase and sentiment
      phrasedata.append(line.split('\t')[2:4])
  # pick a random sample of length limit because of phrase overlapping sequences
  random.shuffle(phrasedata)
  phraselist = phrasedata[:limit]
```

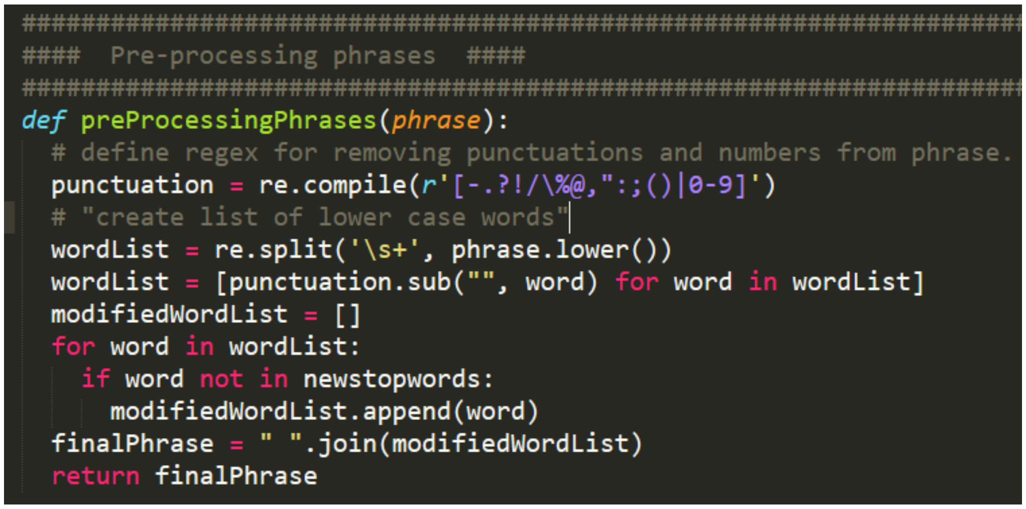
b. Preprocessing and Tokenization:

1. We have maintained two document list phrasedocs and processedPhraseDocs. For unprocessed document list we have used inbuilt word_tokenizer but for preprocessed one we have used a RegexpTokenizer which will get rid of all the punctuations while tokenizing the document lines. We are also maintaining the sentiment along with the document, reason being it could be used while creating test and train data sets.

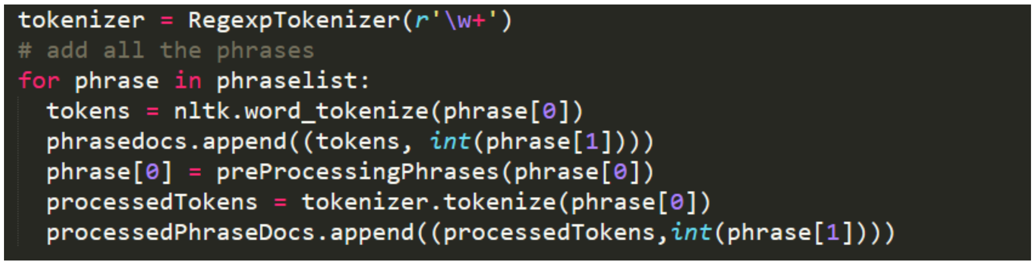
2. We have created a function called preProcessingPhrases which will take each phrase in the document one by one and do some preprocessing steps on it. First, the method compiles a regex to remove all the punctuation marks and digits from the phrase. Then, we convert the whole phrase in lowercase letters and split it by space to make a wordlist. Now, we apply the regex on this newly formed list to remove the undesired words. Additionally, we are using a new stopwords list in addition to the default English stopwords list provided by nltk. The new stopwords list is as below (we have actually removed few words that we needed from the original stopwords list):

```
stopwords = nltk.corpus.stopwords.words('english')
newstopwords = [word for word in stopwords if word not in
['do','does','did','doing','t','can','don','aren','couldn','didn','doesn','hadn','hasn','haven'
,'isn','mightn','needn','shouldn','shan','wasn','weren','won','wouldn','should','have','has',
'had','having']]
```

Now, the method will create and return a new modified phrase skipping the stopwords and joining the words in the modified word list by space.



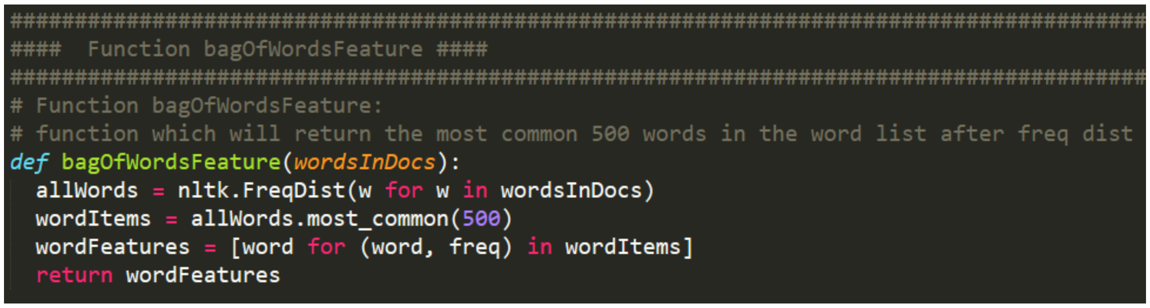
```
####################################################################
####  Pre-processing phrases  ####
####################################################################
def preProcessingPhrases(phrase):
  # define regex for removing punctuations and numbers from phrase.
  punctuation = re.compile(r'[-.?!/\%@,":;()|0-9]')
  # "create list of lower case words"
  wordList = re.split('\s+', phrase.lower())
  wordList = [punctuation.sub("", word) for word in wordList]
  modifiedWordList = []
  for word in wordList:
    if word not in newstopwords:
      modifiedWordList.append(word)
  finalPhrase = " ".join(modifiedWordList)
  return finalPhrase
```



```
tokenizer = RegexpTokenizer(r'\w+')
# add all the phrases
for phrase in phraselist:
  tokens = nltk.word_tokenize(phrase[0])
  phrasedocs.append((tokens, int(phrase[1])))
  phrase[0] = preProcessingPhrases(phrase[0])
  processedTokens = tokenizer.tokenize(phrase[0])
  processedPhraseDocs.append((processedTokens,int(phrase[1])))
```

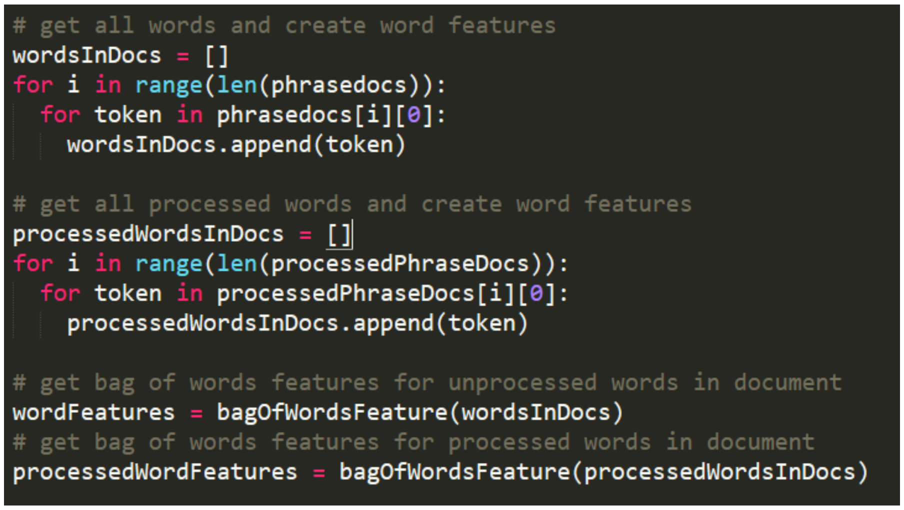
c. Feature Selection:

We used bag of words features to be used with our different feature function. We produced bag of words features for unprocessed and processed data both. For this, we created two lists wordInDoc (which is the list of all the words in all the phrases in the document without any preprocessing) and processedWordsInDocs (which is the list of all the words in all the phrases in the document after preprocessing). We used 500 most frequent words in bag of words feature. This will give us a sparse matrix type of data which we will use in training of the classifier. We did experiments with different values which we will discuss in experiments section. Below is the function definition for bag-of-words.



```
#########################################################################################
####  Function bagOfWordsFeature ####
#########################################################################################
# Function bagOfWordsFeature:
# function which will return the most common 500 words in the word list after freq dist
def bagOfWordsFeature(wordsInDocs):
  allWords = nltk.FreqDist(w for w in wordsInDocs)
  wordItems = allWords.most_common(500)
  wordFeatures = [word for (word, freq) in wordItems]
  return wordFeatures
```

Below is how we used bag-of-words function



```
# get all words and create word features
wordsInDocs = []
for i in range(len(phrasedocs)):
  for token in phrasedocs[i][0]:
    wordsInDocs.append(token)

# get all processed words and create word features
processedWordsInDocs = []
for i in range(len(processedPhraseDocs)):
  for token in processedPhraseDocs[i][0]:
    processedWordsInDocs.append(token)

# get bag of words features for unprocessed words in document
wordFeatures = bagOfWordsFeature(wordsInDocs)
# get bag of words features for processed words in document
processedWordFeatures = bagOfWordsFeature(processedWordsInDocs)
```

d. Producing feature functions:

Unigram features:

Unigram feature acts as the baseline feature function. This is the most basic feature function. The idea is to take each word from the wordFeatures as obtained from the bag-of-words method and check if they exist in the document or not. The feature label will be named as 'contains(keyword)' for each word in the word features set, and the value of the feature will be boolean, i.e. True if the word exists in the document and False if it does not exist.

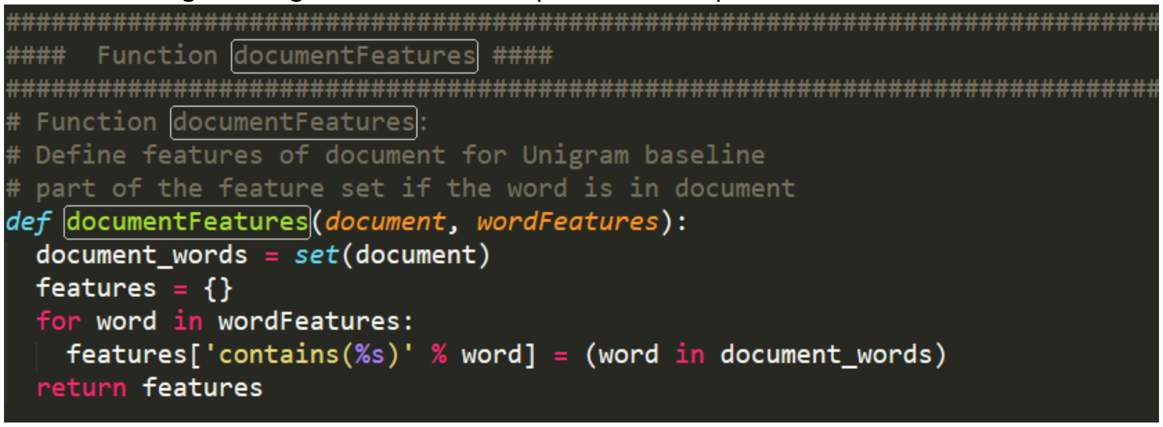
Examples of Unigram features set for unprocessed data:

({'contains(by)': True, "contains('s)": False, 'contains(the)': True, 'contains(.)': False, 'contains(on)': False, 'contains(into)': False, 'contains(a)': False, 'contains(lesson)': False, 'contains(history)': False, 'contains(high)': False, 'contains(grips)': False, 'contains(Rollerball)': False, 'contains(sustains)': False, 'contains(note)': False, 'contains(it)': False, 'contains(serious)': False, 'contains(``)': False, 'contains(put)': False, 'contains(year)': False, 'contains(last)': True, 'contains(debt)': False, 'contains(to)': False, 'contains(suspense)': False, 'contains(struggle)': False, 'contains(infamy)': False, 'contains(begins)': False, 'contains(15-year)': False, 'contains(...)': False, 'contains(beautifully)': False, 'contains(historical)': False, 'contains(and)': False, 'contains(meaningful)': False, 'contains(one)': True, 'contains(context)': False}, 2)

Examples of Unigram features set for processed data:

({'contains(lesson)': False, 'contains(struggle)': False, 'contains(year)': False, 'contains(s)': False, 'contains(grips)': False, 'contains(note)': False, 'contains(serious)': False, 'contains(put)': False, 'contains(high)': False, 'contains(last)': True, 'contains(debt)': False, 'contains(sustains)': False, 'contains(suspense)': False, 'contains(rollerball)': False, 'contains(history)': False, 'contains(infamy)': False, 'contains(begins)': False, 'contains(beautifully)': False, 'contains(historical)': False, 'contains(meaningful)': False, 'contains(one)': True, 'contains(context)': False}, 2)

Code for finding the Unigram features for unprocessed and processed data both is as below:



```
##############################################################################
####  Function documentFeatures ####
##############################################################################
# Function documentFeatures:
# Define features of document for Unigram baseline
# part of the feature set if the word is in document
def documentFeatures(document, wordFeatures):
  document_words = set(document)
  features = {}
  for word in wordFeatures:
    features['contains(%s)' % word] = (word in document_words)
  return features
```

SL Features (Sentiment Lexicon – Subjectivity count):

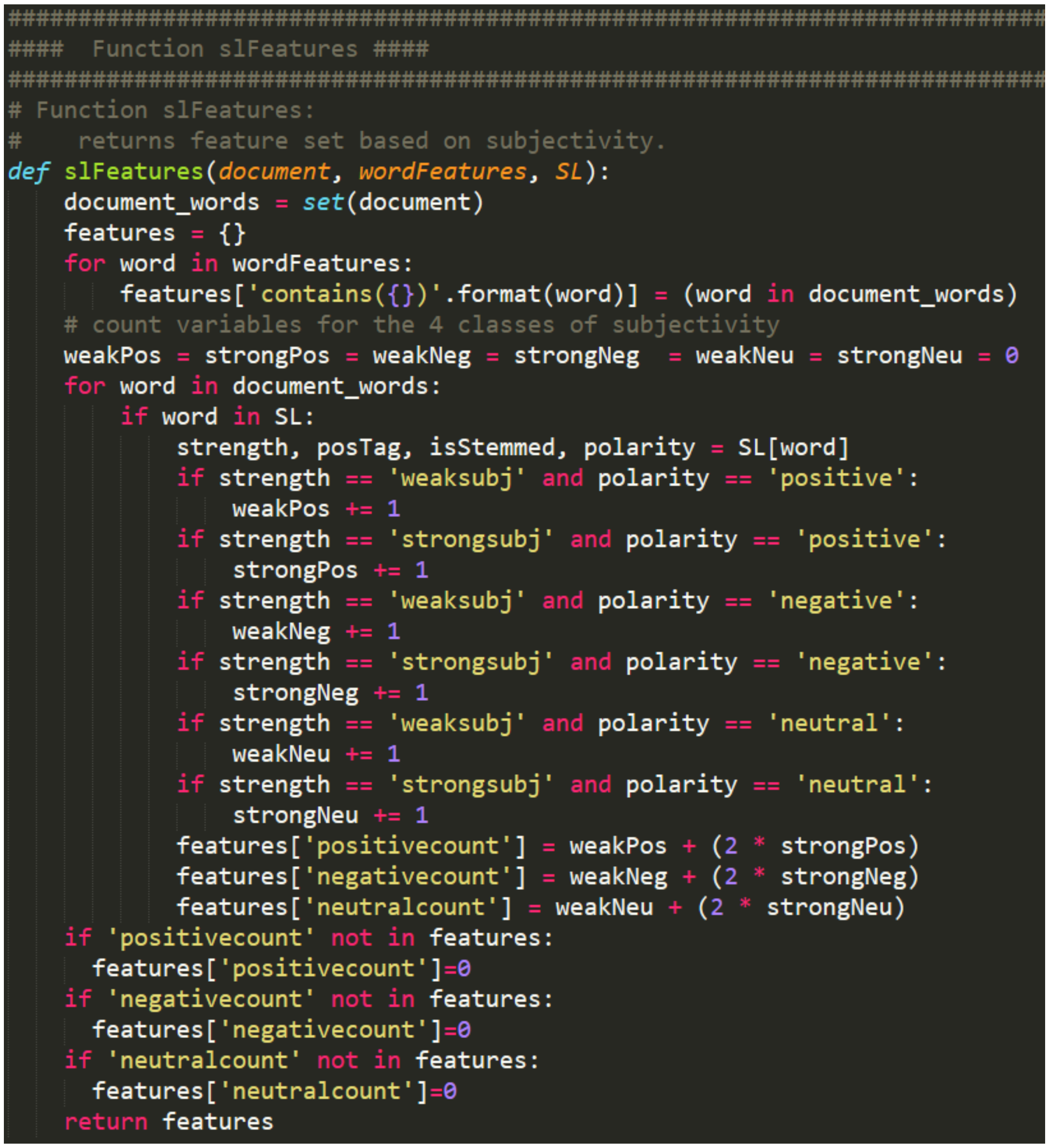
To find the features based on Sentiment Lexicon, first we need to import the sentiment_read_subjectivity.py python file provided as part of project code. Also, there is subjclueslen1-HLTEMNLP05.tff file provided as part of the project code which contains data in the format:

type=weaksubj len=1 word1=abandoned pos1=adj stemmed1=n priorpolarity=negative

this provides strength, posTag, isStemmed, polarity of the words. Using this we can find the sentiment of a phrase based on the subjective polarity of each of the words in that phrase.

In this feature set, we use the readSubjectivity method provided by sentiment_read_subjectivity.py python file which returns a SL (Sentiment List) containing strength, posTag, isStemmed, polarity for each word.

Now, we use this SL list to make a count of positive, negative, and neutral words in the phrase, this way positivecount, negativecount, and neutralcount becomes a feature for each phrase in the document in addition to the already existing ones. Below is the code for extracting SL features. These features contain counts of all the positive, negative, and neutral subjectivity words, where we have counted each weak subjective word once and each strong subjective word twice.



```
##########################################################################
####  Function slFeatures ####
##########################################################################
# Function slFeatures:
#    returns feature set based on subjectivity.
def slFeatures(document, wordFeatures, SL):
    document_words = set(document)
    features = {}
    for word in wordFeatures:
        features['contains({})'.format(word)] = (word in document_words)
    # count variables for the 4 classes of subjectivity
    weakPos = strongPos = weakNeg = strongNeg  = weakNeu = strongNeu = 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
            if strength == 'weaksubj' and polarity == 'neutral':
                weakNeu += 1
            if strength == 'strongsubj' and polarity == 'neutral':
                strongNeu += 1
            features['positivecount'] = weakPos + (2 * strongPos)
            features['negativecount'] = weakNeg + (2 * strongNeg)
            features['neutralcount'] = weakNeu + (2 * strongNeu)
    if 'positivecount' not in features:
      features['positivecount']=0
    if 'negativecount' not in features:
      features['negativecount']=0
    if 'neutralcount' not in features:
      features['neutralcount']=0
    return features
```

NOT Features (Negation feature function):

If a word follows a negation word then its meaning changes, hence, negation of an opinion becomes an important thing to consider while doing the sentiment classification over any dataset. Here, the idea is to negate the meaning of the word following the negation word. In other strategies, we can also negate all the words after a negation word till next punctuation mark, or semantics analysis can be done to find the scope of the negation word. We progressed with the simplest strategy in our approach and negated the word following the negation word.

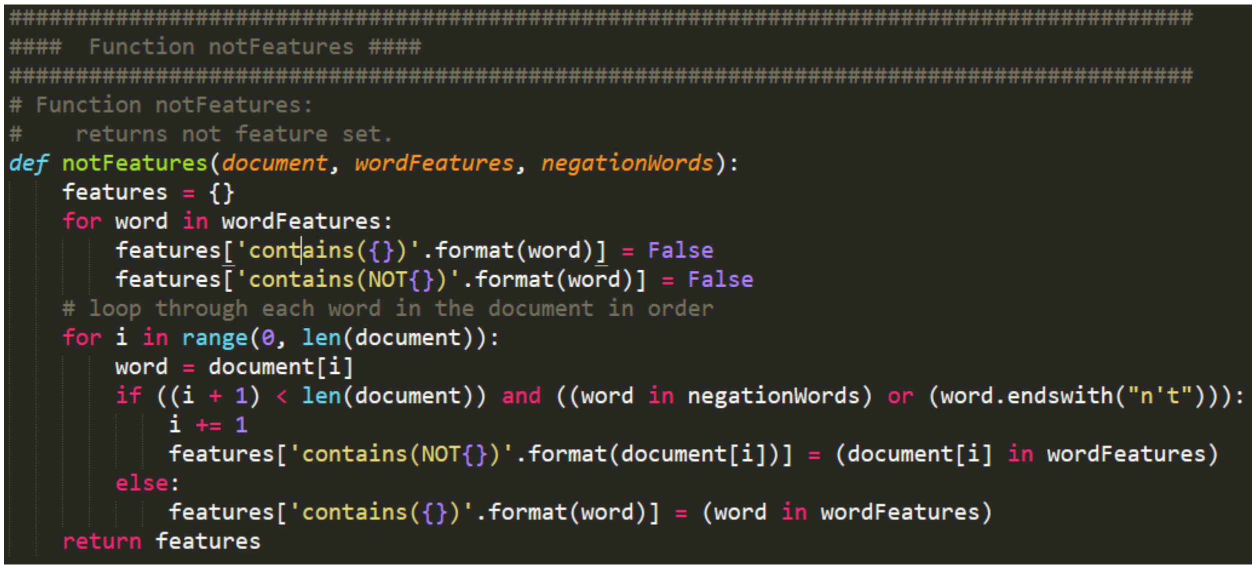
There are phrases in the dataset like: A puzzle whose pieces do not fit.

Example of NOT Feature Set unprocessed:

({'contains(NOTgroundbreaking)': False, 'contains(NOTThis)': False, 'contains(bold)': False, 'contains(NOTso)': False, 'contains(NOTsimply)': False, 'contains(NOTmovies)': False, 'contains(NOTcolors)': False, 'contains(NOTof)': False, 'contains(dope)': False, 'contains(NOTto)': False, 'contains(sugar)': False, 'contains(NOTdope)': False, 'contains(story)': False,
…
'contains(NOTlove)': False, 'contains(NOTits)': False, 'contains(NOTan)': False, 'contains(simply)': False, 'contains(NOThat)': False}, 2)

Example of NOT Feature set processed:

({'contains(NOTgroundbreaking)': False, 'contains(world)': False, 'contains(NOTtime)': False, 'contains(NOTdate)': False, 'contains(NOTmovies)': False, 'contains(dope)': False, 'contains(NOTcorny)': False, 'contains(NOTauteuil)': False, 'contains(seamy)': False, 'contains(NOTdope)': False, 'contains(story)': False, 'contains(writers)': False, 'contains(auteuil)':
…
'contains(NOTothers)': False, 'contains(NOThat)': False}, 2)


```
##########################################################################################
####  Function notFeatures ####
##########################################################################################
# Function notFeatures:
#    returns not feature set.
def notFeatures(document, wordFeatures, negationWords):
    features = {}
    for word in wordFeatures:
        features['contains({})'.format(word)] = False
        features['contains(NOT{})'.format(word)] = False
    # loop through each word in the document 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['contains(NOT{})'.format(document[i])] = (document[i] in wordFeatures)
        else:
            features['contains({})'.format(word)] = (word in wordFeatures)
    return features
```

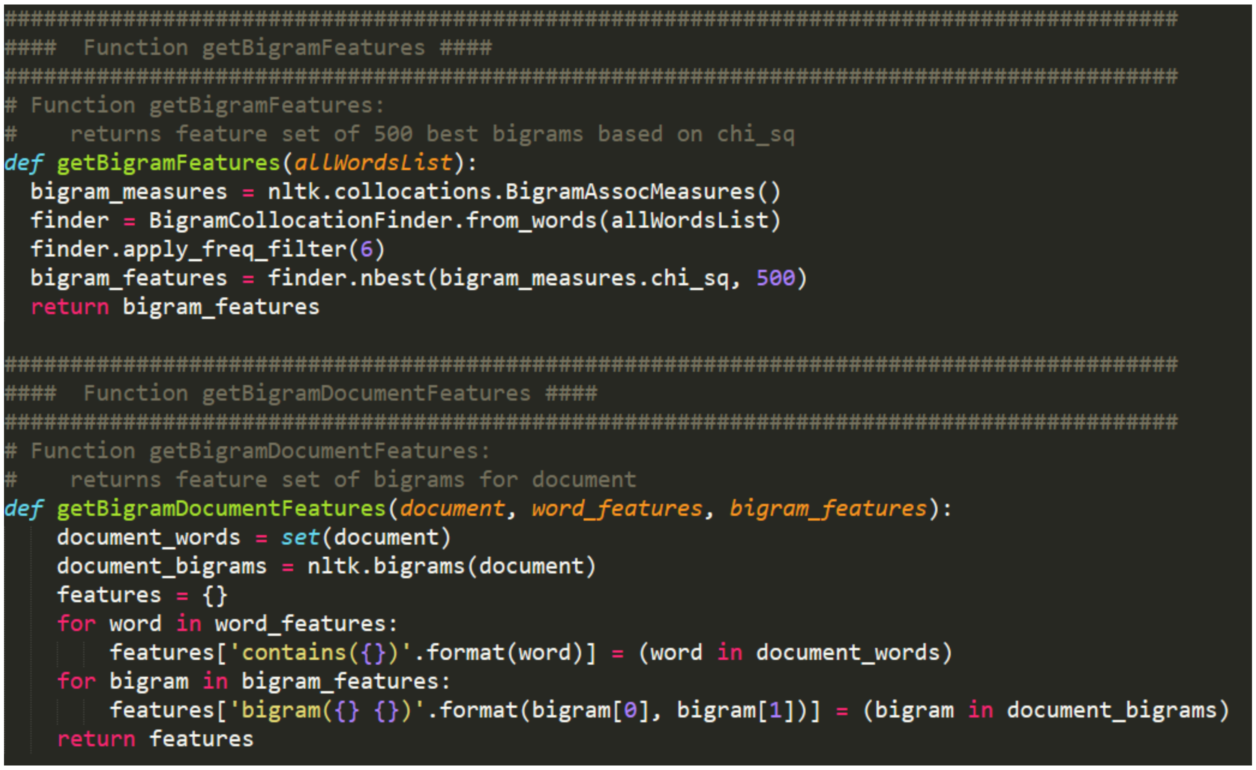
List of negation words used is:

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

Bigram Features:

In addition to generating the Unigram features, we have also written a feature function to generate Bigram features which can be used to generate bigram features for both processed and unprocessed data. We have used the code as taught by professor in the lab to get the bigram measure and to create a bigram collocation finder to run on the data. To increase the accuracy of bigram feature function, we used frequency filter. After experimenting with different frequencies, we found the best accuracy with frequency of 6. Finally, we retrieved 500 highest scoring bigrams based on chi_sq method using nbest function provided by bigram collocation finder.

Below is the code to find the bigram feature sets:

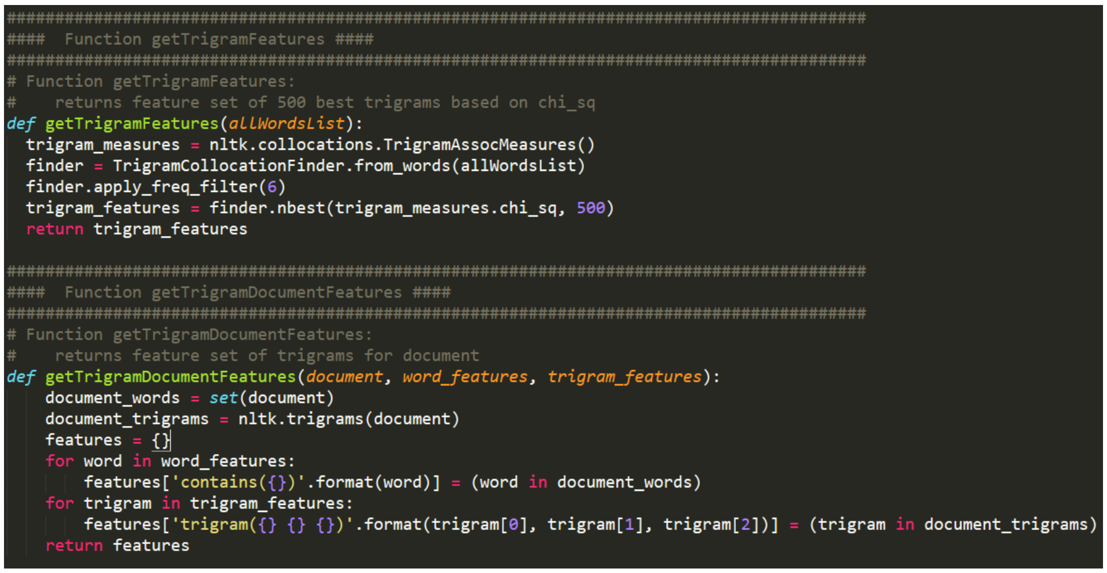


```
#########################################################################################
####  Function getBigramFeatures ####
#########################################################################################
# Function getBigramFeatures:
#    returns feature set of 500 best bigrams based on chi_sq
def getBigramFeatures(allWordsList):
  bigram_measures = nltk.collocations.BigramAssocMeasures()
  finder = BigramCollocationFinder.from_words(allWordsList)
  finder.apply_freq_filter(6)
  bigram_features = finder.nbest(bigram_measures.chi_sq, 500)
  return bigram_features

#########################################################################################
####  Function getBigramDocumentFeatures ####
#########################################################################################
# Function getBigramDocumentFeatures:
#    returns feature set of bigrams for document
def getBigramDocumentFeatures(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
```

Trigram Features:

Similar to Bigram features, we have experimented with Trigram features as well. We have used the code as taught by professor in the lab to get the trigram measure and to create a trigram collocation finder to run on the data. To increase the accuracy of trigram feature function, we used frequency filter. After experimenting with different frequencies, we found the best accuracy with frequency of 6. Finally, we retrieved 500 highest scoring trigrams based on chi_sq method using nbest function provided by trigram collocation finder. After experimenting we noticed that there is not much difference in accuracy of bigrams and trigrams.

Below is the code used to find trigram features from the given data.



```
##########################################################################################
####  Function getTrigramFeatures ####
##########################################################################################
# Function getTrigramFeatures:
#    returns feature set of 500 best trigrams based on chi_sq
def getTrigramFeatures(allWordsList):
  trigram_measures = nltk.collocations.TrigramAssocMeasures()
  finder = TrigramCollocationFinder.from_words(allWordsList)
  finder.apply_freq_filter(6)
  trigram_features = finder.nbest(trigram_measures.chi_sq, 500)
  return trigram_features

##########################################################################################
####  Function getTrigramDocumentFeatures ####
##########################################################################################
# Function getTrigramDocumentFeatures:
#    returns feature set of trigrams for document
def getTrigramDocumentFeatures(document, word_features, trigram_features):
    document_words = set(document)
    document_trigrams = nltk.trigrams(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    for trigram in trigram_features:
        features['trigram({} {} {})'.format(trigram[0], trigram[1], trigram[2])] = (trigram in document_trigrams)
    return features
```

LIWC Features:

LIWC, the Linguistic Inquiry and Word Count resource from James Pennebaker is the resource that provides list of positive and negative emotion words and methods to check them. For constructing LIWC features, we used the sentiment_read_LIWC_pos_neg_words.py python file provided with the project code. To implement this, first we import sentiment_read_LIWC_pos_neg_words in our classifyKaggle.py python code.

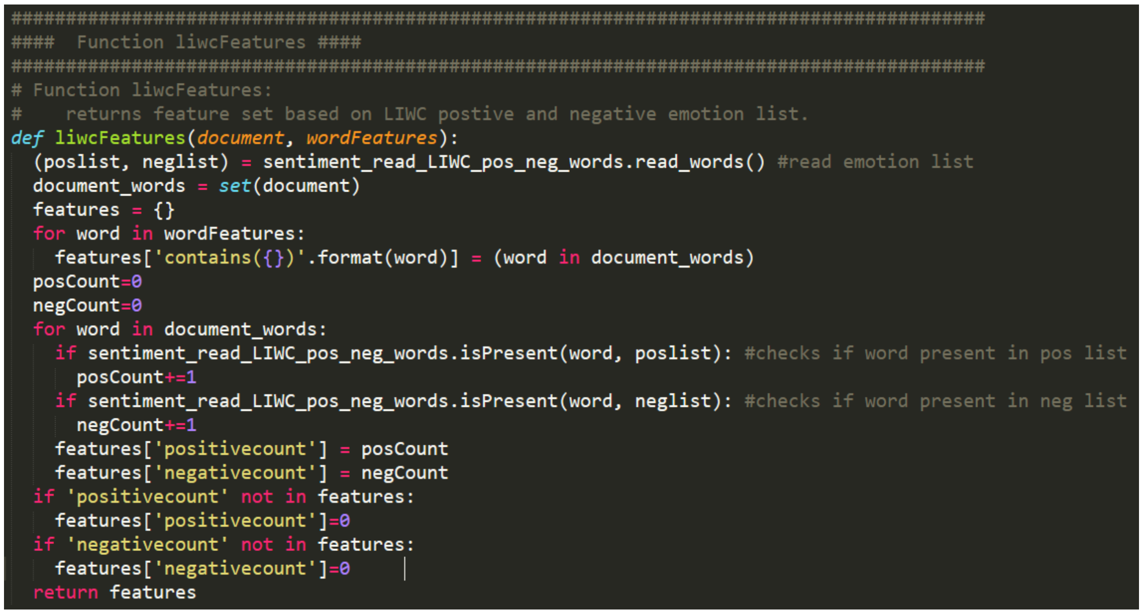
Following are the steps done in the method:

1. Use read_words() method from LIWC file to read the positive and negative word list in poslist and neglist respectively.
2. Then, in addition to contains feature, iterate over each word in the phrase and check if that word is present in the poslist or neglist provided by LIWC resource, use isPresent method for this. Increase positive words count (posCount) and negative words count (negCount) for all the words in the phrase according to the list they are in. This way, positivecount and negativecount will become one of the features with their values.

Examples of LIWC features for processed data:

({'contains(done)': False, 'contains(tonally)': False, 'contains(finish)': False, 'contains(sense)': False, 'contains(uneven)': False, 'contains(liability)': False, 'contains(surreal)': False, 'negativecount': 0, 'contains(severe)': False, 'contains(otherwise)': False, 'contains(personal)': False, 'contains(sluggish)': False, 'contains(music)': False, 'contains(underappreciated)': False, 'contains(terrible)': False, 'contains(go)': False, 'contains(film)': False, 'contains(s)': False, 'contains(work)': True, 'contains(smart)': False, 'contains(time)': False, 'contains(humor)': False, 'contains(documentary)': False, 'contains(material)': False, 'positivecount': 2, 'contains(value)': True, 'contains(loss)': False, 'contains(powerful)': False, 'contains(fans)': False, 'contains(might)': False, 'contains(kind)': False, 'contains(entertainment)': True, 'contains(unnoticed)': False, 'contains(sobering)': False, 'contains(much)': True, 'contains(really)': False, 'contains(technological)': False}, 2)

Below is the code I used to do this:



```
###########################################################################################
####  Function liwcFeatures ####
###########################################################################################
# Function liwcFeatures:
#    returns feature set based on LIWC postive and negative emotion list.
def liwcFeatures(document, wordFeatures):
  (poslist, neglist) = sentiment_read_LIWC_pos_neg_words.read_words() #read emotion list
  document_words = set(document)
  features = {}
  for word in wordFeatures:
    features['contains({})'.format(word)] = (word in document_words)
  posCount=0
  negCount=0
  for word in document_words:
    if sentiment_read_LIWC_pos_neg_words.isPresent(word, poslist): #checks if word present in pos list
      posCount+=1
    if sentiment_read_LIWC_pos_neg_words.isPresent(word, neglist): #checks if word present in neg list
      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
```

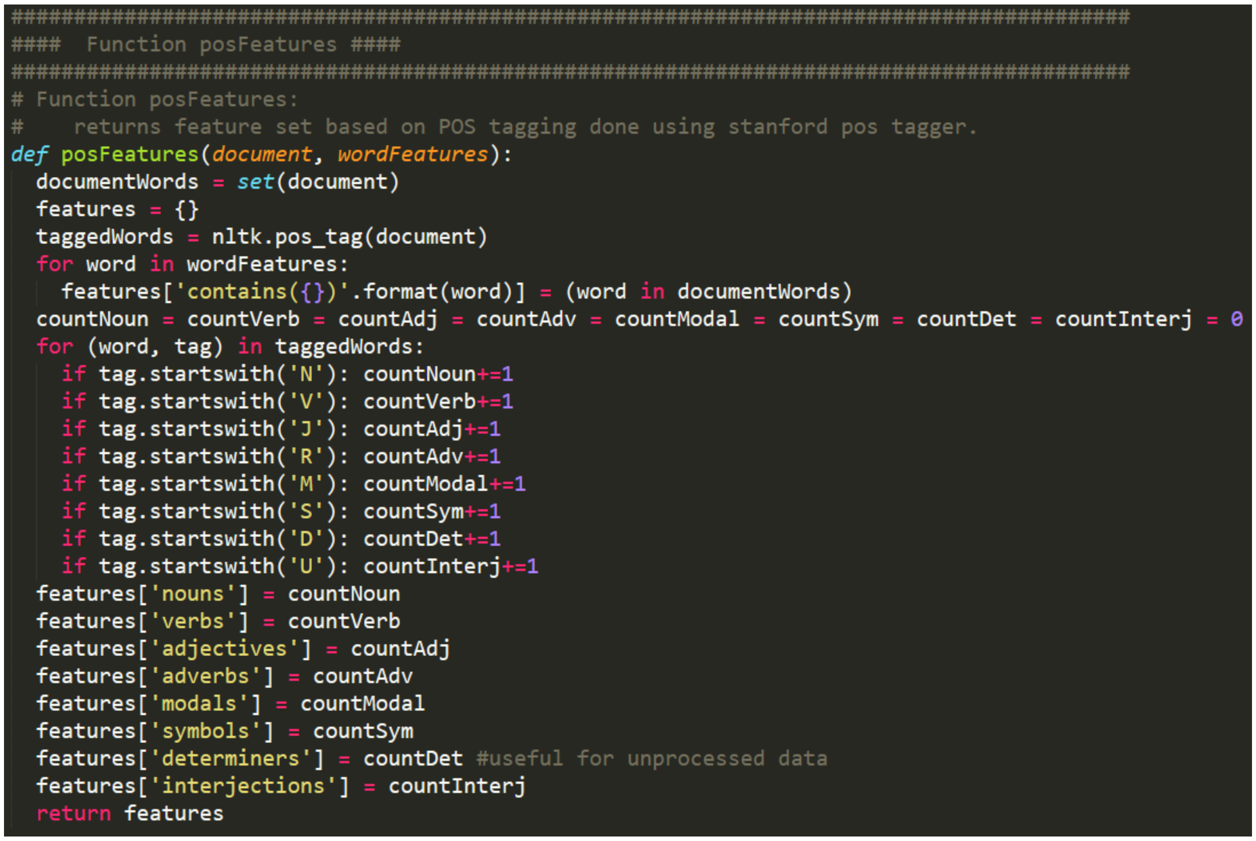
POS Features:

The most intuitive way to use POS tagging information is to include counts of various types of word tags. We have done this classification task with help of part-of-speech tag features using nltk default POS tagger which is basically Stanford POS tagger. POS tagging is most suitable for classification of shorter units like sentence level classification or shorter social media elements such as tweets. Since, default NLTK POS tagger takes too much time, this is difficult to demonstrate for whole data on large training dataset. The most common way to use POS tagging information is to include counts of various types of word tags. In this, using the tag conventions for Stanford POS tagger we have identified and counted important Parts of Speech tags. We have kept count of nouns, verbs, adjectives, adverbs, modals, symbols, determines, and interjections. The reason for keeping these many tags is that they are uniquely identifiable in a sentence if tagged and are most important parts of any English sentence.

Here is an example feature with POS tags for processed data:

({'contains(Macbeth)': False, 'contains(to)': False, 'contains(film)': False, 'contains(Shepard)': False, 'contains(too-long)': False, 'contains(journey)': False, "contains(n't)": False, 'contains(the)': True, 'contains(be)': False, 'contains(when)': False, 'contains(Matthew)': False, 'contains(of)': False, 'determiners': 2, 'contains(yet)': False, 'contains(in)': False, 'contains(certainly)': False, 'contains(bedevilling)': False, 'adverbs': 0, 'contains(this)': False, 'contains(Mothman)': False, 'contains(level)': False, 'contains(TV)': False, 'contains(sustain)': False, 'contains(cross)': False, 'contains(a)': False, 'contains(and)': True, 'contains(Cry)': False, 'contains(Prophecies)': False, 'contains(made)': False, 'contains(best)': False, 'contains(Deliverance)': False, 'contains(lies)': False, 'contains(very)': False, 'contains(set)': False, 'contains(Marker)': False, 'contains(masculine)': False, 'contains(,)': False, 'contains(Ode)': False, 'contains(con)': True, 'contains(illustrating)': False, 'contains(somewhere)': False, 'verbs': 0, 'contains(enough)': False, 'modals': 0, 'nouns': 2, 'contains(invention)': False, 'contains(story)': False, 'contains(high)': False, 'contains(Joe)': False, 'contains(Billy)': False, 'contains(while)': False, 'contains(update)': False, 'contains(Boys)': False, 'adjectives': 0, 'contains(clever)': False, 'contains(touch)': False, 'contains(between)': False, 'contains(-)': False, 'contains(is)': False, 'contains(Do)': False, 'contains(human)': False, 'contains(does)': False, 'interjections': 0, 'contains(spots)': False, 'contains(Shakespeare)': False, 'contains(.)': False, 'contains(movie)': False, 'contains(players)': True, 'contains(modern)': False, 'contains(light)': False, 'contains(demons)': False, 'contains(unentertaining)': False, 'contains(The)': False, 'symbols': 0, 'contains(but)': False, 'contains(that)': False, "contains('s)": False, 'contains(need)': False, 'contains(spoofy)': False}, 2)

Below is the code we wrote for POS feature function:



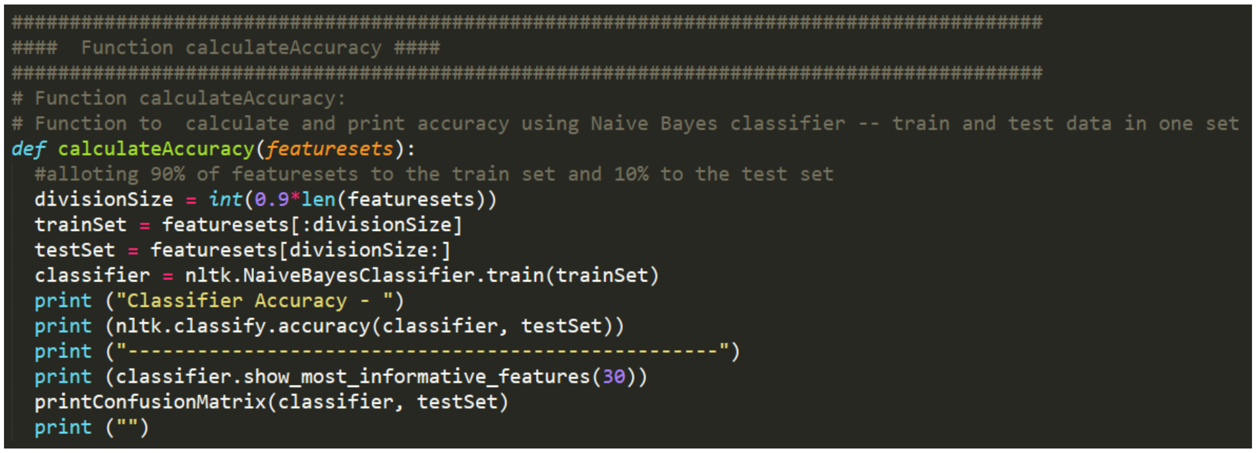
```
##########################################################################################
####  Function posFeatures ####
##########################################################################################
# Function posFeatures:
#    returns feature set based on POS tagging done using stanford pos tagger.
def posFeatures(document, wordFeatures):
  documentWords = set(document)
  features = {}
  taggedWords = nltk.pos_tag(document)
  for word in wordFeatures:
    features['contains({})'.format(word)] = (word in documentWords)
  countNoun = countVerb = countAdj = countAdv = countModal = countSym = countDet = countInterj = 0
  for (word, tag) in taggedWords:
    if tag.startswith('N'): countNoun+=1
    if tag.startswith('V'): countVerb+=1
    if tag.startswith('J'): countAdj+=1
    if tag.startswith('R'): countAdv+=1
    if tag.startswith('M'): countModal+=1
    if tag.startswith('S'): countSym+=1
    if tag.startswith('D'): countDet+=1
    if tag.startswith('U'): countInterj+=1
  features['nouns'] = countNoun
  features['verbs'] = countVerb
  features['adjectives'] = countAdj
  features['adverbs'] = countAdv
  features['modals'] = countModal
  features['symbols'] = countSym
  features['determiners'] = countDet #useful for unprocessed data
  features['interjections'] = countInterj
  return features
```

e. Classification (Using NLTK Naïve Bayes Classifier):

NLTK provides inbuilt Naïve Bayes Classifier. To work on the training set, we had to first divide it into training data to train our classifier and testing data to test the trained classifier on. We used 90% of the data of feature sets constructed from train.tsv to train our classifier and remaining 10% to test it. After training the classifier, we used nltk library method nltk.classify.accuracy to test our test data against the classifier and gives its accuracy. Classifier needs to have reference result class in the test data so as to compare its result with, otherwise it won't be able to find the accuracy. For this purpose, we have kept the resultant sentiment class in our dataset from starting.

We have also displayed the top 30 most informative features as identified by the classifier. We used the calculateAccuracy method to calculate the accuracy of all of our feature functions.

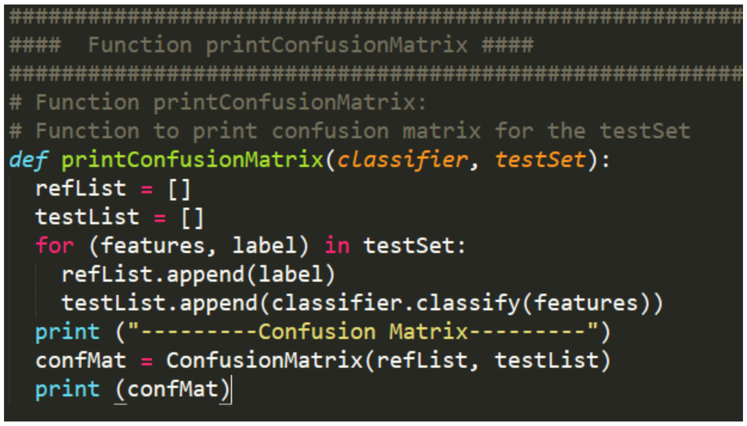
Below is the code snippet for the calculateAccuracy method:



```
##########################################################################################
####  Function calculateAccuracy ####
##########################################################################################
# Function calculateAccuracy:
# Function to  calculate and print accuracy using Naive Bayes classifier -- train and test data in one set
def calculateAccuracy(featuresets):
  #alloting 90% of featuresets to the train set and 10% to the test set
  divisionSize = int(0.9*len(featuresets))
  trainSet = featuresets[:divisionSize]
  testSet = featuresets[divisionSize:]
  classifier = nltk.NaiveBayesClassifier.train(trainSet)
  print ("Classifier Accuracy - ")
  print (nltk.classify.accuracy(classifier, testSet))
  print ("---------------------------------------------------")
  print (classifier.show_most_informative_features(30))
  printConfusionMatrix(classifier, testSet)
  print ("")
```

Finding the accuracy is not enough to truly understand the result of the classification. Hence, we needed to print Confusion Matrix of our Naïve Bayes Classifier's result over each test data. Confusion Matrix shows the results of a test for how many of the actual class labels (the gold standard labels) match with the predicted labels.

Below is the code snippet for printing ConfusionMatrix:



```
########################################################
####  Function printConfusionMatrix ####
########################################################
# Function printConfusionMatrix:
# Function to print confusion matrix for the testSet
def printConfusionMatrix(classifier, testSet):
  refList = []
  testList = []
  for (features, label) in testSet:
    refList.append(label)
    testList.append(classifier.classify(features))
  print ("---------Confusion Matrix---------")
  confMat = ConfusionMatrix(refList, testList)
  print (confMat)
```

Sample Output (Accuracy and Confusion Matrix for all the features):

I observed the accuracy and confusion matrix for all the features for randomly selected 600 records. Below is the output of the same:

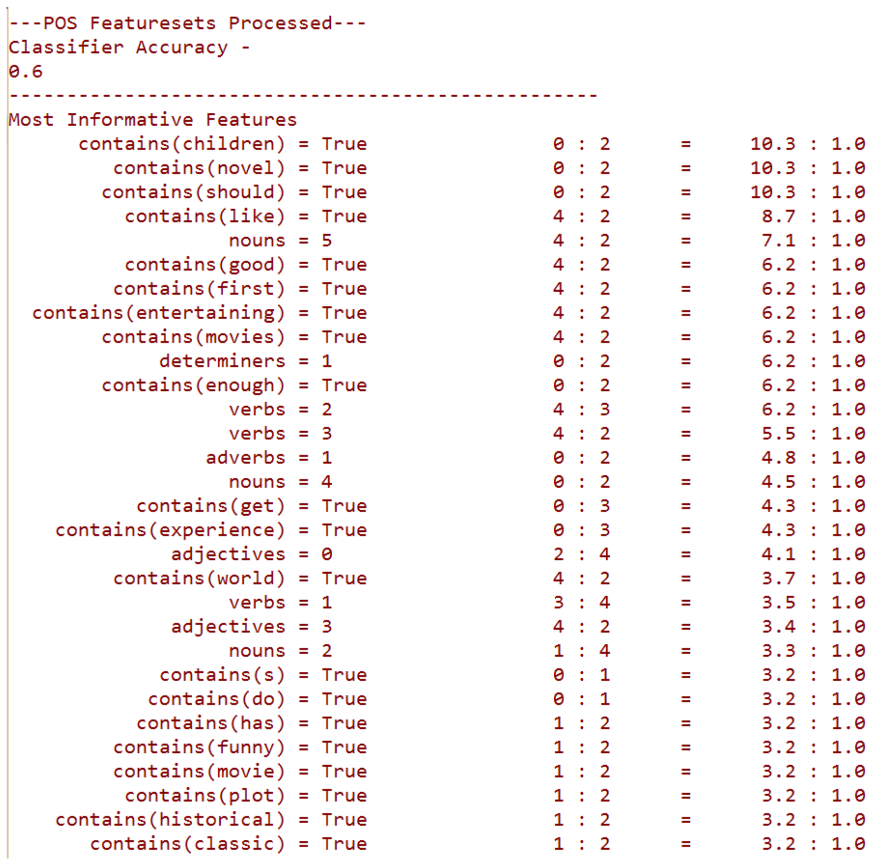
```
E:\NaturalLangProcessing\kagglemoviereviews\kagglemoviereviews>classifyKaggle.py
./corpus 600
        Read 156060 phrases, using 600 random phrases
        ---Unprocessed Featuresets---
        Classifier Accuracy -
        0.48333333333333334
        ---------------------------------------------------
        ---------Confusion Matrix---------
          | 0  1  2  3  4 |
        --+----------------+
        0 | <.> .  1  .  . |
        1 |  . <.> 8  2  . |
        2 |  .  3<25> 4  2 |
        3 |  1  1  6 <3> . |
        4 |  .  .  2  1 <1>|
        --+----------------+
        (row = reference; col = test)

        ---Pre-processed Featuresets---
        Classifier Accuracy -
        0.5166666666666667
        ---------------------------------------------------
        ---------Confusion Matrix---------
          | 0  1  2  3  4 |
        --+----------------+
        0 | <.> .  1  .  . |
        1 |  . <.> 7  3  . |
        2 |  .  1<28> 4  1 |
        3 |  .  1  6 <3> 1 |
        4 |  1  1  2  . <.>|
        --+----------------+
        (row = reference; col = test)


        ---SL Featuresets Processed---
        Classifier Accuracy -
        0.5
        ---------------------------------------------------
        ---------Confusion Matrix---------
          | 0  1  2  3  4 |
        --+----------------+
        0 | <.> .  1  .  . |
        1 |  . <1> 9  . . |
        2 |  .  3<27> 3  1 |
        3 |  1  2  6 <2> . |
        4 |  1  1  2  . <.>|
        --+----------------+
        (row = reference; col = test)


        ---NOT Featuresets Processed---
        Classifier Accuracy -
        0.5333333333333333
        --------------------------------------------------
        ---------Confusion Matrix---------
          |  0  1  2  3  4 |
        --+----------------+
        0 | <.> .  1  . . |
        1 |  . <.> 7  3  . |
        2 |  .  1<29> 3  1 |
        3 |  .  1  6 <3> 1 |
        4 |  1  1  2  . <.>|
        --+----------------+
        (row = reference; col = test)


        ---Processed Bigram Features---
        Classifier Accuracy -
        0.5166666666666667
        --------------------------------------------------
        ---------Confusion Matrix---------
          |  0  1  2  3  4 |
        --+----------------+
        0 | <.> .  1  . . |
        1 |  . <.> 7  3  . |
        2 |  .  1<28> 4  1 |
        3 |  .  1  6 <3> 1 |
        4 |  1  1  2  . <.>|
        --+----------------+
        (row = reference; col = test)


        ---LIWC Featuresets Processed---
        Classifier Accuracy -
        0.5166666666666667
        --------------------------------------------------
        ---------Confusion Matrix---------
          |  0  1  2  3  4 |
        --+----------------+
        0 | <.> .  1  . . |
        1 |  . <1> 7  2  . |
        2 |  .  3<27> 3  1 |
        3 |  .  1  6 <3> 1 |
        4 |  .  1  1  2 <.>|
        --+----------------+
        (row = reference; col = test)


        ---POS Featuresets Processed---
        Classifier Accuracy -
        0.5166666666666667
        --------------------------------------------------
        ---------Confusion Matrix---------
          |  0  1  2  3  4 |
        --+----------------+
        0 | <.> .  1  .  . |
        1 |  . <1> 5  4  . |
        2 |  .  3<26> 3  2 |
        3 |  .  2  4 <4> 1 |
        4 |  1  1  1  1 <.>|
        --+----------------+
        (row = reference; col = test)


        ---Processed Trigram Features---
        Classifier Accuracy -
        0.5166666666666667
        --------------------------------------------------
        ---------Confusion Matrix---------
          |  0  1  2  3  4 |
        --+----------------+
        0 | <.> .  1  .  . |
        1 |  . <.> 7  3  . |
        2 |  .  1<28> 4  1 |
        3 |  .  1  6 <3> 1 |
        4 |  1  1  2  . <.>|
        --+----------------+
        (row = reference; col = test)
```

Most informative features:

After calculating the accuracy, we displayed most informative top 30 features for each of the created feature function. For this we used, show_most_informative_features() method provided by NLTK Naïve Bayes Classifier. Below is the sample code:

```
print (classifier.show_most_informative_features(30))
```

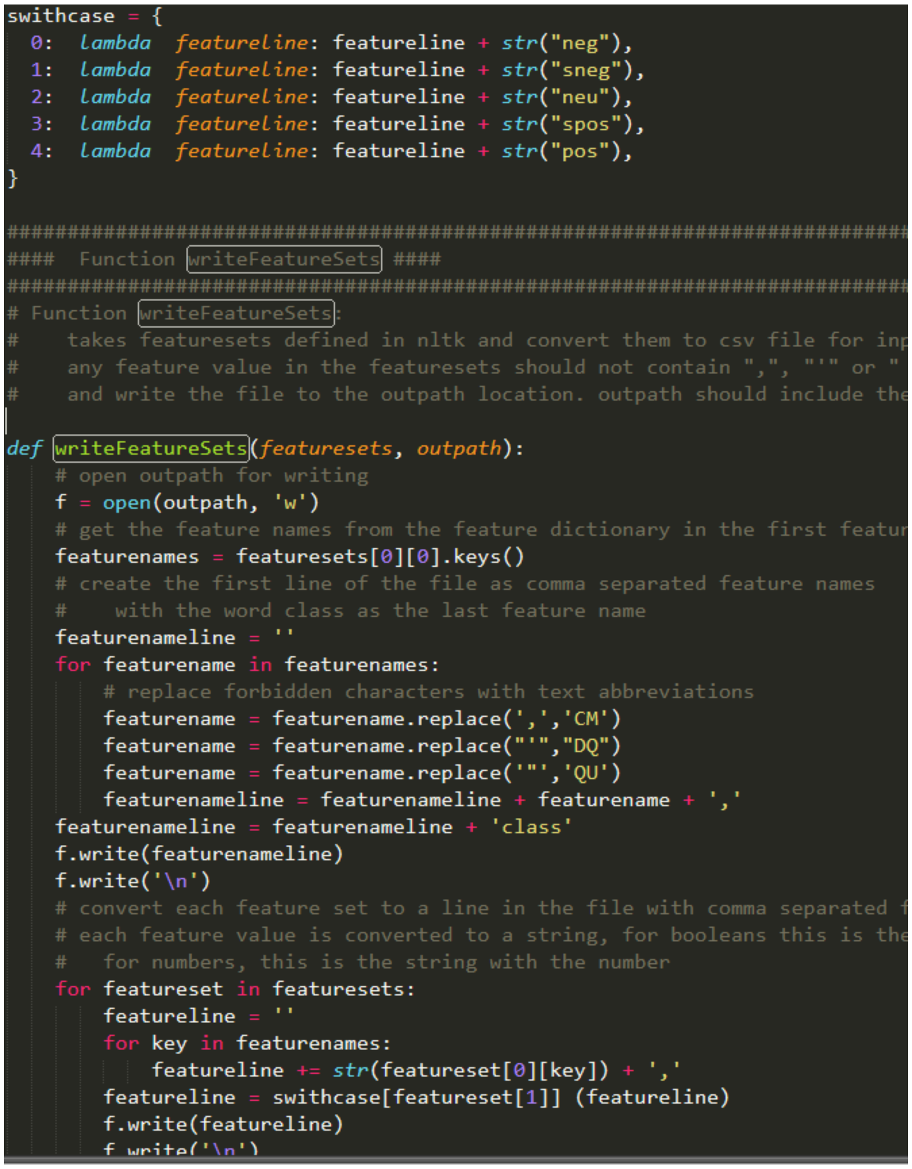
Example of most informative features for POS features set for processed data:



```
---POS Featuresets Processed---
Classifier Accuracy -
0.6
---------------------------------------------------
Most Informative Features
      contains(children) = True              0 : 2      =     10.3 : 1.0
         contains(novel) = True              0 : 2      =     10.3 : 1.0
        contains(should) = True              0 : 2      =     10.3 : 1.0
          contains(like) = True              4 : 2      =      8.7 : 1.0
                   nouns = 5                 4 : 2      =      7.1 : 1.0
          contains(good) = True              4 : 2      =      6.2 : 1.0
         contains(first) = True              4 : 2      =      6.2 : 1.0
  contains(entertaining) = True              4 : 2      =      6.2 : 1.0
        contains(movies) = True              4 : 2      =      6.2 : 1.0
             determiners = 1                 0 : 2      =      6.2 : 1.0
        contains(enough) = True              0 : 2      =      6.2 : 1.0
                   verbs = 2                 4 : 3      =      6.2 : 1.0
                   verbs = 3                 4 : 2      =      5.5 : 1.0
                 adverbs = 1                 0 : 2      =      4.8 : 1.0
                   nouns = 4                 0 : 2      =      4.5 : 1.0
           contains(get) = True              0 : 3      =      4.3 : 1.0
    contains(experience) = True              0 : 3      =      4.3 : 1.0
              adjectives = 0                 2 : 4      =      4.1 : 1.0
         contains(world) = True              4 : 2      =      3.7 : 1.0
                   verbs = 1                 3 : 4      =      3.5 : 1.0
              adjectives = 3                 4 : 2      =      3.4 : 1.0
                   nouns = 2                 1 : 4      =      3.3 : 1.0
             contains(s) = True              0 : 1      =      3.2 : 1.0
            contains(do) = True              0 : 1      =      3.2 : 1.0
           contains(has) = True              1 : 2      =      3.2 : 1.0
         contains(funny) = True              1 : 2      =      3.2 : 1.0
         contains(movie) = True              1 : 2      =      3.2 : 1.0
          contains(plot) = True              1 : 2      =      3.2 : 1.0
    contains(historical) = True              1 : 2      =      3.2 : 1.0
       contains(classic) = True              1 : 2      =      3.2 : 1.0
```

f. Write CSV file for all the features

Now, to use the feature sets in outside tools, like Weka or Python Sci-Kit library, we need to import the features into csv files. For this we used a writeFeatureSet function which writes all the data to a csv file. This function needs to be called for all the types of Features, once the feature set has been constructed. Below is the code for the feature set function. We have created files for each of the features in there unprocessed and processed format. However, we are not using the unprocessed ones further in our processing for any of the tools. We are not going to attach all the CSVs in zip as they are bulky. All the lines are there in the code but uncommented for unprocessed ones. CSVs are created for 2500 records for all the features.



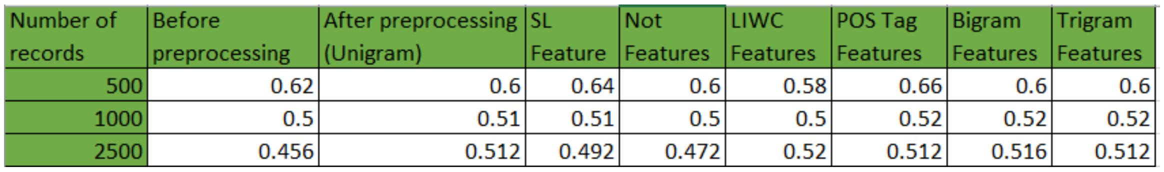
```
swithcase = {
  0:  lambda  featureline: featureline + str("neg"),
  1:  lambda  featureline: featureline + str("sneg"),
  2:  lambda  featureline: featureline + str("neu"),
  3:  lambda  featureline: featureline + str("spos"),
  4:  lambda  featureline: featureline + str("pos"),
}

###########################################################################
####  Function writeFeatureSets ####
###########################################################################
# Function writeFeatureSets:
#    takes featuresets defined in nltk and convert them to csv file for inp
#    any feature value in the featuresets should not contain ",", "'" or "
#    and write the file to the outpath location. outpath should include the

def writeFeatureSets(featuresets, outpath):
    # open outpath for writing
    f = open(outpath, 'w')
    # get the feature names from the feature dictionary in the first featur
    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 = featurenameline + featurename + ','
    featurenameline = featurenameline + 'class'
    f.write(featurenameline)
    f.write('\n')
    # convert each feature set to a line in the file with comma separated f
    # each feature value is converted to a string, for booleans this is the
    #   for numbers, this is the string with the number
    for featureset in featuresets:
        featureline = ''
        for key in featurenames:
            featureline += str(featureset[0][key]) + ','
        featureline = swithcase[featureset[1]] (featureline)
        f.write(featureline)
        f.write('\n')
```

g. Experiments:

1. Filtered by stopwords as mentioned in preprocessing step.

2. Used Subjectivity and LIWC both for feature generation as mentioned with SL Features and LIWC Features in producing features section.

3. We also compared performance of different Feature method, by increasing number of records. We compared the performance before preprocessing and after preprocessing the data by removing special characters and number and applying Unigram, SL Features, Not Features, LIWC Features, POS Tag Features, Bigram Features, and Trigram Features.

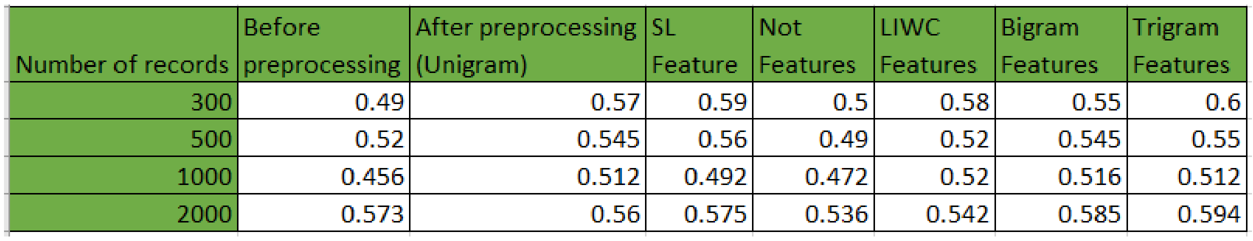


Number of records	Before preprocessing	After preprocessing (Unigram)	SL Feature	Not Features	LIWC Features	POS Tag Features	Bigram Features	Trigram Features
500	0.62	0.6	0.64	0.6	0.58	0.66	0.6	0.6
1000	0.5	0.51	0.51	0.5	0.5	0.52	0.52	0.52
2500	0.456	0.512	0.492	0.472	0.52	0.512	0.516	0.512

Observation:

We can see that the Feature functions gave better accuracy in limited number of records. Also, accuracy increases significantly after preprocessing and removing the stopwords from dataset. In limited records, POS Tag Features gave the best accuracy and in higher number of records, LIWC gave the best accuracy in our experiments. However, the accuracy is moreover same with little differences each time.

4. Comparing accuracy by changing the size of vocabulary.



Number of records	Before preprocessing	After preprocessing (Unigram)	SL Feature	Not Features	LIWC Features	Bigram Features	Trigram Features
300	0.49	0.57	0.59	0.5	0.58	0.55	0.6
500	0.52	0.545	0.56	0.49	0.52	0.545	0.55
1000	0.456	0.512	0.492	0.472	0.52	0.516	0.512
2000	0.573	0.56	0.575	0.536	0.542	0.585	0.594

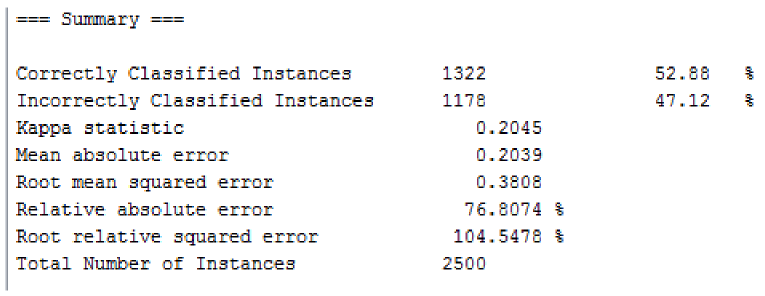
Observation:

Increasing the size of vocabulary, shows a general increase in accuracy across feature functions. This general increase is may be because there are bigger feature sets to classify the data when vocabulary size increases.

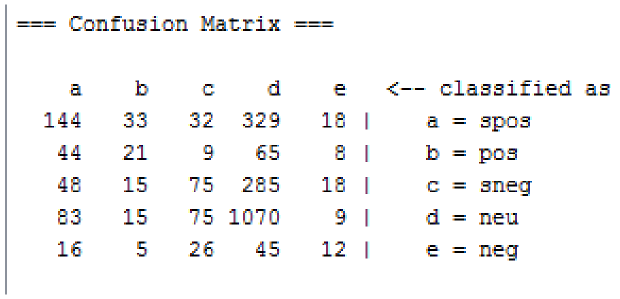
h. Advanced Experiments:

1. Experiments on CSV file in Weka:

Naïve Bayes Classifier with SL Features (2500 records) with cross validation of 10 folds.

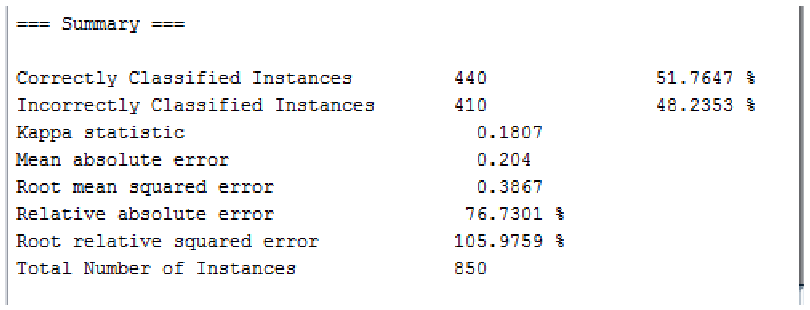


```
=== Summary ===

Correctly Classified Instances        1322               52.88   %
Incorrectly Classified Instances      1178               47.12   %
Kappa statistic                          0.2045
Mean absolute error                      0.2039
Root mean squared error                  0.3808
Relative absolute error                 76.8074 %
Root relative squared error            104.5478 %
Total Number of Instances             2500
```

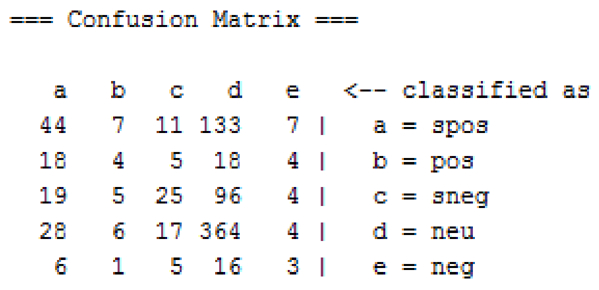


```
=== Confusion Matrix ===

   a    b    c    d    e   <-- classified as
 144   33   32  329   18 |    a = spos
  44   21    9   65    8 |    b = pos
  48   15   75  285   18 |    c = sneg
  83   15   75 1070    9 |    d = neu
  16    5   26   45   12 |    e = neg
```

Naïve Bayes Classifier with SL Features (2500 records) with training set of 66% (i.e. 1650 training records and 850 test records).



```
=== Summary ===

Correctly Classified Instances         440               51.7647 %
Incorrectly Classified Instances       410               48.2353 %
Kappa statistic                          0.1807
Mean absolute error                      0.204
Root mean squared error                  0.3867
Relative absolute error                 76.7301 %
Root relative squared error            105.9759 %
Total Number of Instances              850
```



```
=== Confusion Matrix ===

  a   b   c   d   e   <-- classified as
 44   7  11 133   7 |   a = spos
 18   4   5  18   4 |   b = pos
 19   5  25  96   4 |   c = sneg
 28   6  17 364   4 |   d = neu
  6   1   5  16   3 |   e = neg
```

Naïve Bayes Classifier with SL Features (2500 records) with training set of 90% (i.e. 2250 training records and 250 test records).

```
=== Summary ===

Correctly Classified Instances         142               56.8    %
Incorrectly Classified Instances       108               43.2    %
Kappa statistic                          0.2488
Mean absolute error                      0.1926
Root mean squared error                  0.3645
Relative absolute error                 73.7124 %
Root relative squared error            101.9038 %
Total Number of Instances              250
```

```
=== Confusion Matrix ===

   a   b   c   d   e   <-- classified as
  12   0   5  29   2 |   a = spos
   3   2   1   1   1 |   b = pos
   4   3  13  32   3 |   c = sneg
  12   1   6 113   0 |   d = neu
   2   0   0   3   2 |   e = neg
```

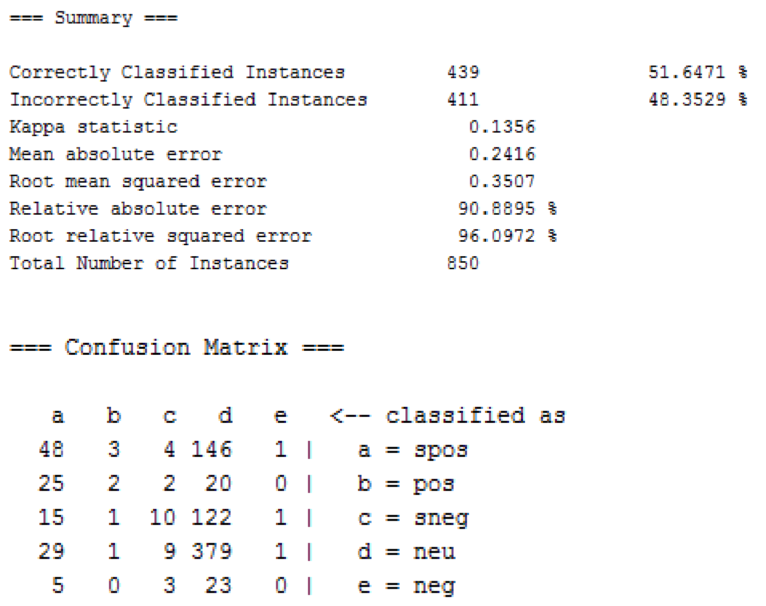
J48 Classifier (with default settings) with SL Features (2500 records) with training set of 66%(i.e. 1650 training records and 850 test records).

```
=== Summary ===

Correctly Classified Instances         415               48.8235 %
Incorrectly Classified Instances       435               51.1765 %
Kappa statistic                          0.1574
Mean absolute error                      0.2435
Root mean squared error                  0.3856
Relative absolute error                 91.614  %
Root relative squared error            105.6569 %
Total Number of Instances              850
```

```
=== Confusion Matrix ===

   a   b   c   d   e   <-- classified as
  78  12   8 102   2 |   a = spos
  27   6   3  13   0 |   b = pos
  27   4  16 102   0 |   c = sneg
  82   3  18 314   2 |   d = neu
   7   1   3  19   1 |   e = neg
```

Simple Logistic Classifier (with default settings) with SL Features (2500 records) with training set of 66%(i.e. 1650 training records and 850 test records).

Tried to work with Logistic classifier in Weka but it kept running for an hour. Hence, tried the same with Simple Logistic Classifier.



```
=== Summary ===

Correctly Classified Instances         439               51.6471 %
Incorrectly Classified Instances       411               48.3529 %
Kappa statistic                          0.1356
Mean absolute error                      0.2416
Root mean squared error                  0.3507
Relative absolute error                 90.8895 %
Root relative squared error             96.0972 %
Total Number of Instances              850

=== Confusion Matrix ===

   a   b   c   d   e   <-- classified as
  48   3   4 146   1 |   a = spos
  25   2   2  20   0 |   b = pos
  15   1  10 122   1 |   c = sneg
  29   1   9 379   1 |   d = neu
   5   0   3  23   0 |   e = neg
```

Observation:

As we can see from the above experiments in Weka that Naïve Bayes Classifier still gives the best accuracy on the SL Feature data set with 2500 records. The highest accuracy we got was **56.8%.** As we are using Naïve Bayes Classifier provided by NLTK in our code accuracy calculation as well, we are also not far off in terms of accuracy of our methods.

2. Using Sci-Kit Learn python library to run cross-validation on different feature CSVs.

We used Sci-Kit python library and assembled the code from online sources to write the script to run on features CSVs generated for each of the feature function. These generated CSVs are for the processed data except the one before preprocessing. In this Sci-Kit script, we did a cross-validation to find out the Precision, Recall, and F-Measure score for all the mentioned classifiers in the below table.

Number of records	Before preprocessing	After preprocessing (Unigram)	SL Feature	Not Features	LIWC Features	POS Tag Features	Bigram Features	Trigram Features
Precision score	0.42	0.44	0.52	0.49	0.54	0.57	0.54	0.54
Recall score	0.41	0.42	0.5	0.48	0.57	0.55	0.52	0.53
F-measure score	0.41	0.43	0.5	0.47	0.58	0.56	0.53	0.53

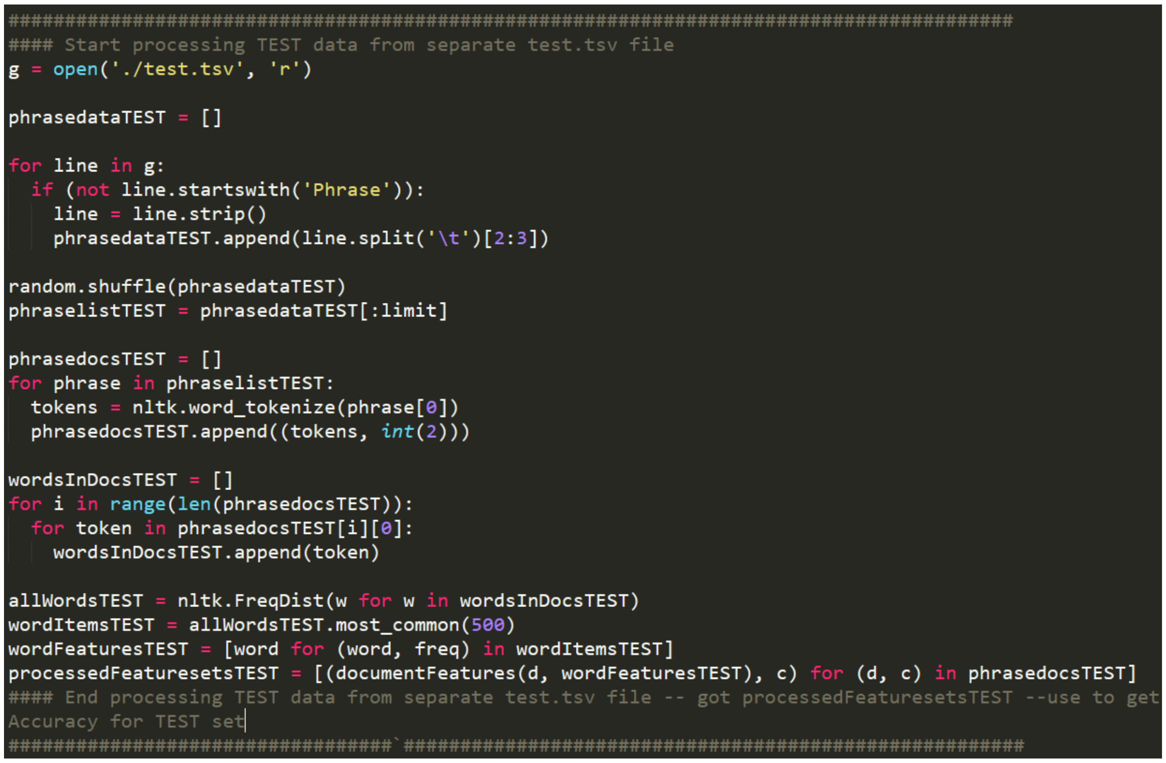
In our experiments, we found the POS Tag Features to be performing slightly better than other Features when comparing their Precision, Recall, and F-Measure score. However, LIWC was comparatively equal to POS Tag Features function.

2. **Train the classifier on the entire training set and test it on a separately available test set:**

To do this experiment, we had to make some changes in the test set as our test.tsv file contains only the phrases and for the accuracy function to compare the results of the classification it needs reference results class.

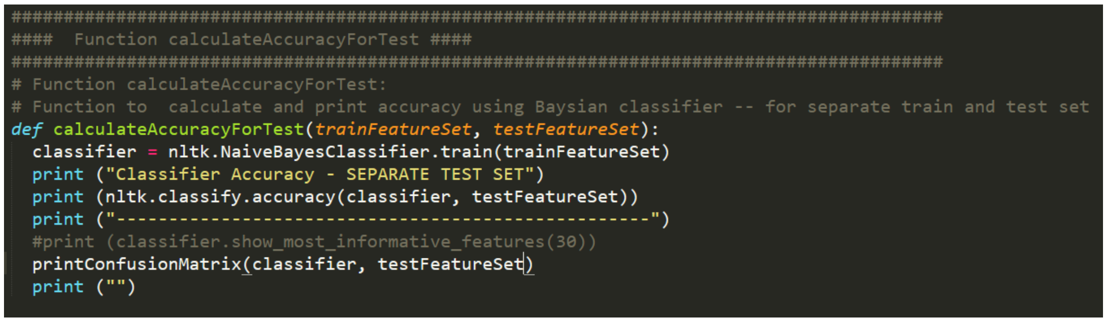
As we don't have the classes for MovieReview data test set in test.tsv. So, just to check the results of the classification we set the output result class as '2' by default in our test data set.

Below is the code for the same:



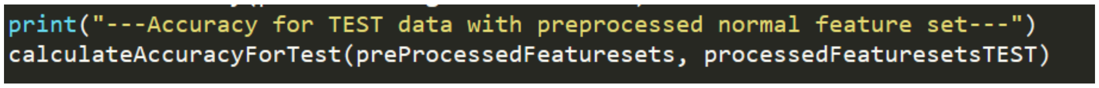
```
##########################################################################################
#### Start processing TEST data from separate test.tsv file
g = open('./test.tsv', 'r')

phrasedataTEST = []

for line in g:
  if (not line.startswith('Phrase')):
    line = line.strip()
    phrasedataTEST.append(line.split('\t')[2:3])

random.shuffle(phrasedataTEST)
phraselistTEST = phrasedataTEST[:limit]

phrasedocsTEST = []
for phrase in phraselistTEST:
  tokens = nltk.word_tokenize(phrase[0])
  phrasedocsTEST.append((tokens, int(2)))

wordsInDocsTEST = []
for i in range(len(phrasedocsTEST)):
  for token in phrasedocsTEST[i][0]:
    wordsInDocsTEST.append(token)

allWordsTEST = nltk.FreqDist(w for w in wordsInDocsTEST)
wordItemsTEST = allWordsTEST.most_common(500)
wordFeaturesTEST = [word for (word, freq) in wordItemsTEST]
processedFeaturesetsTEST = [(documentFeatures(d, wordFeaturesTEST), c) for (d, c) in phrasedocsTEST]
#### End processing TEST data from separate test.tsv file -- got processedFeaturesetsTEST --use to get
Accuracy for TEST set
##################################`#######################################################
```

Also, we cannot use the existing calculateAccuracy method to get the accuracy as it takes only one feature set and splits training and testing data from within it. So, to get the accuracy for the test data we need to create a new calculateAccuracyForTest method which takes two different training and testing data set and finds accuracy for testing data set.



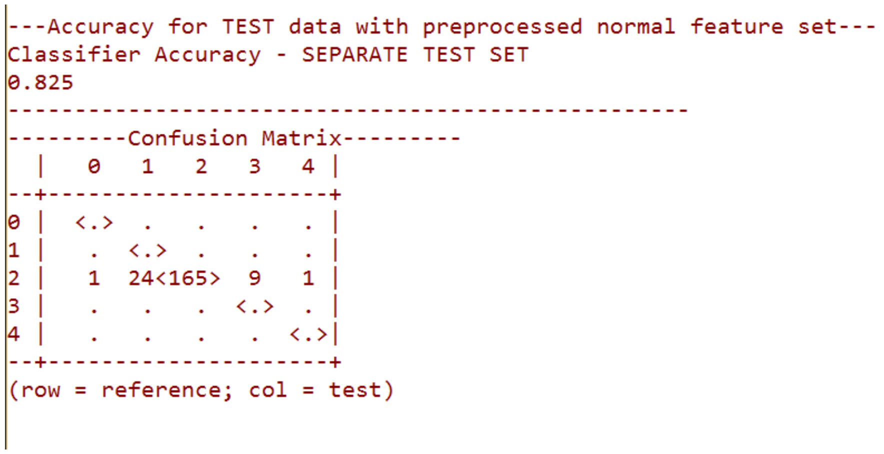
```
##########################################################################################
####  Function calculateAccuracyForTest ####
##########################################################################################
# Function calculateAccuracyForTest:
# Function to  calculate and print accuracy using Baysian classifier -- for separate train and test set
def calculateAccuracyForTest(trainFeatureSet, testFeatureSet):
  classifier = nltk.NaiveBayesClassifier.train(trainFeatureSet)
  print ("Classifier Accuracy - SEPARATE TEST SET")
  print (nltk.classify.accuracy(classifier, testFeatureSet))
  print ("--------------------------------------------------")
  #print (classifier.show_most_informative_features(30))
  printConfusionMatrix(classifier, testFeatureSet)
  print ("")
```

This is how we can make use of the new calculateAccuracyForTest method to find the accuracy.



```
print("---Accuracy for TEST data with preprocessed normal feature set---")
calculateAccuracyForTest(preProcessedFeaturesets, processedFeaturesetsTEST)
```

Below is the accuracy of the Naïve Bayes Classifier with preprocessed normal feature set for separate train and test data (with 200 records):



```
---Accuracy for TEST data with preprocessed normal feature set---
Classifier Accuracy - SEPARATE TEST SET
0.825
--------------------------------------------------
---------Confusion Matrix---------
  |   0   1   2   3   4 |
--+---------------------+
0 |  <.>  .   .   .   . |
1 |   .  <.>  .   .   . |
2 |   1  24<165>  9   1 |
3 |   .   .   .  <.>  . |
4 |   .   .   .   .  <.>|
--+---------------------+
(row = reference; col = test)
```

As we can see here that accuracy is very high, it's due to the fact that most of the records as predicted by tin the test.tsv falls under sentiment category 2 and we have also set our expected resultant output class for all the records to be 2 in code. Hence, most of the records' predicted class matches the expected class. If we change, our expected class to 1, accuracy will drastically drop. We had to this just to check what classes are predicted for each phrase in the test.tsv.

3. Work Distribution:

Rohit Sharma:

- SL Features
- NOT Features
- LIWC Features
- Bigram Trigram Features
- POS Features
- Write CSV file for the features
- experiments with feature functions
- Train on different data, test on different data
- Sci-Kit learn library
- Report creation.

Achal Velani:

- Reading the data from train.tsv file
- Preprocessing the data
- Feature selection
- Unigram/Bigram Features
- SL Features
- NOT Features
- Calculating accuracy and creating Confusion matrix
- Train on different data, test on different data
- Experiment with Weka
- Report creation.

4. References:

1. NLP Class Lab Exercises.
2. https://www.kaggle.com/c/sentiment-analysis-on-movie-reviews
3. https://en.wikipedia.org/wiki/Weka_(machine_learning)
4. http://scikit-learn.org/stable/