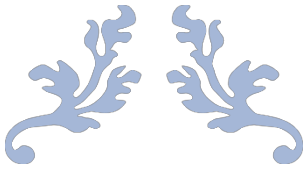

SEMANTIC ANALYSIS OF AMAZON PRODUCT REVIEWS

NLP HOMEWORK 4

APRIL 22, 2018

KOMAL GUJARATHI, 211778351
SYRACUSE UNIVERSITY

Table of Contents

Semantic Analysis of Amazon Product Reviews

It is increasingly common that Internet users engage in various of online reviews. The availability of these review content offers researchers opportunities to better understand and model online social behavior. In this homework, you will conduct sentiment analysis to gain some understanding about the Amazon product reviews.
I have chosen the 'Clothing, Shoes and Jewelry' dataset for analysis.

This File 'clothing_shoes_jewelry.txt' is mainly divided into number of reviews from the users from different years
Each review is of the format –

- reviewerID: ID of the reviewer
- asin: ID of the product
- reviewerName: name of the reviewer
- helpful: helpfulness rating of the review, e.g. 2/3
- reviewText: text of the product
- overall: rating of the product
- summary: summary of the review
- unixReviewTime: time of the review (unix time)
- reviewTime: time of the review (raw)

Data Pre-Processing Step (20%)

Write the Python Code that extracts only review texts.

I have first used the regex pattern matching to match with each review in the given file. Upon processing, I could find that there are around 278225 reviews in this given text, with each review being the combination of one or more sentences by the customer of that product. Since it takes a lot of processing time to run the classifier on each of these reviews, I have mainly decided to work with a small part of this dataset. I will be working with around 5500 reviews from the year 2014.
From the reviews text, I could see that there are many reviews spanning from years including 2012, 2013, 2014. I have decided to work with 2014 reviews mainly because it is the latest year, and it will be more useful for the users of this classifier to get the reviews of the recent years, than the old set of reviews.

In the year 2014, there are total 67617 reviews. I have run the classifier on 5500 of these reviews as my device was taking more than 10 hours to process even small set of inputs. Please find the screen shot below.

Code -

pword=re.compile('reviewerID:(.*)\nasin:(.*)\nreviewerName:(.*)\nhelpful:(.*)\nreviewText:(.*)\n(overall:.*)\nsummary:(.*)\nunixReviewTime:(.*)\nreviewTime:(.*)\n')
reviews = re.findall(pword,file_content)

I have used regex pattern matching to find out the reviewText of each of the chosen review.

```
In [3]: mycorpus = PlaintextCorpusReader('.','clothing_shoes_jewelry.txt')
        type(mycorpus.fileids())

Out[3]: list

In [4]: file_id = mycorpus.fileids()
        file_contents = []
        file = file_id[0]
        file_handle = open(file, 'r')
        file_content = file_handle.read()
        file_handle.close()
        print(file_content[:500])

        reviewerID:A1KLRMWW2FWPL4
        asin:0000031887
        reviewerName:Amazon Customer "cameramom"
        helpful:[0, 0]
        reviewText:This is a great tutu and at a really great price. It doesn't look cheap at all. I'm so glad I looked on Am
        azon and found such an affordable tutu that isn't made poorly. A++
        overall:5.0
        summary:Great tutu-  not cheaply made
        unixReviewTime:1297468800
        reviewTime:02 12, 2011

        reviewerID:A2G5TCU2WDFZ65
        asin:0000031887
        reviewerName:Amazon Customer
        helpful:[0, 0]
        reviewText:I bought this for my

In [5]: # extract all reviews

        pword = re.compile('reviewerID:(.*)\nasin:(.*)\nreviewerName:(.*)\nhelpful:(.*)\nreviewText:(.*)\n(overall:.*)\nsummary
        reviews = re.findall(pword,file_content)

In [6]: len(reviews)

Out[6]: 278225
```

In [7]:
```
# extract the year
int(reviews[11][8][7:])
```

Out[7]:
```
2014
```

In [9]:
```
# count number of reviews in 2014
counter = 0
for i in range(len(reviews)):
    year = int(reviews[i][8][7:])
    if (year == 2014):
        # only [i][4] corresponds to the reviewText field.
        counter+=1

print("number of reviews in 2014 : " + str(counter))
```

```
number of reviews in 2014 : 67614
```

In [69]:
```
# reviews_2014 only first 5500 contains the reviews of year 2014
reviews_2014 = []
counter = 0
for i in range(len(reviews)):
    year = int(reviews[i][8][7:])
    if (year == 2014):
        # only [i][4] corresponds to the reviewText field.
        reviews_2014.append(reviews[i][4])
        counter+=1
        if (counter >= 5500):
            break
```

In [70]:
```
len(reviews_2014)
```

Out[70]:
```
5500
```

In [71]:
```
reviews_2014[:2]
```

Out[71]:
```
['We bought several tutus at once, and they are got high reviews. Sturdy and seemingly well-made. The girls have been
wearing them regularly, including out to play, and the tutus have stood up well. Fits the 3-yr old & the 5-yr old wel
l. Clearly plenty of room to grow. Only con is that when the kids pull off the tutus, the waste band gets twisted, an
```

In [409]:
```
# Sentence level tokenization
sentences=[nltk.sent_tokenize(sent) for sent in reviews_2014]
```

In [411]:
```
sentences[-2]
```

Out[411]:
```
["It won't hold heavy canned goods like a cotton canvas bag, and is a little smaller than paper grocery bag.",
 'Good for the beach or knitting carryall.',
 'The fabric seems to be polyester with a waterproof coating inside that also stabilizes it.',
 'The plastic smell does go away in a couple of days IF you open them up to air out.',
 "There's no stiffener in the bottom so it folds very flat.",
 'While the seams are stitched adequately, they are not finished on the inside, though a couple of colors do have tap
ed seams.',
 'But for $3.50, they are a bargain.']
```

In [431]:
```
# tokenize each review
# Sentence level tokenization, I am using Tweet Tokenizer because
# if we use nltk.sent_tokenize, the words like didn't gets split, which we dont want.

tknzr = TweetTokenizer()
original_tokenized_review_sentences = []
for each_review in sentences:
    # tokenize each review into words by splitting them into different sentences
    sent_wordlevel=[tknzr.tokenize(sent) for sent in each_review]
    for each in sent_wordlevel:
        original_tokenized_review_sentences.append(each)

print(len(original_tokenized_review_sentences))
print(original_tokenized_review_sentences[-3:-1])
```

```
21629
[['I', 'took', 'it', 'to', 'ups', 'myself', 'and', 'paid', 'the', 'return', 'with', '$', '100.00', 'insurance', 'secu
red', 'for', 'what', 'look', 'like', 'are', 'boots', 'worth', '$', '20', '.'], ["I'm", 'sorry', 'I', 'have', 'not',
'had', 'any', 'issues', 'ever', 'with', 'Amazon', 'cause', 'your', 'stuff', 'is', 'great', 'and', "I'm", 'always', 'h
appy', 'with', 'my', 'purchases', 'but', 'since', 'I', 'kept', 'my', 'receipts', 'on', 'the', 'returns', 'of', 'thes
e', 'items', 'please', 'ask', 'your', '3rd', 'party', 'seller', 'what', 'happen', '!']]
```

In [436]:
```
# converting the sentences to lower case to have the uniformity during classification
tokenized_review_sentences = []
# sentences now has the originally selected sentences from the reiviews file
```

Sentiment Analysis (80%)

Now, after getting the reviews, we had to split the reviews in the sentence level. I have used nltk.sent_tokenizer() to get the sentences in each review. Also, during the preprocessing step, I have converted all the text in these sentences into the lower case to make the classification more accurate. Also, while word_tokenizing the sentences, I have used tweetTokenizer. Doing this, I could preserve the n't type words. It can be clearly seen in the following screen shot. If we use the nltk.sent_tokenize, the words like didn't gets identified as 2 different tokens which we don't wish to happen for better classification accuracy.

```
[31]: # tokenize each review
      # Sentence level tokenization, I am using Tweet Tokenizer because
      # if we use nltk.sent_tokenize, the words like didn't gets split, which we

      tknzr = TweetTokenizer()
      original_tokenized_review_sentences = []
      for each_review in sentences:
          # tokenize each review into words by splitting them into different sent
          sent_wordlevel=[tknzr.tokenize(sent) for sent in each_review]
          for each in sent_wordlevel:
              original_tokenized_review_sentences.append(each)

      print(len(original_tokenized_review_sentences))
      print(original_tokenized_review_sentences[-3:-1])
```

```
21629
[['I', 'took', 'it', 'to', 'ups', 'myself', 'and', 'paid', 'the', 'retur
n', 'with', '$', '100.00', 'insurance', 'secured', 'for', 'what', 'look',
'like', 'are', 'boots', 'worth', '$', '20', '.'], ["I'm", 'sorry', 'I',
'have', 'not', 'had', 'any', 'issues', 'ever', 'with', 'Amazon', 'cause',
'your', 'stuff', 'is', 'great', 'and', "I'm", 'always', 'happy', 'with',
'my', 'purchases', 'but', 'since', 'I', 'kept', 'my', 'receipts', 'on',
'the', 'returns', 'of', 'these', 'items', 'please', 'ask', 'your', '3rd',
'party', 'seller', 'what', 'happen', '!']]
```

After this step, my data is ready to be run against the classifier.

In this exercise, I have used the "bag of words" features where I collected all the words in the sentence_polarity corpus and selected some most frequent words to be the word features. I have described the detailed processing steps in the later sections of this report.
Also, I have tried different types of experiments in order to complete my assignment.
Filter by stop words and other pre-processing methods
Using a sentiment lexicon with scores or counts: Subjectivity
Representing Negation
Using the modified word_features inorder to give the preference to the adjectives! Since that is more important in sentiment classification. I have described different classifiers in the later sections.

I found out that the Subjectivity Classifier gave me the best accuracy amoung the rest of the classifiers, It came around **78.8%**
I have used NaiveBayes classifier to train and test the classifier on my feature sets. I have also tried to use the validate my results and found out the number of false positives, false negatives, true positives, true negatives in based on the result of my classifier.

Detailed Technique Description

Sentiment Analysis using Bag of words approach

While writing this classifier, I have started by loading the sentence_polarity corpus and created a list of documents where each document represents a single sentence with the words and its labels. First I created a list of documents where each document(sentence) is paired with it's label. Each item is a pair(sent, cat) where sent is a list of words from the sentence_polarity document and cat is its label, either 'pos' or 'neg'.
Since the documents are in order by label, I have used the random.shuffle() method to suffle the documents, for later separation into training and test sets.

After this, I have defined the set of words that will be used for features. This is all the words in the entire document collection. I have used top 2000 most frequent words from this set of words. After this step, I have defined the features for each document, using just the words, (Bag Of Words)/ Unigrams, and the value of the feature is a Boolean, according to whether the word is contained in that document.
Then I created a training and test sets, train a Naïve Bayes Classifier, and looked at the accuracy. And at this time, I have divided the documents to 90/10 split for test set and train set. This gave the the classifier accuracy of around 75.5%.

Sentiment Analysis

Sentiment Classification - Bag of words approach

In [118]:
```
sentences = sentence_polarity.sents()
print(sentence_polarity.categories())
documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]
```

In [119]:
```
documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]
```

In [120]:
```
random.shuffle(documents)
```

In [121]:
```
all_words_list = [word for (sent,cat) in documents for word in sent]
all_words = nltk.FreqDist(all_words_list)
word_items = all_words.most_common(2000)
word_features = [word for (word, freq) in word_items]
```

In [122]:
```
def document_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features
```

In [123]:
```
featuresets = [(document_features(d,word_features), c) for (d,c) in documents]
```

In [124]:
```
train_set, test_set = featuresets[1000:], featuresets[:1000]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print (nltk.classify.accuracy(classifier, test_set))
```

```
0.755
```

After this step, I have tried to experiment little more with different set of words while defining the word_features. I have chosen to take 3000 most frequent words this time. And tried to find the accuracy again. This time the Classifier accuracy improved to 76.6%. This was better than previously achieved accuracy.

```
# Trying with 3000 most frequent bag of words
sentences = sentence_polarity.sents()
print(sentence_polarity.categories())
documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]

random.shuffle(documents)

all_words_list = [word for (sent,cat) in documents for word in sent]
all_words = nltk.FreqDist(all_words_list)
word_items = all_words.most_common(3000)
word_features = [word for (word, freq) in word_items]

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

featuresets = [(document_features(d,word_features), c) for (d,c) in documents]

train_set, test_set = featuresets[1000:], featuresets[:1000]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print (nltk.classify.accuracy(classifier, test_set))
```

```
['neg', 'pos']
0.766
```

Additional Changes in the word_feature definition

I then tried to rerun the training and test sets and recalculated the document features by removing the stop words. This further improved my accuracy.
While removing the stop words, I simply didn't remove all the stop words. Because, if we closely review the stop words list, then we notice that stop words also has the important negation conveying words, like not, neither, etc. Which should be included in out word_features list.
For this, I have defined the separate list of negating_words and removed those words from the list of the stop words. Also, doing this is not enough. There are also few stop words ending with n't e.g.'aren't'. These words also are important wrt word_features, and should not be removed from the word_features. I have demonstrated this in the following code snippet.
New_stop_words is the list I used while finding the word_features, instead of using the original stop words list provided by nltk.

[209]:
```
# Stop Words
stop_words = set(stopwords.words('english'))
print(stop_words)

negationwords = ['no', 'not', 'never', 'none', 'nowhere', 'nothing', 'noone

neg_stop_words = []
print(type(stop_words))
for word in stop_words:
    if (word in negationwords) or (word.endswith("n't")):
        neg_stop_words.append(word)

#print(neg_stop_words)
#print(stop_words)
neg_stop_words = set(neg_stop_words)
new_stop_words = []
new_stop_words = list(stop_words - neg_stop_words)
```

```
{'each', 'above', 'again', 'but', 'mightn', 'then', 'in', 'herself', "i
t's", 'about', 'o', "you're", 'did', 'shouldn', 'himself', 'him', 'the',
'both', 'myself', 'at', "you'd", 'now', 'couldn', 'against', 'mustn', 'th
ey', 't', 'over', 'no', 'more', 'to', "she's", 'there', "shouldn't", 'v
e', 'during', 'through', "that'll", 'y', 'do', 'll', 'i', 'his', 'hers',
```

Also, in order to test that the remaining stop words in the new_stop_words list are not of prime importance wrt classification, I carried out further analysis of words like ‘very’, ‘too’, ‘again’.
I tried to check the pos_scores, neg_scores, and obj_scores of each of these words. I noticed that for each of these words, the obj_score is greater than pos_score or neg_score. Thus it is ok to remove these set of stop words while defining the word_features.
Please find the screen shot attached.

```
In [ ]: list(swn.senti_synsets("too"))
```

```
In [211]: breakdown3 = swn.senti_synset('besides.r.02')
          print(breakdown3.pos_score())
          print(breakdown3.neg_score())
          print(breakdown3.obj_score())

          0.0
          0.0
          1.0
```

```
In [212]: list(swn.senti_synsets("again"))

Out[212]: [SentiSynset('again.r.01')]
```

```
In [213]: breakdown3 = swn.senti_synset('again.r.01')
          print(breakdown3.pos_score())
          print(breakdown3.neg_score())
          print(breakdown3.obj_score())

          0.0
          0.0
          1.0
```

```
In [214]: list(swn.senti_synsets("very"))

Out[214]: [SentiSynset('very.s.01'),
           SentiSynset('identical.s.02'),
           SentiSynset('very.r.01'),
           SentiSynset('very.r.02')]
```

```
In [215]: breakdown3 = swn.senti_synset('very.s.01')
          print(breakdown3.pos_score())
          print(breakdown3.neg_score())
          print(breakdown3.obj_score())

          0.5
          0.0
          0.5
```

Now, after successfully identifying the stop words to be removed, I have modified the word_features to be defined based on the removal of the words from the new_stop_words list. Doing this, I found out that the accuracy of the classifier was improved to 76.7%, which was better than the previous results. Please find the processing screen shot below.

In [217]:
```
# recalculating Document Feature after removing stop words

sentences = sentence_polarity.sents()
print(sentence_polarity.categories())
documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]

random.shuffle(documents)

# all_word_list after removing the stop words
all_words_list = [word for (sent,cat) in documents for word in sent if word not in new_stop_words]
all_words = nltk.FreqDist(all_words_list)
word_items = all_words.most_common(3000)
word_features = [word for (word, freq) in word_items]

# bag of words approach
def document_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features

# define the feature sets using the document_features
featuresets = [(document_features(d,word_features), c) for (d,c) in documents]

# Train and test your model for accuracy
train_set, test_set = featuresets[1000:], featuresets[:1000]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print (nltk.classify.accuracy(classifier, test_set))
```

```
['neg', 'pos']
0.767
```

Subjective Count Features

In this approach, I first started with reading the subjectivity words from the subjectivity lexicon file. These words are often used as a feature themselves or in conjunction with some other information, I created two more features that involve counting the positive and negative subjectivity words present in the document. These features hold the counts of all the positive and negative subjective words, where each weakly subjective word is counted once and each strongly subjective word is counted twice. After doing this, I again constructed the new feature set and created the training and test sets for this new word_features and calculated the accuracy again! Now it turned out to be **78.5%,** which was higher than the previously defined classifier accuracy. Please find the code in the following screen shot. Note that the word_features used while using this classifier was one after removing the revised stop words list.

```
def SL_features(document, word_features, SL):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains(%s)' % 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)
    return features


#define the feature set for performinh the classification
# word features here is the revised word features after removing the stop words
SL_featuresets = [(SL_features(d, word_features, SL), c) for (d,c) in documents]

print(SL_featuresets[0][0]['positivecount'])
print(SL_featuresets[0][0]['negativecount'])

train_set, test_set = SL_featuresets[1000:], SL_featuresets[:1000]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print(nltk.classify.accuracy(classifier, test_set))
```

```
['strongsubj', 'adj', False, 'neutral']
3
4
0.785
```

Negation Features

The next classification technique I followed is the negation of opinions. There are different ways to handle these negative words ending with n't.
One way to deal with these words is to negate all the words after the negative word. Other technique is to negate the word following the negative word. I have tried using this technique here. I went through the document words in order adding the word_features, but if the word follows a negation words, then change the feature of to negated word.
Here is one list of negation words, including some adverbs called "approximate negators":
no, not, never, none, rather, hardly, scarcely, rarely, seldom, neither, nor, couldn't, wasn't, didn't, wouldn't, shouldn't, weren't, don't, doesn't, haven't, hasn't, won't, hadn't
I rerun the classifier by defining the word_features considering the negation words. Accuracy on randomly split sentences this time gave the accuracy of 78.3% again. Which is better again.

Sentiment Analysis - Negation Features

```
negationwords = ['no', 'not', 'never', 'none', 'nowhere', 'nothing', 'noone', 'rather', 'hardly', 'scarcely', 'rarely',

def NOT_features(document, word_features, negationwords):
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = False
        features['contains(NOT{})'.format(word)] = False
    # go through document words in order
    for 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 word_features)
        else:
            features['contains({})'.format(word)] = (word in word_features)
    return features
```

```
# this word_features is the list of word_features after removing the stop words
NOT_featuresets = [(NOT_features(d, word_features, negationwords), c) for (d, c) in documents]
NOT_featuresets[0][0]['contains(NOTlike)']
NOT_featuresets[0][0]['contains(always)']

train_set, test_set = NOT_featuresets[1000:], NOT_featuresets[:1000]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print(nltk.classify.accuracy(classifier, test_set))

classifier.show_most_informative_features(30)
```

```
0.783
Most Informative Features
          contains(warm) = True              pos : neg    =     19.8 : 1.0
    contains(engrossing) = True              pos : neg    =     19.1 : 1.0
       contains(generic) = True              neg : pos    =     16.3 : 1.0
       contains(routine) = True              neg : pos    =     14.9 : 1.0
          contains(flat) = True              neg : pos    =     14.5 : 1.0
      contains(mediocre) = True              neg : pos    =     14.3 : 1.0
      contains(supposed) = True              neg : pos    =     14.3 : 1.0
```

Since this was one of the highest accuracy I gained with the different classification techniques I have followed, I decided to categorize the reviews in the positive reviews and negative reviews based on this classifier.
I have created 2 files positive.txt and negative.txt, and I have run this classifier on the set of Amazon reviews I collected.

```
positive_file_handle = open("positive1.txt","w")
negative_file_handle = open("negative1.txt","w")

for i in range(len(tokenized_review_sentences)):
    sent, orig_sent = tokenized_review_sentences[i], original_tokenized_review_sentences[i]
    if((classifier.classify(document_features(sent,word_features))) == 'pos'):
        positive_file_handle.write(make_sentence(orig_sent))
        positive_file_handle.write("\n")
    else:
        negative_file_handle.write(make_sentence(orig_sent))
        negative_file_handle.write("\n")

positive_file_handle.close()
negative_file_handle.close()
```

Classifier defined based by making the use of POS tagging and other word features

While defining the word features for this classifier, I used following techniques. First I removed the revised stop words (as discussed in the previous sections), then I reviewed the word_features list and noticed that there are many words in this frequent words list, which are just the special characters. These characters make no sense in the most frequent word features. So I tried to remove those tuples from the list which have the special characters in them. Also, I tried to remove the words less than 4 character length to make the classifier more accurate.

I then realized that the word features in order to perform sentiment analysis must be dominated by the adjectives. E.g. 'Larry has a good dress.' We will identify this sentence as a positive sentiment based on the word good. How did our brain identify this as a positive sentiment? It was by looking at the adjective 'good'. Words like 'good', 'pleasant', play a very important role in sentiment analysis. We can also note that 'Larry' in this case is not really helping us to get the sentiment of this sentence.

Considering both these above points, I decided to remove the nouns, noun phrases from the word_features, so that we could focus on more adjectives than repeating nouns while classification. We can also try with verbs, adverbs, determinant removal, etc and check removal of which of these or not removal of these features yields better results.

Also, I tried to add few more important words like 'love', 'hate' which play important role in semantic analysis to this word_features list.

I also did stemming, because the words like working, worked, work should not be considered different while defining the classifier. Please find it in following screen shots.

Due to randomness of the split of the documents, I couldn't get better accuracy. I could get 73% accuracy when I performed this type of word_feature classification. This accuracy can be further improved by closely looking at the word_features, and identifying the word_features which reduces the classifier accuracy, or by adding new features to this list which are of more importance.

In [344]:

```
# Defining classifier, based on only adjectives and exclaimatory marks

sentences = sentence_polarity.sents()
print(sentence_polarity.categories())
documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]

random.shuffle(documents)

# all_word_list after removing the stop words
all_words_list = [word for (sent,cat) in documents for word in sent if word not in new_stop_words]

all_words = nltk.FreqDist(all_words_list)
word_items = all_words.most_common(15000)

# we can see that all_words_list currently has all the words like nouns
# adjectives, pronouns, exclaimation marks, proper nouns, etc
# print(word_items)

# Since we know that classification of sentiment is purely based on the adjectives
# negative words, etc

# lets remove all the special symbols from the list

print(type(word_items))
word_dict = dict(word_items)
print(type(word_dict))

# remove all non - alpha characters
pattern = "[^A-Za-z]+"
prog = re.compile(pattern)
refined_word_dict = dict()
for key in word_dict:
    if (prog.match(key)):
        # .,"--)?(:';!-•2*[a]902002 all such words are removed from the most frequent words
        if (len(key)<=5):
            print("removing tuple with key "+ key)
        else:
            value = word_dict[key]
        refined_word_dict[key] = value
```

```
# adding 2 important words externally to the word_dict
refined_word_dict['love'] = 100
refined_word_dict['hate'] = 100

print("Len After removing special symbols "+str(len(refined_word_dict)))
word_dict = dict()

# removing nouns, determinants from the list

for key in refined_word_dict:
    word_tokens = []
    word_tokens.append(key)
    tag = nltk.pos_tag(word_tokens)
    tag = tag[0][1]
    if(tag == 'NN' or tag == 'NNS'):
        # print("removing Noun tuple with key "+ key)
        continue
    #elif (tag == 'VB' or tag == 'VBD' or tag == 'VBN' or tag == 'VBP' or tag == 'VBZ' ):
        # print("removing Verb tuple with key " + key)
    #    continue
    else:
        value = refined_word_dict[key]
        word_dict[key] = value
        # print(tag)

print("Len after removing nouns and verbs " + str(len(word_dict)))
# print(word_dict)
import copy

word_dict_copy = copy.copy(word_dict)

# stemming
ps = PorterStemmer()
for key in word_dict_copy:
    new_key = key
    new_key = ps.stem(new_key)
    new_value = word_dict_copy[key]
    word_dict[new_key] = new_value

# converting dictionary to list again
word_items = word_dict.items()
```

```
# stemming
ps = PorterStemmer()
for key in word_dict_copy:
    new_key = key
    new_key = ps.stem(new_key)
    new_value = word_dict_copy[key]
    word_dict[new_key] = new_value

# converting dictionary to list again
word_items = word_dict.items()
word_features = [word for (word, freq) in word_items]



# print(word_features)
```

```
removing tuple with key é
removing tuple with key 5
removing tuple with key 40
removing tuple with key 'i
removing tuple with key 'it's
removing tuple with key 3
removing tuple with key 3000
removing tuple with key 1975
removing tuple with key 95
removing tuple with key 50
removing tuple with key 86
removing tuple with key 1960s
removing tuple with key 84
removing tuple with key 'what
removing tuple with key ]
```

After trying different classifiers, I have tried to use the evaluation measure of the predictive capability of the model that was learned from the training data. We can learn more by looking at the predictions for each of the labels in our classifier.

In nltk the confusion matrix is given by a function that takes two lists of labels for the test sets. First list called as a reference list which is all the correct/gold labels for the test set. Second list is the test list, which are the predicted labels in the test set.

Using this, I found out the number of true positives, true negatives, false positives and false positives, respectively.

```
dir(nltk.metrics)
from nltk.metrics import *
```

```
reflist = []
testlist = []
for (features, label) in test_set:
    reflist.append(label)
    testlist.append(classifier.classify(features))
```

```
cm = ConfusionMatrix(reflist, testlist)
print(cm)
```

```
     |   n   p |
     |   e   o |
     |   g   s |
-----+---------+
neg  |<358>146 |
pos  | 125<371>|
-----+---------+
(row = reference; col = test)
```

Bonus Credit (10%)

I have chosen an additional, more advanced type of the task from the list. I have tried to use the SciKit Learn Classifier with the features produced in nltk. By doing this, I observed that I could get 73% accuracy. I also tried to find out the true negative, false positive, false negative, false positive using the Confusion_Matrix provided in the sklearn.metrics package. Please find the screen shot attached.

In [442]:
```
from sklearn.svm import LinearSVC
from nltk.classify.scikitlearn import SklearnClassifier
classifier=nltk.classify.SklearnClassifier(LinearSVC()).train(train_set)
nltk.classify.accuracy(classifier, test_set)
```

Out[442]: 0.729

In [372]:
```
# calculating true negative, false positive, false negative, true positive
from sklearn.metrics import confusion_matrix
tn, fp, fn, tp = confusion_matrix(reflist, testlist).ravel()
(tn, fp, fn, tp)
```

Out[372]: (353, 151, 119, 377)

Appendix

Screen Shots of code and python processing screen shots

```
[31]: # tokenize each review
      # Sentence level tokenization, I am using Tweet Tokenizer because
      # if we use nltk.sent_tokenize, the words like didn't gets split, which we

      tknzr = TweetTokenizer()
      original_tokenized_review_sentences = []
      for each_review in sentences:
          # tokenize each review into words by splitting them into different sent
          sent_wordlevel=[tknzr.tokenize(sent) for sent in each_review]
          for each in sent_wordlevel:
              original_tokenized_review_sentences.append(each)

      print(len(original_tokenized_review_sentences))
      print(original_tokenized_review_sentences[-3:-1])
```

```
21629
[['I', 'took', 'it', 'to', 'ups', 'myself', 'and', 'paid', 'the', 'retur
n', 'with', '$', '100.00', 'insurance', 'secured', 'for', 'what', 'look',
'like', 'are', 'boots', 'worth', '$', '20', '.'], ["I'm", 'sorry', 'I',
'have', 'not', 'had', 'any', 'issues', 'ever', 'with', 'Amazon', 'cause',
'your', 'stuff', 'is', 'great', 'and', "I'm", 'always', 'happy', 'with',
'my', 'purchases', 'but', 'since', 'I', 'kept', 'my', 'receipts', 'on',
'the', 'returns', 'of', 'these', 'items', 'please', 'ask', 'your', '3rd',
'party', 'seller', 'what', 'happen', '!']]
```

Sentiment Analysis

Sentiment Classification - Bag of words approach

In [118]:
```
sentences = sentence_polarity.sents()
print(sentence_polarity.categories())
documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]
```

In [119]:
```
documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]
```

In [120]:
```
random.shuffle(documents)
```

In [121]:
```
all_words_list = [word for (sent,cat) in documents for word in sent]
all_words = nltk.FreqDist(all_words_list)
word_items = all_words.most_common(2000)
word_features = [word for (word, freq) in word_items]
```

In [122]:
```
def document_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features
```

In [123]:
```
featuresets = [(document_features(d,word_features), c) for (d,c) in documents]
```

In [124]:
```
train_set, test_set = featuresets[1000:], featuresets[:1000]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print (nltk.classify.accuracy(classifier, test_set))
```

```
0.755
```

```
# Trying with 3000 most frequent bag of words
sentences = sentence_polarity.sents()
print(sentence_polarity.categories())
documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]

random.shuffle(documents)

all_words_list = [word for (sent,cat) in documents for word in sent]
all_words = nltk.FreqDist(all_words_list)
word_items = all_words.most_common(3000)
word_features = [word for (word, freq) in word_items]

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

featuresets = [(document_features(d,word_features), c) for (d,c) in documents]

train_set, test_set = featuresets[1000:], featuresets[:1000]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print (nltk.classify.accuracy(classifier, test_set))
```

```
['neg', 'pos']
0.766
```

[209]:
```
# Stop Words
stop_words = set(stopwords.words('english'))
print(stop_words)

negationwords = ['no', 'not', 'never', 'none', 'nowhere', 'nothing', 'noone

neg_stop_words = []
print(type(stop_words))
for word in stop_words:
    if (word in negationwords) or (word.endswith("n't")):
        neg_stop_words.append(word)

#print(neg_stop_words)
#print(stop_words)
neg_stop_words = set(neg_stop_words)
new_stop_words = []
new_stop_words = list(stop_words - neg_stop_words)
```

```
{'each', 'above', 'again', 'but', 'mightn', 'then', 'in', 'herself', "i
t's", 'about', 'o', "you're", 'did', 'shouldn', 'himself', 'him', 'the',
'both', 'myself', 'at', "you'd", 'now', 'couldn', 'against', 'mustn', 'th
ey', 't', 'over', 'no', 'more', 'to', "she's", 'there', "shouldn't", 'v
e', 'during', 'through', "that'll", 'y', 'do', 'll', 'i', 'his', 'hers',
```

In [ ]:
```
list(swn.senti_synsets("too"))
```

In [211]:
```
breakdown3 = swn.senti_synset('besides.r.02')
print(breakdown3.pos_score())
print(breakdown3.neg_score())
print(breakdown3.obj_score())
```

```
0.0
0.0
1.0
```

In [212]:
```
list(swn.senti_synsets("again"))
```

Out[212]:
```
[SentiSynset('again.r.01')]
```

In [213]:
```
breakdown3 = swn.senti_synset('again.r.01')
print(breakdown3.pos_score())
print(breakdown3.neg_score())
print(breakdown3.obj_score())
```

```
0.0
0.0
1.0
```

In [214]:
```
list(swn.senti_synsets("very"))
```

Out[214]:
```
[SentiSynset('very.s.01'),
 SentiSynset('identical.s.02'),
 SentiSynset('very.r.01'),
 SentiSynset('very.r.02')]
```

In [215]:
```
breakdown3 = swn.senti_synset('very.s.01')
print(breakdown3.pos_score())
print(breakdown3.neg_score())
print(breakdown3.obj_score())
```

```
0.5
0.0
0.5
```

In [217]:
```
# recalculating Document Feature after removing stop words

sentences = sentence_polarity.sents()
print(sentence_polarity.categories())
documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]

random.shuffle(documents)

# all_word_list after removing the stop words
all_words_list = [word for (sent,cat) in documents for word in sent if word not in new_stop_words]
all_words = nltk.FreqDist(all_words_list)
word_items = all_words.most_common(3000)
word_features = [word for (word, freq) in word_items]

# bag of words approach
def document_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features

# define the feature sets using the document_features
featuresets = [(document_features(d,word_features), c) for (d,c) in documents]

# Train and test your model for accuracy
train_set, test_set = featuresets[1000:], featuresets[:1000]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print (nltk.classify.accuracy(classifier, test_set))
```

```
['neg', 'pos']
0.767
```

```
def SL_features(document, word_features, SL):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains(%s)' % 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)
    return features


#define the feature set for performinh the classification
# word features here is the revised word features after removing the stop words
SL_featuresets = [(SL_features(d, word_features, SL), c) for (d,c) in documents]

print(SL_featuresets[0][0]['positivecount'])
print(SL_featuresets[0][0]['negativecount'])

train_set, test_set = SL_featuresets[1000:], SL_featuresets[:1000]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print(nltk.classify.accuracy(classifier, test_set))
```

```
['strongsubj', 'adj', False, 'neutral']
3
4
0.785
```

Sentiment Analysis - Negation Features

```
negationwords = ['no', 'not', 'never', 'none', 'nowhere', 'nothing', 'noone', 'rather', 'hardly', 'scarcely', 'rarely',

def NOT_features(document, word_features, negationwords):
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = False
        features['contains(NOT{})'.format(word)] = False
    # go through document words in order
    for 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 word_features)
        else:
            features['contains({})'.format(word)] = (word in word_features)
    return features
```

```
# this word_features is the list of word_features after removing the stop words
NOT_featuresets = [(NOT_features(d, word_features, negationwords), c) for (d, c) in documents]
NOT_featuresets[0][0]['contains(NOTlike)']
NOT_featuresets[0][0]['contains(always)']

train_set, test_set = NOT_featuresets[1000:], NOT_featuresets[:1000]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print(nltk.classify.accuracy(classifier, test_set))

classifier.show_most_informative_features(30)
```

```
0.783
Most Informative Features
          contains(warm) = True              pos : neg    =     19.8 : 1.0
    contains(engrossing) = True              pos : neg    =     19.1 : 1.0
       contains(generic) = True              neg : pos    =     16.3 : 1.0
       contains(routine) = True              neg : pos    =     14.9 : 1.0
          contains(flat) = True              neg : pos    =     14.5 : 1.0
      contains(mediocre) = True              neg : pos    =     14.3 : 1.0
      contains(supposed) = True              neg : pos    =     14.3 : 1.0
```

```
positive_file_handle = open("positive1.txt","w")
negative_file_handle = open("negative1.txt","w")

for i in range(len(tokenized_review_sentences)):
    sent, orig_sent = tokenized_review_sentences[i], original_tokenized_review_sentences[i]
    if((classifier.classify(document_features(sent,word_features))) == 'pos'):
        positive_file_handle.write(make_sentence(orig_sent))
        positive_file_handle.write("\n")
    else:
        negative_file_handle.write(make_sentence(orig_sent))
        negative_file_handle.write("\n")

positive_file_handle.close()
negative_file_handle.close()
```

In [344]:

```
# Defining classifier, based on only adjectives and exclaimatory marks

sentences = sentence_polarity.sents()
print(sentence_polarity.categories())
documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]

random.shuffle(documents)

# all_word_list after removing the stop words
all_words_list = [word for (sent,cat) in documents for word in sent if word not in new_stop_words]

all_words = nltk.FreqDist(all_words_list)
word_items = all_words.most_common(15000)

# we can see that all_words_list currently has all the words like nouns
# adjectives, pronouns, exclaimation marks, proper nouns, etc
# print(word_items)

# Since we know that classification of sentiment is purely based on the adjectives
# negative words, etc

# lets remove all the special symbols from the list

print(type(word_items))
word_dict = dict(word_items)
print(type(word_dict))

# remove all non - alpha characters
pattern = "[^A-Za-z]+"
prog = re.compile(pattern)
refined_word_dict = dict()
for key in word_dict:
    if (prog.match(key)):
        # .,"--)?(:';!-•2*[a]902002 all such words are removed from the most frequent words
        if (len(key)<=5):
            print("removing tuple with key "+ key)
        else:
            value = word_dict[key]
        refined_word_dict[key] = value
```

```
# adding 2 important words externally to the word_dict
refined_word_dict['love'] = 100
refined_word_dict['hate'] = 100

print("Len After removing special symbols "+str(len(refined_word_dict)))
word_dict = dict()

# removing nouns, determinants from the list

for key in refined_word_dict:
    word_tokens = []
    word_tokens.append(key)
    tag = nltk.pos_tag(word_tokens)
    tag = tag[0][1]
    if(tag == 'NN' or tag == 'NNS'):
        # print("removing Noun tuple with key "+ key)
        continue
    #elif (tag == 'VB' or tag == 'VBD' or tag == 'VBN' or tag == 'VBP' or tag == 'VBZ' ):
        # print("removing Verb tuple with key " + key)
    #    continue
    else:
        value = refined_word_dict[key]
        word_dict[key] = value
        # print(tag)

print("Len after removing nouns and verbs " + str(len(word_dict)))
# print(word_dict)
import copy

word_dict_copy = copy.copy(word_dict)

# stemming
ps = PorterStemmer()
for key in word_dict_copy:
    new_key = key
    new_key = ps.stem(new_key)
    new_value = word_dict_copy[key]
    word_dict[new_key] = new_value

# converting dictionary to list again
word_items = word_dict.items()
```

```
# stemming
ps = PorterStemmer()
for key in word_dict_copy:
    new_key = key
    new_key = ps.stem(new_key)
    new_value = word_dict_copy[key]
    word_dict[new_key] = new_value

# converting dictionary to list again
word_items = word_dict.items()
word_features = [word for (word, freq) in word_items]



# print(word_features)
```

```
removing tuple with key é
removing tuple with key 5
removing tuple with key 40
removing tuple with key 'i
removing tuple with key 'it's
removing tuple with key 3
removing tuple with key 3000
removing tuple with key 1975
removing tuple with key 95
removing tuple with key 50
removing tuple with key 86
removing tuple with key 1960s
removing tuple with key 84
removing tuple with key 'what
removing tuple with key ]
removing tuple with key 3d
```

In [ ]:
```
list(swn.senti_synsets("too"))
```

In [211]:
```
breakdown3 = swn.senti_synset('besides.r.02')
print(breakdown3.pos_score())
print(breakdown3.neg_score())
print(breakdown3.obj_score())
```

```
0.0
0.0
1.0
```

In [212]:
```
list(swn.senti_synsets("again"))
```

Out[212]:
```
[SentiSynset('again.r.01')]
```

In [213]:
```
breakdown3 = swn.senti_synset('again.r.01')
print(breakdown3.pos_score())
print(breakdown3.neg_score())
print(breakdown3.obj_score())
```

```
0.0
0.0
1.0
```

In [214]:
```
list(swn.senti_synsets("very"))
```

Out[214]:
```
[SentiSynset('very.s.01'),
 SentiSynset('identical.s.02'),
 SentiSynset('very.r.01'),
 SentiSynset('very.r.02')]
```

In [215]:
```
breakdown3 = swn.senti_synset('very.s.01')
print(breakdown3.pos_score())
print(breakdown3.neg_score())
print(breakdown3.obj_score())
```

```
0.5
0.0
0.5
```

In [217]:
```
# recalculating Document Feature after removing stop words

sentences = sentence_polarity.sents()
print(sentence_polarity.categories())
documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]

random.shuffle(documents)

# all_word_list after removing the stop words
all_words_list = [word for (sent,cat) in documents for word in sent if word not in new_stop_words]
all_words = nltk.FreqDist(all_words_list)
word_items = all_words.most_common(3000)
word_features = [word for (word, freq) in word_items]

# bag of words approach
def document_features(document, word_features):
    document_words = set(document)
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = (word in document_words)
    return features

# define the feature sets using the document_features
featuresets = [(document_features(d,word_features), c) for (d,c) in documents]

# Train and test your model for accuracy
train_set, test_set = featuresets[1000:], featuresets[:1000]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print (nltk.classify.accuracy(classifier, test_set))
```

```
['neg', 'pos']
0.767
```

Sentiment Analysis - Negation Features

```
negationwords = ['no', 'not', 'never', 'none', 'nowhere', 'nothing', 'noone', 'rather', 'hardly', 'scarcely', 'rarely',

def NOT_features(document, word_features, negationwords):
    features = {}
    for word in word_features:
        features['contains({})'.format(word)] = False
        features['contains(NOT{})'.format(word)] = False
    # go through document words in order
    for 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 word_features)
        else:
            features['contains({})'.format(word)] = (word in word_features)
    return features
```

```
# this word_features is the list of word_features after removing the stop words
NOT_featuresets = [(NOT_features(d, word_features, negationwords), c) for (d, c) in documents]
NOT_featuresets[0][0]['contains(NOTlike)']
NOT_featuresets[0][0]['contains(always)']

train_set, test_set = NOT_featuresets[1000:], NOT_featuresets[:1000]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print(nltk.classify.accuracy(classifier, test_set))

classifier.show_most_informative_features(30)
```

```
0.783
Most Informative Features
          contains(warm) = True              pos : neg    =     19.8 : 1.0
    contains(engrossing) = True              pos : neg    =     19.1 : 1.0
       contains(generic) = True              neg : pos    =     16.3 : 1.0
       contains(routine) = True              neg : pos    =     14.9 : 1.0
          contains(flat) = True              neg : pos    =     14.5 : 1.0
      contains(mediocre) = True              neg : pos    =     14.3 : 1.0
      contains(supposed) = True              neg : pos    =     14.3 : 1.0
```

```
positive_file_handle = open("positive.txt","w")
negative_file_handle = open("negative.txt","w")

for i in range(len(tokenized_review_sentences)):
    sent, orig_sent = tokenized_review_sentences[i], original_tokenized_review_sentences[i]
    if((classifier.classify(document_features(sent,word_features))) == 'pos'):
        positive_file_handle.write(make_sentence(orig_sent))
        positive_file_handle.write("\n")
    else:
        negative_file_handle.write(make_sentence(orig_sent))
        negative_file_handle.write("\n")

positive_file_handle.close()
negative_file_handle.close()
```

In [344]:

```
# Defining classifier, based on only adjectives and exclaimatory marks

sentences = sentence_polarity.sents()
print(sentence_polarity.categories())
documents = [(sent, cat) for cat in sentence_polarity.categories()
    for sent in sentence_polarity.sents(categories=cat)]

random.shuffle(documents)

# all_word_list after removing the stop words
all_words_list = [word for (sent,cat) in documents for word in sent if word not in new_stop_words]

all_words = nltk.FreqDist(all_words_list)
word_items = all_words.most_common(15000)

# we can see that all_words_list currently has all the words like nouns
# adjectives, pronouns, exclaimation marks, proper nouns, etc
# print(word_items)

# Since we know that classification of sentiment is purely based on the adjectives
# negative words, etc

# lets remove all the special symbols from the list

print(type(word_items))
word_dict = dict(word_items)
print(type(word_dict))

# remove all non - alpha characters
pattern = "[^A-Za-z]+"
prog = re.compile(pattern)
refined_word_dict = dict()
for key in word_dict:
    if (prog.match(key)):
        # .,"--)?(:';!-•2*[a]902002 all such words are removed from the most frequent words
        if (len(key)<=5):
            print("removing tuple with key "+ key)
        else:
            value = word_dict[key]
        refined_word_dict[key] = value
```

```
# adding 2 important words externally to the word_dict
refined_word_dict['love'] = 100
refined_word_dict['hate'] = 100

print("Len After removing special symbols "+str(len(refined_word_dict)))
word_dict = dict()

# removing nouns, determinants from the list

for key in refined_word_dict:
    word_tokens = []
    word_tokens.append(key)
    tag = nltk.pos_tag(word_tokens)
    tag = tag[0][1]
    if(tag == 'NN' or tag == 'NNS'):
        # print("removing Noun tuple with key "+ key)
        continue
    #elif (tag == 'VB' or tag == 'VBD' or tag == 'VBN' or tag == 'VBP' or tag == 'VBZ' ):
        # print("removing Verb tuple with key " + key)
    #    continue
    else:
        value = refined_word_dict[key]
        word_dict[key] = value
        # print(tag)

print("Len after removing nouns and verbs " + str(len(word_dict)))
# print(word_dict)
import copy

word_dict_copy = copy.copy(word_dict)

# stemming
ps = PorterStemmer()
for key in word_dict_copy:
    new_key = key
    new_key = ps.stem(new_key)
    new_value = word_dict_copy[key]
    word_dict[new_key] = new_value

# converting dictionary to list again
word_items = word_dict.items()
```

```
# stemming
ps = PorterStemmer()
for key in word_dict_copy:
    new_key = key
    new_key = ps.stem(new_key)
    new_value = word_dict_copy[key]
    word_dict[new_key] = new_value

# converting dictionary to list again
word_items = word_dict.items()
word_features = [word for (word, freq) in word_items]



# print(word_features)
```

```
removing tuple with key é
removing tuple with key 5
removing tuple with key 40
removing tuple with key 'i
removing tuple with key 'it's
removing tuple with key 3
removing tuple with key 3000
removing tuple with key 1975
removing tuple with key 95
removing tuple with key 50
removing tuple with key 86
removing tuple with key 1960s
removing tuple with key 84
removing tuple with key 'what
removing tuple with key ]
```

```
dir(nltk.metrics)
from nltk.metrics import *
```

```
reflist = []
testlist = []
for (features, label) in test_set:
    reflist.append(label)
    testlist.append(classifier.classify(features))
```

```
cm = ConfusionMatrix(reflist, testlist)
print(cm)
```

```
     |   n   p |
     |   e   o |
     |   g   s |
----+---------+
neg |<358>146 |
pos | 125<371>|
----+---------+
(row = reference; col = test)
```

In [442]:
```
from sklearn.svm import LinearSVC
from nltk.classify.scikitlearn import SklearnClassifier
classifier=nltk.classify.SklearnClassifier(LinearSVC()).train(train_set)
nltk.classify.accuracy(classifier, test_set)
```

Out[442]: 0.729

In [372]:
```
# calculating true negative, false positive, false negative, true positive
from sklearn.metrics import confusion_matrix
tn, fp, fn, tp = confusion_matrix(reflist, testlist).ravel()
(tn, fp, fn, tp)
```

Out[372]: (353, 151, 119, 377)

negative.txt

We bought several tutus at once , and they are got high reviews .
Sturdy and seemingly well-made .
The girls have been wearing them regularly , including out to play , and the tutus have stood up well .
Clearly plenty of room to grow .
But this is not difficult .
Thank you Halo Heaven great product for Little Girls .
Will buy more from this seller .
Made well and cute on the girls .
Thanks for a great product.NEVER BUY FROM DRESS UP DREAMS ... I will buy more as long as I don't buy from " Dress Up Dreams " I never rec'd or order in FL .
Only rec'd pink , the purple one was missing .
Company is a rip off .
I received this today and I'm not a fan of it but my daughter is I thought it would be puffier as it looks in the pic but it's not and the one they sent me is pink underneath and the waist band is pink which is not what I wanted due to the fact she already had the sandals she was gonna wear with it now I gotta find another pair of sandals , ima just keep it cuz she likes it .
She fits a 4T and it's just above her knee , and has a little bit of growing room , although I'm not so sure as much as others are saying .
But considering how often she wears it , I'm not worried !
;)
Was hoping to order more in different colors.She had hardly used this , the stitching came apart in 2weeks.now it's lying in her closet .. Altogether she wore it like 4-5 times for 20 mins or so.wish the stitching was of better quality to hold up while little ones wear it .
Can't recommend .
Never GOT this item - but gave a 1 STAR because the replies from the SUPPLIER was

positive.txt

Fits the 3 - yr old & the 5 - yr old well .
Only con is that when the kids pull off the tutus , the waste band gets twisted , and an adult has to un-tangle .
My Great Grand Daughters Love these Tutu's .
REFUSES to make good on purchase ... Real creeps .
My daughter has worn this skirt almost every day since she received it and it's even been through the washer along with the other clothes .
It's amazing quality !
My 3yr old loved this tutu skirt in pink !
Love this Jewelry Box so well put together holds plendy ... Love the pink & looks so nice on my vanity .
My granddaughter likes the Shining Image Jewelry Box given to her for a birthday gift from Grandma .
Also , I wish the drawers had a stop , preventing the drawers from tipping contents entirely when trying to remove one piece of jewelry .
I , on the other hand , have had lacks of years of practice with jewelry drawers , but also struggle with removing the drawer without spilling the contents .
I bought one of these neck stashes for my most recent cruise and used it everyday while on vacation .
I stuffed cash , my passport , driver's license and several credit cards in it and still had some room to spare .
This travel pouch is just as described .
It fits effortlessly under a shirt and holds all your valuables .
My husband won't use it because it feels like a purse to him , but that is just because he is ultra manly .
When I travel out of country , I have this around my neck and under one layer of clothing .
Again though perhaps being flimsy is a good thing being under clothing making it conceal more .