Sentiment Analysis - Amazon Reviews

Introduction

1.Dataset

2.Data Pre-processing

3.Sentiment Analysis

A. Sentiment Classification – Words as Features

B. Sentiment Classification - Subjectivity Count features

C. Negation features

1. Dataset

In this problem, you will analyze the review contents from Amazon Product Data provided by Julian McAuley at http://jmcauley.ucsd.edu/data/amazon/. This dataset contains product reviews and metadata from Amazon, including 142.8 million reviews spanning May 1996 – July 2014. It includes reviews (ratings, text, helpfulness votes), product metadata (descriptions, category information, price, brand, and image features), and links (also viewed/also bought graphs). For our tasks, we will use only 5-core subsets of three categories (Baby / Clothing, Shoes and Jewelry / Health and Personal Care). 5-core subsets mean that all users and items in the dataset have at least 5 reviews. Originally, the dataset was a zipped file of json format and the content was arranged in dictionaries. For your convenience, the dataset was modified into text file and is available for download in the Assignment folder in the course web site: Baby.txt.

Code:

```
import nltk
import re
from nltk.corpus import PlaintextCorpusReader
from nltk.tokenize import TweetTokenizer
from nltk.corpus import sentence_polarity
import random
from nltk.corpus import wordnet as wn
from nltk.corpus import sentiwordnet as swn
```

```
from nltk.corpus import stopwords
from nltk.corpus import subjectivity
from nltk.stem import PorterStemmer
from nltk.metrics import precision
from nltk.tokenize import sent_tokenize, word_tokenize
from nltk.metrics import ConfusionMatrix

reviewdata=open('C:/Users/Harsh Darji/Desktop/baby.txt').readlines()
```

2. Data Pre-processing (20%) You will write a Python code that extracts only review texts. Please submit the sample screenshot of the output (included in your report file).

Code:

```
reviewTextContent=[]
reviewYear=[]
pattern = r'''(?x)
   reviewText[:](.+)
   '''
year_pattern = r''' (?x)
  reviewTime[:](.+)
'''
for line in reviewdata:

   sentlist = []
   reviewyear_tokens = nltk.regexp_tokenize(line,year_pattern)
   reviewtext_tokens = nltk.regexp_tokenize(line,pattern)

   if(len(reviewtext_tokens)>0):
      current_review = reviewtext_tokens[0].strip("\n")
      reviewTextContent.append(current_review)

   if(len(reviewyear_tokens)>0):
      current_year = reviewyear_tokens[0].split(',')[1].strip('\n').strip(' ')
      reviewYear.append(current_year)
finaldata=[]
for i in range(0, len(reviewYear)):
   if(reviewYear[i]== '2012'):
      finaldata.append(reviewTextContent[i])
```

Output:

```
reviewTextContent=[]
reviewYear=[]
```

```
pattern = r'''(?x)
    reviewText[:](.+)
    '''
year_pattern = r''' (?x)
    reviewTime[:](.+)
'''
```

```
for line in reviewdata:

    sentlist = []
    reviewyear_tokens = nltk.regexp_tokenize(line,year_pattern)
    reviewtext_tokens = nltk.regexp_tokenize(line,pattern)

    if(len(reviewtext_tokens)>0):
        current_review = reviewtext_tokens[0].strip("\n")
        reviewTextContent.append(current_review)

    if(len(reviewyear_tokens)>0):
        current_year = reviewyear_tokens[0].split(',')[1].strip('\n').strip(' ')
        reviewYear.append(current_year)
```

```
finaldata=[]
for i in range(0, len(reviewYear)):
    if(reviewYear[i]== '2012'):
        finaldata.append(reviewTextContent[i])
```

```
print("Total no of reviews",len(finaldata))
```

```
Total no of reviews 27587
```

Explaination:

In this assignment, I have extracted reviews from baby.txt using regular expression from the nltk library along with the use of tokenization. As data was too big and It takes lot of time to analyze the entire dataset I have taken reviews from year 2012. There are total 27587 reviews for the year 2012 and I have performed my sentiment analysis on the same.

Code:

```
# Sentence level tokenization
sentences=[nltk.sent_tokenize(sent) for sent in finaldata]
# 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])
# 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
# and lower_case_sentences has the
for sentence in original_tokenized_review_sentences:
  tokenized_review_sentences.append([item.lower() for item in sentence])

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

Output:

```
# Sentence level tokenization
sentences=[nltk.sent_tokenize(sent) for sent in finaldata]
```

```
# 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])
```

```
173865
[['I', 'should', 'have', 'bought', 'hemp', 'inserts', 'from', 'the', 'beginning', '.'], ['They', "don't", 'stink', 'like', 'th
e', 'microfiber', 'ones', '.']]
```

```
# 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
# and lower_case_sentences has the
for sentence in original_tokenized_review_sentences:
    tokenized_review_sentences.append([item.lower() for item in sentence])

print(tokenized_review_sentences[-3:-1])
```

```
[['i', 'should', 'have', 'bought', 'hemp', 'inserts', 'from', 'the', 'beginning', '.'], ['they', "don't", 'stink', 'like', 'th
e', 'microfiber', 'ones', '.']]
```

Anaconda Prompt - JUPYTER NOTEBOOK

```
print(len(tokenized_review_sentences))
```

```
173865
```

Explaination:

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. Length of my tokens is 173865

Code:

```
# Stop Words
stop_words = set(stopwords.words('english'))
print(stop_words)

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

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)
```

Output:

```
# Stop Words
stop_words = set(stopwords.words('english'))
print(stop_words)

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

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)
```

```
{'do', 'but', 'for', 're', 'yourself', 'we', "shan't", 'out', 'very', 'his', 'after', "doesn't", 'into', 'be', 'd', "weren't",
'these', 'aren', 'now', 'being', "needn't", 'which', 'nor', 'had', 'when', 'more', 'theirs', 'hasn', 'ours', 't', 'what', "sh
e's", 'because', 'once', 'both', 'ain', 'itself', 'in', 'so', 'hers', 'wouldn', 'does', 'at', "isn't", 'they', 'own', 'is', 'h
erself', 'only', 'there', 'yourselves', "shouldn't", 'of', "wasn't", 'mightn', 'will', "haven't", 'with', 'ma', 'above', 'tha
t', 'wasn', 'can', 'under', 'few', 'was', 'she', 'a', 'its', 'hadn', 'isn', 'why', "mustn't", 'each', 'any', 'won', 'my', 'yo
u', 'did', 'while', 'has', 'were', 'an', 'or', 'then', "should've", 'himself', "won't", 'just', 'as', 'those', 'below', 'agai
n', "wouldn't", 've', 'are', 'off', "hasn't", 'if', "it's", 'don', 'doesn', 'to', 'no', 'been', 'shan', "don't", 'and', 'he',
'here', 'him', "you've", 's', 'couldn', 'further', 'shouldn', 'such', 'should', 'most', 'through', 'our', 'mightn't", 'doing',
'not', 'too', 'ourselves', 'o', 'against', 'by', 'haven', 'this', 'the', 'up', 'yours', "you're", 'it', 'before', 'some', 'who
m', 'where', "didn't", 'i', 'm', 'how', "you'd", 'me', 'myself', 'y', 'her', 'other', 'who', 'from', "that'll", 'having', 'abo
ut', 'over', 'needn', "aren't", 'mustn', 'am', "you'll", "hadn't", 'them', 'all', 'have', "couldn't", 'until', 'same', 'were
n', 'during', 'your', 'didn', 'down', 'between', 'than', 'on', 'll', 'their', 'themselves'}
<class 'set'>
```

Explaination:

In this step, I have identified and excluded stop words. Both positive and negative stop words are dealt with so that are classifier is more accurate**.**

3.Sentiment Analysis (80%)

A. Classsification-Bag of Words

Code:

```
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)]
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(2000)
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))
```

Output:

```
# SENTIMENT ANALYSIS # Classsification-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)]
```

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

```
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(2000)
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))
```

```
0.732
```

Explaination:

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 73.2%.

Code:

```
# 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))
```

Output:

```
# 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']
0.785
```

Explaination:

Here, 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 3000 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 78.5%.

Here the accuracy increases by 3% when I use more frequent bag of words.

Code:

```
# 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))
```

Output:

```
# 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']
0.774
```

Explaination:

Here, 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.

After successfully identifying the stop words to be removed The accuracy obtained is 77.4%

B. Subjectivity count features

Code:

```
SLpath = 'subjclueslen1-HLTEMNLP05.tff'
SL = readSubjectivity(SLpath)
print(SL['absolute'])

# define the features, to find out the feature_set
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))
```

Output:

```
SLpath = 'subjclueslen1-HLTEMNLP05.tff'
SL = readSubjectivity(SLpath)
print(SL['absolute'])
```

```
['strongsubj', 'adj', False, 'neutral']
```

```
# define the features, to find out the feature_set
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))
```

```
7
0
0.778
```

Explaination:

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.

Accuracy is **77.8%,** which was lower than the previously defined classifier accuracy

C.Negation

Code:

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

def NOT_features(document, word_features, negationwords):

```
    features = {}
    for word in word_features:
        features['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)
```

Output:

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

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)
```

```
0.788
Most Informative Features
        contains(boring) = True              neg : pos    =     18.5 : 1.0
      contains(captures) = True              pos : neg    =     18.4 : 1.0
    contains(engrossing) = True              pos : neg    =     17.7 : 1.0
      contains(mediocre) = True              neg : pos    =     15.0 : 1.0
          contains(dull) = True              neg : pos    =     14.5 : 1.0
          contains(flat) = True              neg : pos    =     13.8 : 1.0
           contains(90) = True               neg : pos    =     13.6 : 1.0
      contains(powerful) = True              pos : neg    =     13.0 : 1.0
    contains(refreshing) = True              pos : neg    =     12.4 : 1.0
  contains(refreshingly) = True              pos : neg    =     11.7 : 1.0
       contains(routine) = True              neg : pos    =     11.6 : 1.0
         contains(stale) = True              neg : pos    =     11.6 : 1.0
     contains(NOTenough) = True              neg : pos    =     11.6 : 1.0
          contains(warm) = True              pos : neg    =     11.0 : 1.0
      contains(delicate) = True              pos : neg    =     11.0 : 1.0
     contains(wonderful) = True              pos : neg    =     10.6 : 1.0
        contains(stupid) = True              neg : pos    =     10.6 : 1.0
         contains(ages) = True               pos : neg    =     10.4 : 1.0
     contains(realistic) = True              pos : neg    =     10.4 : 1.0
      contains(tiresome) = True              neg : pos    =     10.3 : 1.0
      contains(chilling) = True              pos : neg    =      9.7 : 1.0
       contains(quietly) = True              pos : neg    =      9.7 : 1.0
     contains(offensive) = True              neg : pos    =      9.6 : 1.0
      contains(annoying) = True              neg : pos    =      9.6 : 1.0
    contains(apparently) = True              neg : pos    =      9.6 : 1.0
    contains(meandering) = True              neg : pos    =      9.6 : 1.0
    contains(unexpected) = True              pos : neg    =      9.4 : 1.0
      contains(supposed) = True              neg : pos    =      9.4 : 1.0
          contains(loud) = True              neg : pos    =      9.0 : 1.0
        contains(unless) = True              neg : pos    =      9.0 : 1.0
```

Explaination:

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 78.8% again. Which is the highest