NLP Lab Session
Week 10, November 2, 2017
Constructing Feature Sets for Sentiment Classification in the NLTK

**Getting Started**

For this lab session download the following files from Blackboard and put them in your class folder for copy/pasting examples.

LabWeek10sentimentfeatures.sents.ipynb or .py
Subjectivity.py
subjclueslen1-HLTEMNLP05.tff.zip

Unzip the subjclues file and remember the location and start your jupyter notebook session.

```
>>> import nltk
```

**Sentiment/Opinion Classification (using the Movie Review corpus sentences)**

In today's lab, we will look at two ways to add features that are sometimes used in various sentiment or opinion classification problems. In addition to providing a corpus of the 2000 positive and negative movie review documents, Pang and Lee had a subset of the sentences of the corpus annotated for sentiment in each sentence. We will illustrate the process of sentiment classification on this corpus of sentences with positive or negative sentiment labels.

We start by loading the sentence_polarity corpus and creating a list of documents where each document represents a single sentence with the words and its label.

```
>>> from nltk.corpus import sentence_polarity
>>> import random
```

Look at sentences from the entire list of sentences.

```
>>> sentences = sentence_polarity.sents()
>>> len(sentences)
>>> type(sentences)
>>> sentence_polarity.categories()
```

The movie review sentences are not labeled individually, but can be retrieved by category. We first create the list of documents where each document(sentence) is paired with its label.

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

In this list, each item is a pair (sent,cat) where sent is a list of words from a movie review sentence and cat is its label, either 'pos' or 'neg'.

```
>>> documents[0]
>>> documents[-1]
```

Since the documents are in order by label, we mix them up for later separation into training and test sets.

```
>>> random.shuffle(documents)
```

We need to define the set of words that will be used for features. This is essentially all the words in the entire document collection, except that we will limit it to the 2000 most frequent words. Note that we lowercase the words, but do not do stemming or remove stopwords.

```
>>> 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]
```

Now we can define the features for each document, using just the words, sometimes called the BOW or unigram features. The feature label will be 'contains(keyword)' for each keyword (aka word) in the word_features set, and the value of the feature will be Boolean, according to whether the word is contained in that document.

```
>>> 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 for the documents.

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

We create the training and test sets, train a Naïve Bayes classifier, and look at the accuracy, and this time we'll do a 90/10 split of our approximately 10,000 documents.

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

The function show_most_informative_features shows the top ranked features according to the ratio of one label to the other one. For example, if there are 20 times as many positive documents containing this word as negative ones, then the ratio will be reported as 20.00: 1.00 pos:neg.

```
>>> classifier.show_most_informative_features(30)
```

**Sentiment Lexicon: Subjectivity Count features**

We will first read in the subjectivity words from the subjectivity lexicon file created by Janyce Wiebe and her group at the University of Pittsburgh in the MPQA project. Although these words are often used as features themselves or in conjunction with other information, we will create two features that involve counting the positive and negative subjectivity words present in each document.

Copy and paste the definition of the readSubjectivity function from the Subjectivity.py module. We'll look at the function to see how it reads the file into a dictionary.

Create a path variable to where you stored the subjectivity lexicon file. Here is an example from my mac, making sure the path name goes on one line:
## nancymacpath =
"/Users/njmccrac1/AAAdocs/research/subjectivitylexicon/hltemnlp05clues/subjcluesl
en1-HLTEMNLP05.tff"

Create your own path for the lab PC:
>>> SLpath = <put the path here>

Now run the function that reads the file. It creates a Subjectivity Lexicon that is represented here as a dictionary, where each word is mapped to a list containing the strength, POStag, whether it is stemmed and the polarity. (See more details in the Subjectivity.py file.)
>>> SL = readSubjectivity(SLpath)

Now the variable SL (for Subjectivity Lexicon) is a dictionary where you can look up words and find the strength, POS tag, whether it is stemmed and polarity. We can try out some words.
>>> SL['absolute']
>>> SL['shabby']
Or we can use the Python multiple assignment to get the 4 items:
>>> strength, posTag, isStemmed, polarity = SL['absolute']

Now we create a feature extraction function that has all the word features as before, but also has two features 'positivecount' and 'negativecount'. These features contains counts of all the positive and negative subjectivity words, where each weakly subjective word is counted once and each strongly subjective word is counted twice. Note that this is only one of the ways in which people count up the presence of positive, negative and neutral words in a document.

```
>>> def SL_features(document, 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
```

Now we create feature sets as before, but using this feature extraction function.

```
>>> SL_featuresets = [(SL_features(d, SL), c) for (d,c) in documents]

# features in document 0
>>> SL_featuresets[0][0]['positivecount']
7
>>> SL_featuresets[0][0]['negativecount']
2
>>> SL_featuresets[0][1]
'pos'
>>> train_set, test_set = SL_featuresets[1000:], SL_featuresets[:1000]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set)
```

In my random training, test split, these particular sentiment features did improve the classification on this dataset. But also note that there are several different ways to represent features for a sentiment lexicon, e.g. instead of counting the sentiment words, we could get one overall score by subtracting the number of negative words from positive words, or other ways to score the sentiment words. Also note that there are many different sentiment lexicons to try.

**Negation features**

Negation of opinions is an important part of opinion classification. Here we try a simple strategy. We look for negation words "not", "never" and "no" and negation that appears in contractions of the form "doesn't".

One strategy with negation words is to negate the word following the negation word, while other strategies negate all words up to the next punctuation or use syntax to find the scope of the negation.

We follow the first strategy here, and we go through the document words in order adding the word features, but if the word follows a negation words, change the feature 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

The form of some of the words is a verb followed by n't. Now in the Movie Review Corpus itself, the tokenization has these words all split into 3 words, e.g. "couldn", "'", and "t". (and I have a NOT_features definition for this case). But in this sentence_polarity corpus, the tokenization keeps these forms of negation as one word ending in "n't".

```
>>> for sent in list(sentences)[:50]:
...   for word in sent:
...     if (word.endswith("n't")):
...       print(sent)
```

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

Start the feature set with all 2000 word features and 2000 Not word features set to false. If a negation occurs, add the following word as a Not word feature (if it's in the top 2000 feature words), and otherwise add it as a regular feature word.

```
>>> 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
```

Create feature sets as before, using the NOT_features extraction funtion, train the classifier and test the accuracy.

```
>>> 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[200:], NOT_featuresets[:200]
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> print nltk.classify.accuracy(classifier, test_set)
```

```
>>> classifier.show_most_informative_features(20)
```

In my random split, using the negation features did improve the classification.

**Other features**

There are other types of possible features. For example, sometimes people use bigrams in addition to just words/unigrams or use the words together with a POS tagger. Also, there are many other forms of negation features.

For some problems, the word features can be pruned with a stop word list, but care should be taken that the list doesn't remove any negation or useful function words. A very small stop word list is probably better than a large one.

**Exercise**

Let's try using a stopword list to prune the word features. We'll start with the NLTK stop word list, but we'll remove some of the negation words, or parts of words, that our negation filter uses. This list is still pretty large.

```
>>> stopwords = nltk.corpus.stopwords.words('english')
>>> len(stopwords)
127
>>> stopwords

>>> newstopwords = [word for word in stopwords if word not in negationwords]
>>> len(newstopwords)
>>> newstopwords
```

Now take the new stop words out of the collection of all words and then take the top 2000 to be the word features.

```
>>> new_all_words_list = [word for word in all_words_list if word not in
newstopwords]
```

Now continue to get new word features of length 2000 after the stopwords are removed:

```
>>> new_all_words = nltk.FreqDist(new_all_words_list)
>>> new_word_items = new_all_words.most_common(2000)
>>> new_word_features = [word for (word,count) in new_word_items]
>>> new_word_features[:30]
```

Now choose to re-run one of the classifiers with the word_features having stop words removed. Noting that the definition of the feature functions uses the word_features variable, choose to redefine either

featuresets
SL_featuresets
or NOT_featuresets

Rerun the training and test sets, train the classifier and report on classifier accuracy in the discussion. Be sure to post the baseline accuracy that you got for that type of feature set when you first ran it and the new accuracy score with stopwords removed.

Another option would be to re-define the SL_features function to have just one numeric feature that would subtract the number of negative words from positive words. Again you would post a baseline accuracy score for the original SL_features and a new accuracy score for the new definition of features.