NLP Lab Session
Week 10, October 30, 2013
More Classification and Feature Sets in the NLTK

Getting Started

As usual, we will work together through a series of small examples using the IDLE window that will be described in this lab document. However, for purposes of using cut-and-paste to put examples into IDLE, the examples can also be found in a python file on the iLMS system, under Resources.

Labweek10classify.py

Open an IDLE window. Use the File-> Open to open the labweek10classify.py file. This should start another IDLE window with the program in it. **Each example line(s**) can be cut-and-paste to the IDLE window to try it out.

```
import nltk
```

These examples and others appear in Chapter 6 of the NLTK book.

POS Tagging Classifier

We first use the example of POS tagging in order to show how to build a feature set in the NLTK and to run a classifier.

As we saw last week, for each item to be classified, in this case a single word, in NLTK we build the features of that item as a dictionary that maps each feature name to a value, which can be a Boolean, a number or a string. A feature set is the feature dictionary together with the label of the item to be classified, in this case the POS tag.

We start by looking at suffixes of words and building features, which we will name 'endswith(s)', where s can be any suffix. The value of the feature will be True or False, depending on whether the word ends with that suffix.

To get a set of suffixes to use for features, we will select the 100 most frequent ones in our corpus, using suffixes of lengths 1, 2 or 3.

```
>>> from nltk.corpus import brown
>>> suffix_fdist = nltk.FreqDist()
>>> for word in brown.words():
        word = word.lower()
        suffix_fdist.inc(word[-1:])
        suffix_fdist.inc(word[-2:])
        suffix_fdist.inc(word[-3:])

>>> common_suffixes = suffix_fdist.keys()[:100]
>>> print common_suffixes
```

Now we write a function that will take a word and create the features for that word.

```
>>>def pos_features(word):
        features = {}
        for suffix in common_suffixes:
                features['endswith(%s)' % suffix]=word.lower().endswith(suffix)
        return features
```

We can test this on some words.

```
>>> pos_features('lovely')
>>> pos_features('expansion')
```

In the NLTK book chapter, they define a decision tree classifier using just these single word features, but we'll go on to define additional features using the word context of the word we're classifying. (This example is included here because it is similar to the word vector features of text categorization, using Boolean features.)

We know that we can improve POS tagging if we take account of the context of the word. So we define a new POS feature function that takes an entire sentence and can use the previous word in the sentence. Also, instead of using the suffixes for feature names and giving Boolean values, we now use feature names of 'suffix(1)', 'suffix(2)', and 'suffix(3)' and the values of the these features will be the string that contains the suffix letters of lengths 1, 2, and 3.

```
# the pos features function takes the sentence and the index of a word i
#   it creates features for word i, including the previous word i-1
>>> def pos_features(sentence, i):
        features = {"suffix(1)": sentence[i][-1:],
                    "suffix(2)": sentence[i][-2:],
                    "suffix(3)": sentence[i][-3:]}
        if i == 0:
                features["prev-word"] = "<START>"
    else:
                features["prev-word"] = sentence[i-1]
        return features
```

Recall that the corpus function "sents" returns a list of sentences, where each sentence is a list of tokens. Look at the features of the first 8 words of the first sentence in the Brown corpus:

```
>>> brown.sents()[0]
>>> pos_features(brown.sents()[0], 8)
```

Now we take all the sentences in the news portion of Brown and apply our function to get the POS features, as a dictionary, of each (untagged) word. In order to apply the pos_features function, we use the nltk.tag.untag function to get an untagged sentence and then the python enumerate function to get a list that pairs the index number of each word with the word and tag. After applying the pos_features function to get features for the word, we pair the features with the correct (gold) tag to get a feature set for each word.

```
>>> tagged_sents = brown.tagged_sents(categories='news')
>>> featuresets = []
>>> for tagged_sent in tagged_sents:
        untagged_sent = nltk.tag.untag(tagged_sent)
        for i, (word, tag) in enumerate(tagged_sent):
                featuresets.append( (pos_features(untagged_sent, i), tag) )
```

Look at the feature sets of the first 10 words.

```
>>> for f in featuresets[:10]:
        print f
```

Finally we separate our corpus into training and test sets and use these feature sets to train a Naïve Bayes classifier and look at the accuracy. Remember that the nltk.classify.accuracy function uses the classifier to classify the unlabeled words from the test set and then compares those tags with the gold tags.

```
>>> size = int(len(featuresets) * 0.1)
>>> train_set, test_set = featuresets[size:], featuresets[:size]
>>> len(train_set)
>>> len(test_set)
>>> classifier = nltk.NaiveBayesClassifier.train(train_set)
>>> nltk.classify.accuracy(classifier, test_set)
```

Note that we have not incorporated the previous tag in our features for learning. To do that, we could keep a variable called history that recorded the tag of the previous words as we run the tagger. This is shown in the NLTK book example, but we won't do it here.

Sentence Segmentation

Sentence segmentation can be viewed as a classification task that labels each punctuation symbol that could end a sentence with whether it does end a sentence or not.

We get sentences already segmented from the treebank corpus, where the sents() function appears to return a list where each element is a sentence represented as a list of tokens, and some extra sentences with a single token ['.', 'START'] appear between some of the regular sentences (perhaps these are related to groupings of sentences in files?)

```
>>> sents = nltk.corpus.treebank_raw.sents()
>>> sents[:10]
```

In order to get data for classifiying without sentence boundaries, we merge the sentences into one long list of tokens for classifying, but keep track of the index numbers where the sentence boundaries are. Then our classifier is going to try to find the punctuation at the sentence boundaries.

```
>>> tokens = [ ]
>>> boundaries = set()
>>> offset = 0
>>> for sent in nltk.corpus.treebank_raw.sents():
    tokens.extend(sent)
    offset += len(sent)
    boundaries.add(offset - 1)
```

Look at some tokens and test some boundary numbers to see if they are in the set.

```
>>> tokens[:30]
```

Next, we set up a feature extractor function that works on each token. It checks in the next word is capitalized, puts in the previous word, the actual token (called 'punct') and if the previous word is one character long.

```
def punct_features(tokens, i):
   return {'next-word-capitalized': tokens[i+1][0].isupper(),
           'prevword': tokens[i-1].lower(),
           'punct': tokens[i],
           'prev-word-is-one-char': len(tokens[i-1]) == 1}
```

Now we go through all the tokens, and for any token that is potential sentence ender, i.e. a ".", "?", or "!" , we build a feature set for that occurrence of the token (at index i).

```
>>> Sfeaturesets = [(punct_features(tokens, i), (i in boundaries))
     for i in range(1, len(tokens) - 1)
     if tokens[i] in '.?!']
```

Now we separate the feature sets into training and test sets, train a classifier and get the accuracy.

```
>>> size = int(len(Sfeaturesets) * 0.1)
>>> Strain_set, Stest_set = Sfeaturesets[size:], Sfeaturesets[:size]
>>> Sclassifier = nltk.NaiveBayesClassifier.train(Strain_set)
>>> nltk.classify.accuracy(Sclassifier, Stest_set)
```

Finally, after training the classifier, we demonstrate how we could use the classifier to find sentence boundaries. For this, we take list of tokens/words and whenever one is a ".", "?", or "!", we apply the classifier to label it. If it is an end of sentence marker, we have a special START symbol into the stream of tokens. We apply this to a small set of the merged tokens from Penn treebank.

```
>>> def segment_sentences(words):
    start = 0
     sents = []
     for i, word in enumerate(words):
        if word in '.?!' and Sclassifier.classify(punct_features(words, i)) == True:
          sents.append(words[start:i+1])
```

```
            start = i+1
    if start < len(words):
        sents.append(words[start:])
    return sents

>>>len(tokens)
102055
>>> smalltokens = tokens[:1000]
>>> for s in segment_sentences(smalltokens)
   print s
```

Text Classification (aka Text Categorization)

For a different type of classification problem, we next look at text classification. In this problem, the items to be classified are documents. Most widely known are datasets that label each document with a topic category (hence the name categorization), but we will look at documents from the NLTK Movie Review corpus, where each document is labeled either 'pos' for positive or 'neg' for negative, according to the opinion of the review. There are 1000 positive reviews and 1000 negative reviews in the corpus.

The features of each document will be the words contained in the document, out of a set of words that are frequent in the whole document collection.

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

>>> movie_reviews.categories()
```

The movie review documents are not labeled individually, but are separated into file directories by category. We first create the list of documents where each document is paired with its label.

```
>>> documents = [(list(movie_reviews.words(fileid)), category)
            for category in movie_reviews.categories()
            for fileid in movie_reviews.fileids(category)]
```

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

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

We look at the first document, which will consist of all the words in the review, followed by the label. Since we did independent shuffles, each person should have a different document.

```
>>> documents[0]
```

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.

```
>>> all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
```

```
>>> word_features = all_words.keys()[:2000]
```

Look at the first 100.

```
>>> word_features[:100]
```

Now we can define the features for each document. 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.

(For topic categorization, it is better to represent each word feature by its frequency (or a related score) in the document, but for sentiment classification, it is better to just use True of False depending on whether the word is present.)

```
>>> def document_features(document):
        document_words = set(document)
        features = {}
        for word in word_features:
            features['contains(%s)' % word] = (word in document_words)
        return features
```

Define the feature sets for the documents. We can look at the first one, but remember that it contains 2000 words.

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

(optional – very long)

```
>>> featuresets[0]
```

We create the training and test sets, train a Naïve Bayes classifier, and look at the accuracy.

```
>>> train_set, test_set = featuresets[100:], featuresets[:100]
>>> 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(20)
```

Exercise:

Each group/person should choose a different number of words to use for features, instead of the 2000 used here. First note down what classifier accuracy you got with the 2000 words.

Then each group/person should re-run the movie review text classification problem (but don't shuffle again!) and post their two accuracy numbers to the discussion list. If we get a wide variety of the number of words, we should get an idea of how the accuracy increases (or not) according to the number of words used. (Although this will only be an estimate, since each person ran their own random train/test split.)