NLP Lab Session
Week 10, November 4, 2015
More Classification and Feature Sets in the NLTK

Getting Started

For this lab session download the examples: LabWeek10classify.txt and put it in your class folder for copy/pasting examples. Start your Python interpreter or IDLE session.

```
>>> import nltk
```

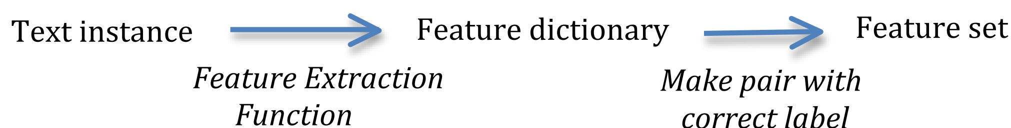
These examples and others appear in Chapter 6 of the NLTK book.

The goal of this week's lab is to show how to represent several types of features for classification in the NLTK.

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. We will set up the POS tagging problem as a classification problem that tries to label each word with the correct POS tag. From our previous discussions of POS tagging in the lectures, we know that the best POS taggers use a combination of an HMM sequential tagger that can use the previous tag and a feature-based classifier similar to the one that we'll set up here.

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.



One source of information for POS tagging is the morphology of the word, and we can start by looking at suffixes of words and building features.

We also know that we can improve POS tagging if we take account of the context of the word. So we define a POS feature function that takes an entire sentence and can use the previous word in the sentence. We will 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.

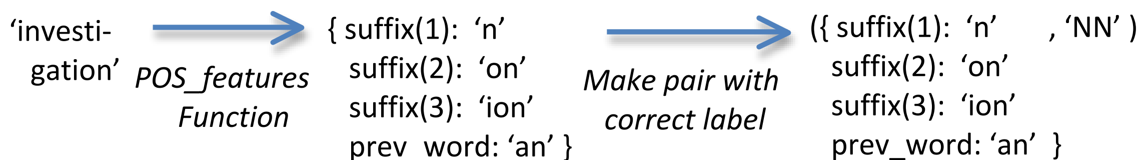
```
# the pos features function takes the sentence and the index of a word i
#   it creates features for word i, including the previous word i-1
>>> def pos_features(sentence, i):
        features = {"suffix(1)": sentence[i][-1:],
                    "suffix(2)": sentence[i][-2:],
```

```
                "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 word at index 8 of the first sentence in the Brown corpus:

```
>>> sentence0 = brown.sents()[0]
>>> sentence0
>>> sentence0[8]
>>> pos_features(sentence0, 8)
```

For this word, our POS_features function gives the feature dictionary. Now we need to apply our feature function to all words in the training corpus and to pair it with the correct tag.



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, e.g. here is the untag function applied to the first sentence.

```
>>> tagged_sents = brown.tagged_sents(categories='news')
>>> tag_sent0 = tagged_sents[0]
>>> nltk.tag.untag(tag_sent0)
```

In order to apply the POS_features function to the untagged sentence, we need an index number for each word, and the python enumerate function will return a list that pairs the index number of each word with the word and tag.

```
>>> for i,(word,tag) in enumerate(tag_sent0):
    print i, word, 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.

```
>>> 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 other features from the surrounding words, so this classifier accuracy is not bad for this amount of feature information. And, of course, the feature-based classifier should be combined with a sequential classifier, such as HMM, to achieve the highest performance.

Sentence Segmentation

Sentence segmentation can be viewed as a classification task that labels each punctuation symbol that could end a sentence, i.e. either a ".", "!" or "?", with whether it does end a sentence or not.

We get sentences already segmented from the treebank corpus, where the sents() function returns a list where each element is a sentence represented as a list of tokens. 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?), yielding 4355 sentences.

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

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 for our gold standard data. 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[:40]
>>> 19 in boundaries
>>> 20 in boundaries
```

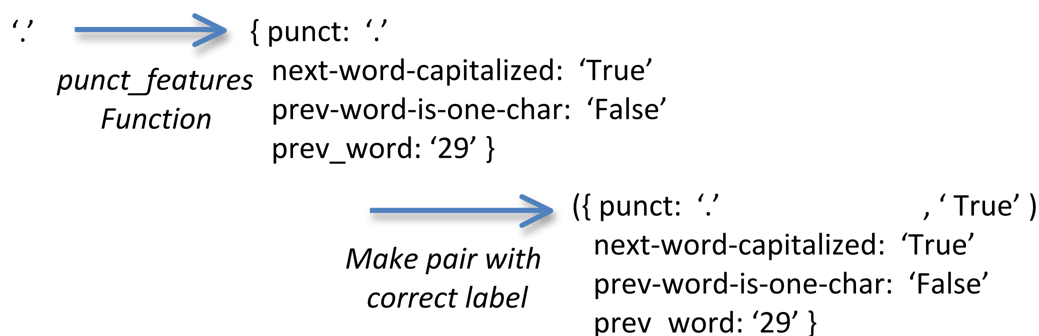
Next, we set up a feature extractor function that works on each token, where we intend to use it on potential end-of-sentence punctuation. It checks in the next word is capitalized, puts in the previous word, the actual token (called 'punct') and if the previous word is one character long.

```
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}
```

Here is the feature dictionary for the token '.' at index 20:

```
>>> tokens[20]
>>> punct_features(tokens,20)
```

Next we need to build the list of feature sets that pairs every feature dictionary with its label, for all the candidate punctuation in the token list.



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)
```

We can apply the classifier to individual tokens to see if they are at the end of a sentence:

```
# this is the . after Nov
>>> Sclassifier.classify(punct_features(tokens, 18))
# this is the . after 29, which should be true!
>>> Sclassifier.classify(punct_features(tokens, 20))
# this is the . after group
>>> Sclassifier.classify(punct_features(tokens, 36))
```

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
```

As a digression, we note that the actual built-in sentence segmenter in the NLTK is from a package called punkt and can be imported from the tokenize module:

```
>>> from nltk.tokenize import sent_tokenize
```

It is also a trained classifier, but it works at the character level instead of the token level, so you give it the raw text to segment into sentences, and then you tokenize.

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 part of the corpus in NLTK.

The features of each document will be the words contained in the document, but limited to 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. (If you're using NLTK 3.0, you must sort by frequency first.)

```
>>> all_words = nltk.FreqDist(w.lower() for w in movie_reviews.words())
>>> word_features = all_words.keys()[:2000]
```

[Note: when using NLTK 3.x, use the most_common function instead, which returns (word, frequency) pairs of the 2000 most frequent words.

```
>>> word_items = all_words.most_common(2000)
```

And then, the word features is the list of just the words, without frequencies.

```
>>> word_features = [word for (word, freq) in word_items]
```

End Note: 3.x]

Look at the first 100 words in the word_features list.

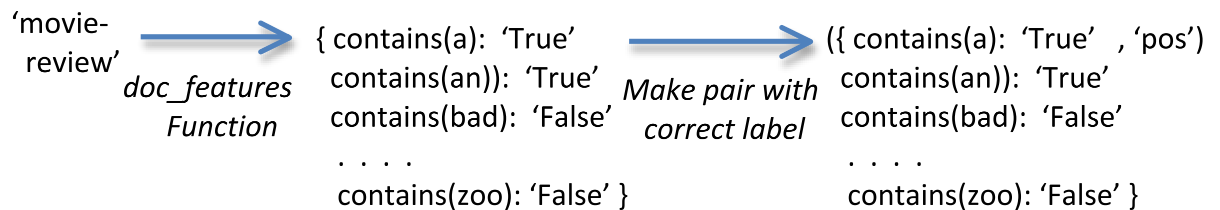
```
>>> word_features[:100]
```

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
```

The feature dictionary has 2000 features, each with the value True or False.



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(30)
```

Exercise:

Each 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 pick a different number. Examples of the types of numbers to try are 1000, 1200, 2500, etc., and we don't have to be too systematic.

Then re-run the movie review text classification problem (but don't shuffle again!) and post the 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.)