NLP Lab Session
Week 10

More Classification and Feature Sets in the NLTK

Getting Started

```
>>> import nltk
>>>from nltk.corpus import brown
```

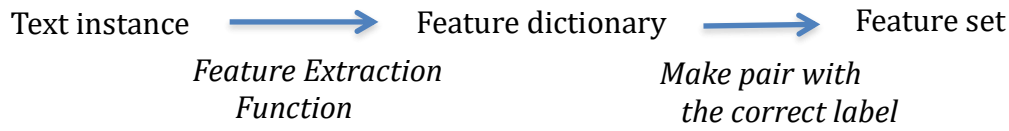
These examples and others appear in Chapter 6 of the NLTK book.

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 in the previous lab sessions, 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 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.

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 (*in the format of the token list)* and can use the previous word in the sentence.

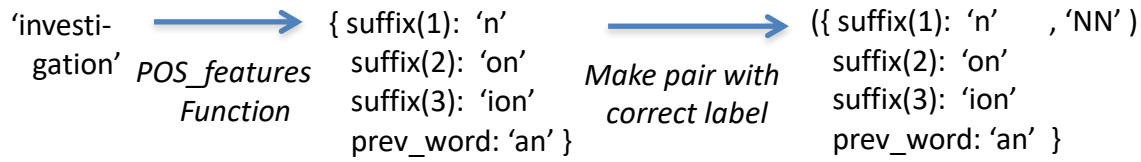
```
# the pos features function takes the sentence and the index of a word i
#   it creates features for word i, including the previous word i-1
>>> def pos_features(sentence, i):
        features = {"suffix(1)": sentence[i][-1:],
                    "suffix(2)": sentence[i][-2:],
                  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

[Note: you can also try the following to get this untagged sentence list and see if there is a difference between the two approaches:
>>>newssent = brown.sents(categories = 'news')
End Note]

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 4193 sentences.

```
>>> sents = nltk.corpus.treebank_raw.sents()
>>> sents[:10]
>>> len(sents)
>>> for sent in sents[:10]:
   print (sent)
```

In order to get data for classifying 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
>>> for num, tok in enumerate(tokens[:40]):
    print (num, tok, '\t', num 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 if 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)
```

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)
101797
>>> 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.

```
>>> rawtext = 'Pierre Vinken, 61 years old, will join the board as a nonexecutive
director Nov. 29.  Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing
group.'
>>> sents = nltk.sent_tokenize(rawtext)
>>> sents
```