NLP Lab Session Week 5
February 18, 2010

POS taggers in NLTK

Installing NLTK Toolkit

Reinstall nltk-2.0b7.win32.msi and Copy and Paste nltk_data from H:\ nltk_data to C:\ nltk_data. Note that if you need to download the nltk installer again from nltk.org, that the installer is now separated into two parts and you must install them both – nltk and yaml.

Getting Started

In this lab session, we will work together through a series of small examples using the IDLE window and 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.

Labweek5examples.py

Open an IDLE window. Use the File-> Open to open the labweek4examples.py file. This should start another IDLE window with the program in it. Each example line can be cut-and-paste to the IDLE window to try it out.

Reading Tagged Corpora

The NLTK corpus readers have additional methods (aka functions) that can give the additional tag information from reading a tagged corpus. Both the Brown corpus and the Penn Treebank corpus have text in which each token has been tagged with a POS tag. (These were manually assigned by annotaters.)

The tagged_sents function gives a list of sentences, each sentence is a list of (word, tag) tuples. (Python tuples are a 2-element list that you can't change the order of the elements.) We'll first look at the Brown corpus, which is described in Chapter 2 of the NLTK book.

```
import nltk
from nltk.corpus import brown
brown.tagged_sents()[:2]
```

The tagged_words function just gives a list of all the (word, tag) tuples, ignoring the sentence structure.

```
brown.tagged_words()[:50]
```

The Brown corpus is organized into different types of text, which can be selected by the categories argument, and it also allows you to map the tags to a simplified tag set, described in table 5.1 in the NLTK book.

```
brown_news_tagged = brown.tagged_words(categories='news', simplify_tags=True)
brown_news_tagged[:50]
```

Other tagged corpora also come with the tagged_words method:

```
nltk.corpus.nps_chat.tagged_words()[:50]
```

The Penn Treebank has the tagged_words and tagged_sents methods as well.

```
from nltk.corpus import treebank
treebank.tagged_words()[:50]
len(treebank.tagged_words())
```

In our previous labs, we used the Frequency Distributions of the NLTK to map words to numbers (the frequencies), and we noted that this is just a Python dictionary. NLTK also has a structure called the Conditional Frequency Distribution which is a nested dictionary. It allows you to map a (word, tag) pair to a frequency.

Here is a function definition that finds the most frequent words that are tagged with any of the noun tags, those prefixed with NN.

```
# find the most frequent words in Penn Treebank that have one of the noun tags
def findtags(tag_prefix, tagged_text):
        cfd = nltk.ConditionalFreqDist((tag, word) for (word, tag) in tagged_text
                                        if tag.startswith(tag_prefix))
        return dict((tag, cfd[tag].keys()[:20]) for tag in cfd.conditions())

tagdict = findtags('NN', treebank.tagged_words())
for tag in sorted(tagdict):
        print tag, tagdict[tag]
```

POS Tagging

The process of classifying words into their parts of speech and labeling them accordingly is known as part-of-speech tagging, POS-tagging, or simply tagging. Parts of speech are also known as word classes or lexical categories. The collection of tags used for a particular task is known as a tagset. We will use two different parsers to tagging text.

First we initialize some variables from NLTK, the Penn Treebank. The first has the tagged sentences. We will use those in training later taggers and to compare for the evaluation. The second just has the words that we will use to test our taggers.

```
treebank_tagged = treebank.tagged_sents()
treebank_text = treebank.words()
len(treebank_text)
```

To introduce the N-gram taggers in NLTK, we start with a default tagger that just tags everything with the most frequent tag: NN. We create the tagger and run it on text.

```
default_tagger = nltk.DefaultTagger('NN')
default_tagger.tag(treebank_text[:50])
```

The NLTK includes a function for taggers that computes tagging accuracy by comparing the result of a tagger with the original "gold standard" tagged text. Here we apply the default tagger (to the untagged text) and compare it with the gold standard tagged text.

```
default_tagger.evaluate(treebank_tagged)
```

Other simple taggers described in the NLTK book are the Regular Expression Tagger and the Lookup Tagger.

Next we train a Unigram tagger. It tags each word with the most frequent tag in the corpus. First we compare it to the same corpus that it learned from.

```
unigram_tagger = nltk.UnigramTagger(treebank_tagged)
unigram_tagger.tag(treebank_text[:50])
```

Next, we do separate the tagged data into a training set and a test set. This allows us to test the tagger's accuracy on similar , but not the same, data that it was trained on. We take the first 90% of the data for the training set, and the remaining 10% for the test set.

```
size = int(len(treebank_tagged) * 0.9)
treebank_train = treebank_tagged[:size]
treebank_test = treebank_tagged[size:]
unigram_tagger = nltk.UnigramTagger(treebank_train)
unigram_tagger.evaluate(treebank_test)
```

Finally, NLTK has a Bigram tagger that can be trained using 2 tag-word sequences. But there will be unknown frequencies in the test data for the bigram tagger, and unknown words for the unigram tagger, so we can use the backoff tagger capability of NLTK to create a combined tagger. This tagger uses bigram frequencies to tag as much as possible. If a word doesn't occur in a bigram, it uses the unigram tagger to tag that word. If the word is unknown to the unigram tagger, then we use the default tagger to tag it as 'NN'.

```
t0 = nltk.DefaultTagger('NN')
t1 = nltk.UnigramTagger(treebank_train, backoff=t0)
t2 = nltk.BigramTagger(treebank_train, backoff=t1)
t2.evaluate(treebank_test)
```

Exercise:

For this exercise, you are to choose any document, including any of the untagged documents in the Gutenberg corpus, run the bigram POS tagger, and find the top frequency words by POS tag class. As usual, you may work in groups.

(There is one caveat on using the bigram POS tagger on these texts, which is that it was trained on text which had been separated by sentences. So we should really run a sentence detector on our raw text first, and we'll talk about how to do that in the future.)

1. Choose a document, as we did for the word frequencies, and tokenize it. Do not lowercase the words or stem them. For example, if I chose emma, the first document in the Gutenberg corpus, I would get the raw text and tokenize it with wordpunct, ending up with something named emmatokens.

2. Now run the bigram pos tagger - it was named t2 in our development – and save the results in a variable (don't try to print the whole thing). For example, I would save mine in a variable called emmatagged.

3. Use the findall function to define 4 dictionaries for nouns (prefix 'N'), verbs (prefix 'V'), adjectives (prefix 'J') and adverbs (prefix 'R'). Display these dictionaries.

4. Choose the results of one of your dictionaries and post it to the discussion for today's lab. Include the names of everyone in your group in the discussion title. The discussion entry should include the name of your document and what type of words you are giving.