NLP Lab Session Week 6
October 1, 2014

POS taggers in NLTK and Notes on Homework 2

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 in blackboard, under Lab Sessions and Resources.

Labweek6examples.py

Open an IDLE window. Use the File-> Open to open the labweek6examples.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.

Also from blackboard, under Assignments, **download the SpamLord.zip** file and put it in your NLP class folder. We'll talk in the lab about how to unzip this file.

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 annotators.)

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.categories()
```

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

Other tagged corpora also come with the tagged_words method. Note that the chat corpus is tagged with Penn Treebank POS tags.

```
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())

treebank.tagged_sents()[:2]
len(treebank.tagged_sents())
```

The NLTK has almost 4,000 sentences of tagged data from Penn Treebank, while the actual Treebank has much more. This will limit the accuracy of the parsers that we can define in lab, but also make the running times short enough for labs.

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. NLTK calls the first set of keys the "conditions", which index a regular frequency distribution with keys and values.

This Conditional Frequency Distribution maps POS tags and words to frequencies:

```
cfd = nltk.ConditionalFreqDist((tag, word) for (word, tag) in treebank_tagged_words)
```

At the top level the keys are also called conditions. Note that these keys/conditions are the POS tags.

```
cfd.conditions()
```

Each tag is mapped to a Frequency Dictionary whose keys are the words and whose values are the frequencies of the words.

```
cfd['NN'].keys()[:20]
cfd['NN']['company']
```

From all the words that are tagged NN, we can print the 20 top frequency words.

```
for word in cfd['NN'].keys()[:20]:
   print word, cfd['NN'][word]
```

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 a tag starting with a prefix
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]
```

In this section, we looked at both the Brown corpus and the Penn Treebank corpus; we will work with the Penn Treebank corpus to learn POS tag frequencies.

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 different parsers to tagging text.

We will use the tagged sentences and words from the Penn Treebank that we defined in the previous section.

We separate our tagged data into a training set, where we'll learn the probabilities of the words and their tags, and a test set to evaluate how out taggers perform. The training set is the first 90% of the data and the test set is the remaining 10%.

```
size = int(len(treebank_tagged) * 0.9)
treebank_train = treebank_tagged[:size]
treebank_test = treebank_tagged[size:]
```

In the NLTK, a number of POS taggers are included in the tag module, including one that we can use that has been trained on all of Penn Treebank. But for instructional purposes, we will look at some of the other taggers.

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.

```
t0 = nltk.DefaultTagger('NN')
t0.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 use the NLTK function "evaluate" to apply the default tagger (to the untagged text) and compare it with the gold standard tagged text.

```
t0.evaluate(treebank_tagged)
```

The evaluate function first takes the tagged text and removes the tags, so that only tokens are left. Then it runs the tagger, in this case t0, to tag all the text. Then it compares the tags predicted by the tagger with the "gold standard" tags already given. It reports the accuracy, which is the percentage of words with correct tags.

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 that word has in the corpus. For example, if the word "bank" occurs 30 times with the tag "NN" and 10 times with the tag "VB", we'll just tag it with "NN".

```
t1 = nltk.UnigramTagger(treebank_tagged)
t1.tag(treebank_text[:50])
```

Now instead of just evaluating the tagger on the entire set of data, we 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.

```
t1 = nltk.UnigramTagger(treebank_train)
t1.evaluate(treebank_test)
```

In the lecture slides, this Unigram Tagger is what Chris Manning called their baseline tagger and they got about 90% accuracy. Why isn't ours quite as good?

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)
```

This accuracy is not bad, especially on only part of Penn Treebank! We know that HMM and other feature techniques can raise the accuracy to between 95 and 98%.

Let's use this tagger to tag some example text. Define some example text, tokenize it, and apply the tagger.

```
text = "Three Calgarians have found a rather unusual way of leaving snow and ice
behind. They set off this week on foot and by camels on a grueling trek across the
burning Arabian desert."
tokens = nltk.wordpunct_tokenize(text)
taggedtext = t2.tag(tokens)
taggedtext
```

We observe that this text has quite a few words that appear to be unknown to this tagger from the data it was trained on. Examples of this are "Calgarians" and "camels". In both cases, these two words are tagged as NN instead of the correct tags of NNPS and NNS, respectively. This points out the benefit of adding sequence information such as an HMM tagger would use and lexical information, such as a Maximum Entropy tagger could use if you defined such features. In the NLTK, another strategy would be to use a Regular Expression tagger as a backoff tagger that could take into account word features.

In order to look at some of the tags in a text, we can use the findtags function from the last section.

```
tagdictNN = findtags('NN', taggedtext)
#Find the most frequent words in this example that have one of the noun tags
for tag in sorted(tagdictNN):
    print tag, tagdictNN[tag]
```

Stanford POS Tagger

One of the problems with training our own POS tagger is that we don't have all the Penn Treebank data. But NLTK also provides some taggers that come pre-trained on the larger amount of data. One of these is the Stanford POS tagger, which was trained using a maximum entropy classifier. This is described in the nltk.tag module:

http://www.nltk.org/_modules/nltk/tag.html

This tagger is available in the module:
'taggers/maxent_treebank_pos_tagger/english.pickle'
and it is used for the standard nltk.pos_tag function.

```
taggedtextStanford = nltk.pos_tag(tokens)
taggedtextStanford
```

Homework 2: Development of Regular Expressions in the SpamLord program

You should have downloaded the zipped SpamLoard file to somewhere on your H: drive and unzipped it. Please read the homework 2 assignment to see the directory structure and an overview of the SpamLord program.

We are going to look at the program SpamLord.base.py and work on the program SpamLord.py in this lab. You can edit the latter program either by opening it in a program like NotePad++ or by opening it in an IDLE window.

Instead of running this program in IDLE, we are going to run it as a stand-alone Python program from the command line. To do that, open a Command Prompt window by going to the program run box and typing in "cmd", which should open a window for you to type commands.

In the command prompt window, we want to go to the SpamLord directory. First we get to the H: drive and then use the 'cd', change directory, command to navigate to the SpamLord directory.

```
% H:
%  cd NLPclass\SpamLord
```

or whatever path you need to type to get to the SpamLord directory. If you want to see the contents of whatever directory you are in, type the command 'dir'

```
% dir
```

and you should see the two SpamLord programs and the data directory. But if you see another directory called SpamLord, just use cd SpamLord to get down one more level until you are in the same directory as the programs.

We can run the program by just issuing the python program as a command, giving the SpamLord base program and two arguments:

```
% SpamLord.py data/dev data/devGOLD
```
Note: on Macs, use % python SpamLord.base.py data/dev data/devGOLD

The output of the program lists the number of correct matches as True Positives (0), the number of incorrent matches as False Positives (0) and the number of unmatched gold answers as False Negatives (117). Note that each gold/correct answer has the form (filename, 'e' or 'p', standard format email or phone).

To start making patterns, we look at some of the unmatched examples. Let's pick the email address for 'balaji'.

Now in your Explorer computer window (NOT Internet Explorer), go to the data/dev directory and open the file called 'balaji' in a text editor like NotePad++. (But we are only going to read this file and not edit it.) Find an occurrence of an email address for balaji; it's about 2/3 of the way down the file after a mailto: tag. This email address is not obscured, so we can write a regular expression that matches it directly. Let's allow any alphabetic character before the @ sign and any alphabetic character after the @ sign and before .edu. So we will add the following line to SpamLord.base.py after the line epatterns = [ ]. And remember that we are supposed to have one set of parentheses around the "someone" part and another set around the "somewhere" part but not including the .edu.

```
epatterns.append('([A-Za-z]+)@([A-Za-z]+)\.edu')
```

This line should already be in your program and you just need to remove the # comment sign at the beginning of the line. We save our program and run it again. This pattern not only matched the balaji email address but correctly matched three others.

But we note that we incorrectly matched young. We'll go to the data/dev folder again and this time we'll open the file psyoung in a text editor. When we find the email address, we see that the problem is that we didn't match ALL the email address. We also need to be able to match a '.' before the @, and we might as well allow a '.' after the @ as well. We change our pattern to:

```
epatterns.append('([A-Za-z.]+)@([A-Za-z.]+)\.edu')
```

That change not only corrected psyoung, but it added a bunch of additional correct email addresses that had a '.' after the @.

Now let's work on one of the unmatched examples. Open the file for ashishg and find his email address after Email: in the file. We can see that he has put a space before and after the @ sign. Now we can add a second pattern or we can change our first pattern to have an optional space. Here is a second pattern:

```
epatterns.append('([A-Za-z.]+)\s@\s([A-Za-z.]+)\.edu')
```

Save the program and run it again. That captured both of ashishg's email addresses.

Just a reminder that the epatterns list expects each regular expression to have exactly 2 parentheses to capture the userid and the domain and that the result will be followed by .edu. If the text can't be matched in this format, then the epattern list isn't sufficient to do it.

To start you on your homework assignment, I have put this epattern and a phone pattern

ppatterns.append('(\d{3})-(\d{3})-(\d{4})')

into the file SpamLord.py. You can start with this program and keep developing patterns to increase the number of correct matches.

1. Select an unmatched gold example.
2. Find the data/dev file that the example should have come from.
3. Decide if you can add to an existing pattern, write a new pattern, or if the example cannot be matched with the format of the epatterns and ppatterns.
4. If you can add to or write a pattern, edit the program and run it again.
5. Observe the results!
6. If you can't write a pattern to match it with only 2 parentheses, save that example for optional parts 2 and 3.