NLP Lab Session Week 6
February 25, 2010
Running and saving 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

Getting Started

As we did in the last 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.

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(s)** can be cut-and-paste to the IDLE window to try it out.

Continuing with Tagged Corpora

Recall that the NLTK corpus readers have additional methods (aka functions) that can give the additional tag information from reading a tagged corpus.

```
from nltk.corpus import brown
from nltk.corpus import treebank
treebank_tagged = treebank.tagged_sents()
 treebank_text = treebank.words()
```

Continuing with POS Tagging

For our backoff and initial taggers, we need to create the default and unigram taggers.

```
default_tagger = nltk.DefaultTagger('NN')
```

Next, we do separate the tagged data into a training set and a test set:

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

And we create the unigram tagger:

```
unigram_tagger_NN = nltk.UnigramTagger(treebank_train, backoff=default_tagger)
```

unigram_tagger_NN.evaluate(treebank_test)

NLTK has another kind of tagger that allows you to handcraft rules in the form of regular expressions to create a tagger.

First, we create a list of regular expression patterns. The NLTK RegexpTagger will apply this list in turn to the tokens of the text and assign the POS tag that is given along with the regular expression. Here are some (very incomplete) rules:

```
# Regular Expression Tagger
POSpatterns = [
   (r'.*ing$', 'VBG'),      #gerunds
   (r'.*ed$', 'VBD'),     #simple past tense
   (r'.*es$', 'VBZ'),      #3rd singular present
   (r'.*ould$', 'MD'),     #modals
   (r'.*\'s$', 'POS'),     #possesive nouns
   (r'.*s$', 'NNS'),       #plural nouns
   (r'^-?[0-9]+(.[0-9]+)?$', 'CD') #cardinal numbers
   ]
```

We can create a regular expression tagger and test it on text.

```
regexp_tagger = nltk.RegexpTagger(POSpatterns)
# test on example sentence
text = "The man's dog would bite the cat that was running 15.6 miles."
tokens = nltk.wordpunct_tokenize(text)
regexp_tagger.tag(tokens)
```

A better regular expression tagger with combine with a default tagger for unknown words.

```
# better tagger combines with default tagger for unknown words
regexp_tagger = nltk.RegexpTagger(POSpatterns, backoff=default_tagger)
regexp_tagger.tag(tokens)
regexp_tagger.evaluate(treebank_test)
```

Our last tagger from the NLTK is the Brill tagger. It needs a set of templates in order to create transformation rules. Here is the set suggested in the HOWTO documentation of the nltk.org web site. But notice that this version of the Brill tagger does not have any of the unknown word rules using the morphology of words. (Look at the HOWTO section on taggers and the nltk API for the Brill tagger.)

```
# Brill Tagger
from nltk.tag.brill import *

# Brill tagger transformation templates
templates = [
    SymmetricProximateTokensTemplate(ProximateTagsRule, (1,1)),
```

```
    SymmetricProximateTokensTemplate(ProximateTagsRule, (2,2)),
    SymmetricProximateTokensTemplate(ProximateTagsRule, (1,2)),
    SymmetricProximateTokensTemplate(ProximateTagsRule, (1,3)),
    SymmetricProximateTokensTemplate(ProximateWordsRule, (1,1)),
    SymmetricProximateTokensTemplate(ProximateWordsRule, (2,2)),
    SymmetricProximateTokensTemplate(ProximateWordsRule, (1,2)),
    SymmetricProximateTokensTemplate(ProximateWordsRule, (1,3)),
    ProximateTokensTemplate(ProximateTagsRule, (-1, -1), (1,1)),
    ProximateTokensTemplate(ProximateWordsRule, (-1, -1), (1,1)),
    ]
```

The Brill tagger also needs an initial tagger to get things started and we can use the unigram tagger that we created earlier. The steps are repeated here, but you don't actually have to do them again.

```
# Brill tagger with unigram tagger to initialize
default_tagger = nltk.DefaultTagger('NN')
unigram_tagger_NN = nltk.UnigramTagger(treebank_train, backoff=default_tagger)
```

Finally, we run the Brill training algorithm on the Treebank training set. This step takes a long time, so be patient.

```
# This step takes a long time
brill_trainer = FastBrillTaggerTrainer(initial_tagger=unigram_tagger_NN,
                        templates=templates, trace=3,
                        deterministic=True)
brill_tagger = brill_trainer.train(treebank_train, max_rules=10)
```

The Brill training prints out a number of rules that we can see. We can also test its accuracy.

```
print brill_tagger.evaluate(treebank_test)
```

Since some taggers take so long to train, we would like to save the trained tagger in a file. Python has a package called pickle that will write our any types of objects to a file (not just text) and then later retrieve them. There is also a package called cPickle that is supposed to be faster (written in C), but this package didn't always work for me. The lab example uses cPickle, but later I made saved versions of these processes that use pickle. Some notes about using Pickle and cPickle can be found in http://blog.doughellmann.com/2007/06/pymotw-pickle-and-cpickle.html.

```
# Storing Taggers
from cPickle import dump
# open the output file
output = open('brillsaved.pkl', 'wb')
# the dump function serializes the brill_tagger object and write it to the output file
dump (brill_tagger, output, -1)
```

```
output.close()

# Testing the saved tagger
# These actions could be done in a new shell
from cPickle import load
# open the file where the tagger was saved
input = open('brillsaved.pkl', 'rb')
# use the load function to unserialize the object in the file and save it in a variable “tagger”
tagger = load(input)
input.close()

# use the tag function of the restored tagger to tag some text
text = "The action of the board shows what free enterprise is up against."
tokens = text.split()
tagger.tag(tokens)
```

These saved files are called brilltrain.py and brillsavetest.py. The first program will train the Brill tagger (on all the Treebank data) and save it to a file. The second program will retrieve the tagger from a file and run it on the test sentence.

These programs are on the iLMS system.

There is no group exercise today. Please use extra time for homework questions or discussions with your group. And if you didn’t get the exercise done last week, you may complete it today.

Notes for homework:

1. Examples of using plot in nltk (from a conditional frequency distribution).

2. The ANEW data has arrived.