NLP Lab Session Week 8
October 15, 2014

Noun Phrase Chunking and WordNet in NLTK

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.

Labweek8examples.py

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

Chunk Parsing for Base Noun Phrases using Regular Expressions

In other labs, we have looked at the Penn Treebank to see sentences, words and tagged sentences so far. Later in this lab, if time permits, we'll look at parsed sentences in Penn Treebank. But NLTK also has a version of the Penn Treebank which just has base noun phrases annotated, and this corpus is in nltk.corpus.treebank_chunk. There are 200 files in this corpus, which have multiple sentences in each file, making up the almost 4,000 sentences in this part of the Penn Treebank. Here we look at the first file, and it contains 2 sentences.

```
>>> fileid = nltk.corpus.treebank_chunk.fileids()[0]
```

Let's first let the variable s0 be the sentence tree of the first sentence.

```
>>> s0 = nltk.corpus.treebank_chunk.chunked_sents(fileid)[0]
>>> type(s0)
<class 'nltk.tree.Tree'>
```

If we look at this one tree, it has type nltk.tree.Tree and objects of this type have a draw() function that can be used to show the graph. The draw method opens a graphical window that could be hiding behind other windows, so look around!

```
>>> s0.draw()
```

Note that each sentence lists the S tag and each word in the sentence with its POS tag, but it only groups together the NP base noun phrases. There is a function "subtrees()" for nltk trees that will list all the subtrees, including the tree itself. We can use this to see the tree for the entire sentence followed by all the chunked base noun phrases.

```
>>> for senttree in nltk.corpus.treebank_chunk.chunked_sents(fileid):
    for t in senttree.subtrees():
        print t
    print  # print a blank line between sentences
```

Now that we've seen some examples of base noun phrases from the Treebank_chunk corpus, we can build a (shallow) parser that finds those noun phrases.

The NLTK has a chunker function, called a regular expression parser, that uses regular expressions to define a pattern of a sequence of POS tags that should make up a chunk. Note that we are using the annotated data to see examples of what we need to chunk; we look at the annotated data and try to make patterns of which sequences of POS tags should make up a base noun phrase.

First, we define a base Noun Phrase chunk that consists of an optional determiner, followed by 0 or more adjectives, ending in a single common noun. We'll call it "cp" for chunk parser. Note that each possible POS tag is given inside < > tag brackets.

```
>>> cp = nltk.RegexpParser("NP: {<DT>?<JJ>*<NN>}")
```

Next we want to test this chunk parser on a sentence. The chunk parser assumes that you'll give it a list of tagged tokens, i.e. a list of pairs, where every pair is a word and a POS tag.

```
>>> tagged_tokens = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"),("dog", "NN"),
("barked", "VBD"), ("at", "IN"),  ("the", "DT"), ("cat", "NN")]
```

When we apply the parse function of the chunk parser, we get back a parsed tree, which we can print or draw.

```
>>> senttree = cp.parse(tagged_tokens)
>>> type(senttree)
<class 'nltk.tree.Tree'>
>>> print senttree
>>> senttree.draw()
```

Now, we want to extend the regular expressions to identify more types of base noun phrases. For example, so far we have said that a base noun phrase can have an optional determiner and some number of adjectives followed by a noun, and we have seen that it will match phrases like "the little yellow dog" and "a cat". But what if the noun phrase has a possessive, as in "my big cat". The POS tag for possessives is 'PP$' and it never occurs with a determiner, just instead of a determiner. So we change our regular expression, and we use the triple quote string notation so we can add a comment on each line.

```
>>> NPgrammar1 = r"""
```

```
  NP: {<DT|PP\$>?<JJ>*<NN>}   # determiner/possessive, adjectives and nouns
"""
```

We can test our regular expression chunk parser on more than one sentence at a time by testing it on the Penn Treebank sentences themselves. We define a new chunk parser.

```
>>> cp1 = nltk.RegexpParser(NPgrammar1)
```

We create an NLTK ChunkScore object that will test sentences on gold standard chunks and save the results in a score structure.

```
>>> chunkscore = nltk.chunk.ChunkScore()
```

Next we get the first 5 files of gold standard chunked sentences from Penn Treebank chunked, run the parse function on each sentence (flatten() removes the chunk structure), and run the score function which compares the resulting parse with the gold standard and saves the score in the ChunkScore object.

```
>>> for fileid in nltk.corpus.treebank_chunk.fileids()[:5]:
        for chunk_struct in nltk.corpus.treebank_chunk.chunked_sents(fileid):
           # runs the chunker cp on the sentences without chunks
                test_sent = cp1.parse(chunk_struct.flatten())
                # compares them with the gold standard chunks
                chunkscore.score(chunk_struct, test_sent)
```

The ChunkScore gives the result in terms of IOB Accuracy, precision, recall and F-measure, where
IOB Accuracy is the accuracy of the base noun phrase boundaries
Precision is the percentage of parsed noun phrases that were correct
Recall is the percentage of gold parsed noun phrases that were parsed
F-Measure is an average of precision and recall

```
>>> print chunkscore
```

The ChunkScore also will give examples of those that were missed (False Negatives) and those that were incorrect (False Positives).

```
>>> missed = chunkscore.missed()
>>> len(missed)

# look at the first 20 missed
>>> for m in missed[:20]:
        print m

# or we can use a random shuffle to look at a random set
>>> from random import shuffle
```

```
>>> shuffle(missed)
>>> for m in missed[:20]:
        print m
```

We can do the same for the incorrect examples.

```
>>> incorrect = chunkscore.incorrect()
>>> for m in incorrect[:20]:
        print m
```

Note that some of the correct look like they could be base noun phrases, but they are incorrect because they are part of a longer base noun phrase that was missed.

```
>>> shuffle(incorrect)
>>> for m in incorrect[:20]:
        print m
```

Let's look at the list of missed (False Negatives) and see what else we can add to our regular expressions. We can see that some noun phrases end in NN but also in a plural noun NNS:

```
        (NP six-month/JJ Treasury/NNP bills/NNS)
```

And we can see that there are noun phrases consisting of proper nouns:

```
        (NP Lorillard/NNP Inc./NNP)
```

So we add the option of having NNS instead of NN at the end of the first rule, and we add a second rule to match just proper nouns.

```
>>> NPgrammar2 = r"""
  NP: {<DT|PP\$>?<JJ>*<NN|NNS>}   # determiner/possessive, adjectives and nouns
      {<NNP>+}                # sequences of proper nouns
"""

>>> cp2 = nltk.RegexpParser(NPgrammar2)
>>> chunkscore2 = nltk.chunk.ChunkScore()
>>> for fileid in nltk.corpus.treebank_chunk.fileids()[:5]:
        for chunk_struct in nltk.corpus.treebank_chunk.chunked_sents(fileid):
            test_sent = cp2.parse(chunk_struct.flatten())
            chunkscore2.score(chunk_struct, test_sent)

>>> print chunkscore2
```

We improved our Recall score a lot! We can again look at missed and incorrect.

We see that we need to have rules to deal with noun phrases that have other modifiers besides adjectives, like NNP and VBG:

```
        (NP the/DT few/JJ industrialized/VBN nations/NNS)
        (NP all/DT remaining/VBG uses/NNS)
        (NP 160/CD workers/NNS)
```

(NP large/JJ burlap/NN sacks/NNS)

Other types of noun phrases end in something besides a noun NN, NNS or NNP:

(NP that/WDT)
(NP the/DT 1950s/CD)
(NP he/PRP)

Again, the incorrect look like parts of longer phrases that were missed and not truly incorrect, so we won't work on those.

```
>>> NPgrammar3 = r"""
NP: {<RB|DT|PP\$|PRP\$>?<JJ.*>*<VBN|VBG|NNP|CD>*<NN|NNS>+}
   {<DT>?<CD>+}
   {<DT>?<NNP>+}
   {<DT>+}
   {<WP>+}
   {<PRP>+}
   {<EX>+}
   {<WDT>+}
"""

>>> cp3 = nltk.RegexpParser(NPgrammar3)
>>> chunkscore3 = nltk.chunk.ChunkScore()
>>> for fileid in nltk.corpus.treebank_chunk.fileids()[:5]:
        for chunk_struct in nltk.corpus.treebank_chunk.chunked_sents(fileid):
            test_sent = cp3.parse(chunk_struct.flatten())
            chunkscore3.score(chunk_struct, test_sent)

>>> print chunkscore3
```

Or try this even higher scoring one:

```
NPgrammar3 = r"""
NP:
   {<DT>?<JJ|JJR|VBN|VBG>*<CD><JJ|JJR|VBN|VBG>*<NNS|NN>+}
   {<DT>?<JJS><NNS|NN>?}
   {<DT>?<PRP|NN|NNS><POS><NN|NNP|NNS>*}
   {<DT>?<NNP>+<POS><NN|NNP|NNS>*}
   {<DT|PRP\$>?<RB>?<JJ|JJR|VBN|VBG>*<NN|NNP|NNS>+}
   {<WP|WDT|PRP|EX>}
   {<DT><JJ>*<CD>}
   {<\$>?<CD>+}
"""
```

To continue the development, we should take into consideration all 200 files of Penn Treebank and not just the first 5 files. Even with all the files, scores in the low 90's are achievable.

One problem that we would run into is that sometimes the gold standard is incorrect due to human error.

In this example from the gold standard:

```
  (NP the/DT five/CD surviving/VBG workers/NNS)
  have/VBP
  (NP asbestos-related/JJ diseases/NNS)
  ,/,
  including/VBG
  (NP three/CD)
  with/IN
  recently/RB
  diagnosed/VBN
  (NP cancer/NN)
  ./.)
```

The last five words of this sentence should have been tagged as follows:

```
(NP three/CD)
  with/IN
(NP recently/RB diagnosed/VBN cancer/NN)
  ./.)
```

Our rules will get the latter longer NP, and the chunkscore will say that we are wrong and that we are missing (NP cancer/NN).

Techniques for Chunking using the Annotated Data for Training: N-gram chunker

Each year, CoNLL, the Conference on Natural Language Learning, has a shared task for which annotated data is provided for training and development of the task. In the year 2000, the task was to chunk noun phrases and this corpus is in the NLTK. A few sentences are available as 'train.txt' in a tree structure of the chunks.

```
>>> from nltk.corpus import conll2000
>>> conll2000.chunked_sents('train.txt')
```

One representation of chunks is the IOB format. In this representation, each word is notated as either B (beginning a chunk), I (internal to a chunk), or O (outside of a chunk). The nltk.chunk.tree2conlltags maps the annotated chunk trees to this IOB format, where each word is represented by a triple of the word, the POS tag, and the chunk notation. Get word,tag,chunk triples from the CoNLL 2000 corpus and map these to tag,chunk pairs

```
>>> chunk_data = [[(t,c) for w,t,c in nltk.chunk.tree2conlltags(chtree)]
                for chtree in conll2000.chunked_sents('train.txt')]
```

Look at the first sentence to see the IOB tag format:

```
>>> print chunk_data[0]
```

One way to define a chunker is to essentially use POS tagging classifier techniques to learn the tags containing IOB prefixes.

NLTK can train and score a unigram chunker, similar to a unigram tagger, by collecting frequencies for which POS tags have which chunk labels. Although the chunker itself does not take too long to train, the tagging accuracy function takes several minutes.

```
unigram_chunker = nltk.UnigramTagger(chunk_data)
>>> print unigram_chunker.evaluate(chunk_data)
0.781378851068
```

And similarly, we could do a bigram chunker trained on two words sequences with POS tags, but this also takes a while.

```
>>> bigram_chunker = nltk.BigramTagger(chunk_data, backoff=unigram_chunker)
>>> print bigram_chunker.evaluate( chunk_data)
0.89312652614
```

Lessons Learned about Chunking

For shallow parsing tasks, including base noun phrases and other chunking,
- regular expressions using just POS tags works very well and
- POS tagging techniques with bigrams using words and POS tags works well.

For more complex parsing structures, these techniques will not work!

WordNet in NLTK

WordNet is imported from NLTK like other corpus readers and more details about using WordNet can be found in the NLTK book in section 2.5 of chapter 2. Remember that you can browse WordNet on-line at http://wordnetweb.princeton.edu/perl/webwn or you can use the NLTK wordnet browser by opening a command prompt (or terminal) window, typing “python” to get a separate python environment. Then type “import nltk” and “nltk.app.wordnet()”, and nltk should open your default browser to a wordnet browse page.

Back in our IDLE window, for convenience in typing examples, we can shorten its name to ‘wn’.

```
>>> from nltk.corpus import wordnet as wn
```

Synsets and lemmas

Although WordNet is usually used to investigate words, its unit of analysis is called a synset, representing one sense of the word. For an arbitrary word, i.e. dog, it may have different senses, and we can find its synsets. Note that **each synset is given an identifier** which includes **one** of the actual words in the synset, whether it is a noun, verb, adjective or adverb, and a number, which is relative to all the synsets listed for the particular actual word.

While using the wordnet functions in the following section, it is useful to search for the word 'dog' in the on-line WordNet at http://wordnetweb.princeton.edu/perl/webwn

```
>>> wn.synsets('dog')
```

Once you have a synset, there are functions to find the information on that synset, and we will start with "lemma_names", "lemmas", "definitions" and "examples".

For the first synset 'dog.n.01', which means the first noun sense of 'dog', we can first find all of its words/lemma names. These are all the words that are synonyms of this sense of 'dog'.

```
>>> wn.synset('dog.n.01').lemma_names
```

Given a synset, find all its lemmas, where **a lemma is the pairing of a word with a synset**.

```
>>> wn.synset('dog.n.01').lemmas
```

Given a lemma, find its synset

```
>>> wn.lemma('dog.n.01.domestic_dog').synset
```

Given a word, find lemmas contained in all synsets it belongs to

```
>>> for synset in wn.synsets('dog'):
    print synset, ": ", synset.lemma_names
```

Given a word, find all lemmas involving the word. Note that these are the synsets of the word 'dog', but just also showing that 'dog' is one of the words in each of the synsets.

```
>>> wn.lemmas('dog')
```

Definitions and examples:

The other functions of synsets give the additional information of definitions and examples. Find definitions of the synset for the first sense of the word 'dog':

```
>>> wn.synset('dog.n.01').definition
```

Display an example use of the synset

```
>>> wn.synset('dog.n.01').examples
```

Or we can show all the synsets and their definitions:

```
>>> for synset in wn.synsets('dog'):
    print synset, ": ", synset.definition
```

Lexical relations

WordNet contains many relations between synsets. In particular, we quite often explore the hierarchy of WordNet synsets induced by the hypernym and hyponym relations. (These relations are sometimes called "is-a" because they represent abstract levels of what things are.) Take a look at the WordNet Hierarchy diagram, Figure 2.11, in section 2.5 WordNet of the NLTK book.

Find hypernyms of a synset of 'dog':

```
>>> dog1 = wn.synset('dog.n.01')
 >>> dog1.hypernyms()
```

Find hyponyms

```
 >>> dog1.hyponyms()
```

We can find the most general hypernym as the root hypernym

```
>>>dog1.root_hypernyms()
```

There are other lexical relations, such as those about part/whole relations. The components of something are given by meronymy; NLTK has two functions for two types of meronymy, part_meronymy and substance_meronomy. It also has a function for things they are contained in, member_holonymy.

NLTK also has functions for antonymy, or the relation of being opposite in meaning. Antonymy is a relation that holds between lemmas, since words of the same synset may have different antonyms.

```
>>> good1 = wn.synset('good.a.01')
>>> wn.lemmas('good')
>>> good1.lemmas[0].antonyms()
```

Another type of lexical relation is the entailment of a verb (the meaning of one verb implies the other)

```
>>> wn.synset('walk.v.01').entailments()
```

There are more functions to use hypernyms to explore the WordNet hierarchy. In particular, we may want to use paths through the hierarchy in order to explore word similarity, finding words with similar meanings, or finding how close two words are in meaning.

We can use hypernym_paths to find all the paths from the first sense of dog to the root, and list the synset names of all the entities along those two paths.

```
dog1.hypernyms()
paths=dog1.hypernym_paths()
len(paths)
[synset.name for synset in paths[0]]
[synset.name for synset in paths[1]]
```

Exercise:

1. Pick a word and show all the synsets of that word and their definitions.
2. Pick one synset of the word and show all of its hypernyms.
3. Show the hypernym path between the top of the hierarchy and the word.

Put the results of your three steps into the discussion for this week in the iLMS, along with any other interesting examples (as long as they are not too lengthy!). Please put your **word in the title of your post**.