NLP Lab Session Week 8
March 6, 2013

Noun Phrase Chunking in NLTK
and Dependency Grammars and Using the Stanford Parser (if time)

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]

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

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.

If we look at just 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 = nltk.corpus.treebank_chunk.chunked_sents(fileid)[0]
>>> type(s0)
<class 'nltk.tree.Tree'>
>>> s0.draw()
```

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 not building the chunker from any of the annotated data; 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]
```

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
- bigrams using words and POS tags works well.

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

Possible Activities leading to an Exercise, if time permits:

Dependency Grammar Parser in NLTK

In the last lab, we looked at how the NLTK can define a parsing algorithm from a Context-Free Grammar, using either the Recursive Descent or Shift-Reduce parsing algorithms.

NLTK also has a dependency parser for projective sentences. For that, we need to make a dependency grammar that shows the dependency relation between words. Note that this is an **unlabeled dependency grammar**.

```
>>> groucho_dep_grammar = nltk.parse_dependency_grammar("""
'shot' -> 'I' | 'elephant' | 'in'
'elephant' -> 'an' | 'in'
'in' -> 'pajamas'
'pajamas' -> 'my'
""")
```

In this grammar, each production (or rewrite rule) represents an unlabeled dependency; we can view them individually by printing the grammar.

```
>>> print groucho_dep_grammar
```

Next, we create an NLTK Projective Dependency Parser using this grammar.

```
>>> pdp = nltk.ProjectiveDependencyParser(groucho_dep_grammar)
```

Define a sentence and apply the parser to the sentence.

```
>>> sentlist = "I shot an elephant in my pajamas".split()
>>> trees = pdp.parse(sentlist)
>>> for tree in trees:
    print tree
```

In this flat representation of a dependency tree, each node in the tree is represented by words in parentheses, where the first word is the node label and the remaining items are the children of the node. The edges of the tree are unlabelled dependencies. We'll draw these trees in the lab.

Annotated syntax trees

In the first part of this lab, we used a version of the Penn Treebank data that was annotated with chunks.

But in the original Penn Treebank, the sentences were annotated with full parses. Recall that these were hand annotated and can be used to make context free grammars. For help in understanding the parse tree node non-terminals (which Penn Treebank calls tags), there is an overview at http://bulba.sdsu.edu/jeanette/thesis/PennTags.html, and more detailed information at the Penn Treebank tagging guide, "Bracketing Guidelines for Treebank II Style Penn Treebank Project", which is a detailed annotation document.

```
>>> from nltk.corpus import treebank

>>> t0 = treebank.parsed_sents('wsj_0001.mrg')[0]
>>> print t0
```

(Note that here, the NLTK **print** function puts out a more visual notation for a parse tree than observing the tree value directly.)

We'll also look at the next 2 or 3 sentences. It will be easier to cycle through the sentences if we use the default argument.

```
>>> t1 = treebank.parsed_sents()[1]
>>> print t1
```

etc.

In looking at these trees, we observe that there are embedded S tags of several kinds. For some examples, see the introduction to Chapter 8 of the NLTK book and the section on Clause Types on page 16 of the Treebank guidelines. (For simple definitions of grammatical concepts such as complement and complementizer, wikipedia is a helpful source.)

For information on NULL and TRACE elements, see page 60 of the Treebank guidelines.

NLTK also has some methods to help extract grammars from Penn Treebank. We'll look at some examples in chapter 8 showing tools for prepositional attachment and for examining particular verb constructions.

Exercise in understanding treebank annotations and parser output:

Choose a different sentence from the Penn Treebank (randomly pick a sentence number) and print the parse tree. Examine the parse tree to understand it, or choose another one if you randomly got an uninteresting sentence.

Now open the Stanford parser demo at http://nlp.stanford.edu:8080/parser/. Use your Penn Treebank sentence or make up a sentence of suitable length and complexity. The sentence should have approximately 8 – 12 words. Run it in the Stanford parser demo window, but if the parse looks too complex, choose a simpler or shorter sentence.

Now the parse will look something like this (ignoring Root) for the simple example of "Bob saw a man":

```
(ROOT
   (S
     (NP (NNP Bob))
     (VP (VBD saw)
       (NP (DT a) (NN man)))
     (. .)))

 nsubj(saw-2, Bob-1)
 root(ROOT-0, saw-2)
 det(man-4, a-3)
 dobj(saw-2, man-4)
```
where the numbers are the order of the words in the sentence.

1. Draw the parse tree of your example as an actual tree on a piece of paper.
2. Now draw the dependency tree as an actual tree on a piece of paper.

The goal of this exercise is to understand how parse output represents the structure of sentences on some more complex and realistic examples. Unfortunately, you won't be able to submit this to a discussion and I'll just make a quick observation of your results.

[Optional: if you're curious about collapsed dependencies, see the example at http://nlp.stanford.edu/software/stanford-dependencies.shtml where the prepositions are collapsed into the dependency types.]