NLP Lab Session Week 7
October 13, 2011
Chunking in NLTK

Installing NLTK Toolkit

Reinstall nltk_data if needed.

Getting Started

As we did in the last lab session, we will work together through a series of small examples using the IDLE window 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.

Labweek7chunking.py

Open an IDLE window. Use the File-> Open to open the labweek7chunking.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.

Build a simple chunker

The first approach to chunking in the NLTK uses regular expressions for the patterns of the chunks. Before starting, we need to:

```
>>> import nltk
>>> from nltk.corpus import conll2000
```

The chunkers work on text that already has POS tags and define patterns of POS tags that can make up the chunk phrase. Here is a sentence for testing.

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

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. 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.

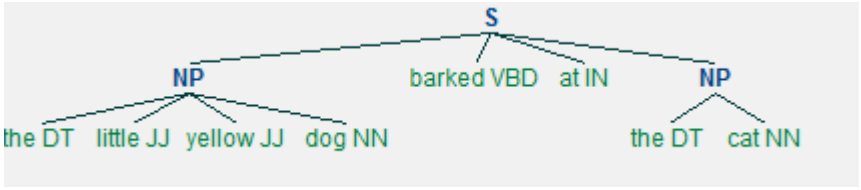
```
>>> cp = nltk.RegexpParser("NP: {<DT>?<JJ>*<NN>}")
```

We can run this chunk parser, cp, on the tagged sentence and observe the resulting chunks.

```
>>> print cp.parse(tagged_tokens)
```

Or we can also use the drawing package to view the results as a tree. The draw method opens a graphical window that could be hiding behind other windows, so look around!

>>> cp.parse(tagged_tokens).draw()



Larger Regular expression chunker

This chunker has two regular expression patterns: one for noun phrases as before, but also allowing pronoun possessive words (tagged PP$, so we must write PP\$ in the regular expression) and also a pattern to make proper noun sequences.
(By the way, the notation of these strings comes from Python: the r in front of a string denotes a raw string which leaves in \, and the rest is a triply quoted string, which allows you not to have to escape quotes.)

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

Use the grammar to make a Regular Expression chunker:

```
>>> cp = nltk.RegexpParser(NPchunkgrammar)
```

Input some tokenized and POS tagged text

```
>>> tagged_tokens2 = [("Rapunzel", "NNP"), ("let", "VBD"), ("down", "RP"),
                      ("her", "PP$"), ("long", "JJ"), ("golden", "JJ"), ("hair", "NN")]
```

Display chunking results, or draw it as before.

```
>>> print cp.parse(tagged_tokens2)
>>> cp.parse(tagged_tokens2).draw()
```

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.

```
>>> 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]
```

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)
```

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)
```

Instead of an exercise today, we will do the examples related to the homework assignment, given in Homework2chunker.py and in Homework2.contentextracter.py.