NLP Lab Session Week 6
October 6, 2011

Parsing and using Grammars in NLTK

Installing NLTK data

If needed, do an nltk.download() and choose nltk_book and save on the H: drive if you have space. Or, if you prefer, I can give you the dataset on a memory stick.

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 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 can be cut-and-paste to the IDLE window to try it out.

As always, we start by importing from nltk all of the programs

```
import nltk
```

Running parsing demos

The first parsing demo shows the recursive descent parser, which is a top-down, back-tracking parser. The second shows the shift-reduce parser, which is a bottom-up parser and needs guidance as to what operation (shift or reduce) to apply at some steps. The third shows a chart parser in top-down strategy (1); it also has strategies for bottom-up, bottom-up left corner and stepping.

We already looked at these two in class.

```
nltk.app.rdparser()
nltk.app.srparser()
```

Here is a chart parser demo. You can omit the first argument to see the parser choices.

```
nltk.parse.chart.demo(1, should_print_times=False, trace=1)
```

Running Parsers

In NLTK, the parsers that are provided all need a grammar to operate. The parse_cfg function is given to take a normal string representation of a CFG grammar and convert it to a form that the parsers can use. Here is an example:

```
>>> grammar = nltk.parse_cfg("""
S -> NP VP
VP -> V NP | V NP PP
PP -> P NP
V -> "saw" | "ate" | "walked"
NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
Det -> "a" | "an" | "the" | "my"
N -> "man" | "dog" | "cat" | "telescope" | "park"
P -> "in" | "on" | "by" | "with"
""")
```

First, we define a recursive descent parser from this grammar and then test it on a short sentence. The recursive descent parser is further described in the NLTK book in section 8.5.

```
>>> rd_parser = nltk.RecursiveDescentParser(grammar)
```

Note that another way to tokenize a string is to use the Python “split” function. With no argument, it will produce a list of tokens that were separated by white space. (You can also put a regular expression argument to say what string to split on, but the result leaves out whatever matches.)

```
>>> sent = "Mary saw Bob".split()
>>> for tree in rd_parser.nbest_parse(sent):
        print tree
```

Note that this parser returns n (all?) the parse trees, so we can try this out on a syntactically ambiguous sentence.

```
>>> sent2 = "John saw the man in the park with a telescope".split()
>>> for tree in rd_parser.nbest_parse(sent2):
        print tree
```

If you try other sentences, don’t put the punctuation at the end because we didn’t include any punctuation in the grammar.

We can add words to our grammar in order to parse other sentences.

```
groucho_grammar = nltk.parse_cfg("""
S -> NP VP
VP -> V NP | V NP PP
PP -> P NP
V -> "saw" | "ate" | "walked" | "shot"
NP -> "John" | "Mary" | "Bob" | "I" | Det N | Det N PP
Det -> "a" | "an" | "the" | "my"
N -> "man" | "dog" | "cat" | "telescope" | "park" | "elephant" | "pajamas"
P -> "in" | "on" | "by" | "with"
""")
```

Next we make a shift-reduce parser from the groucho grammar and test it on a simple sentence. The shift-reduce parser is also further described in section 8.5 of the NLTK book.

```
sr_parse = nltk.ShiftReduceParser(groucho_grammar)
```

```
sent3 = 'Mary saw a dog'.split()
print sr_parse.parse(sent3)
```

Next we test it on a more complicated sentence, but it doesn't find a parse tree because its automatic selection of shift-reduce operators is not sophisticated enough.

```
sent4 = "I shot an elephant in my pajamas".split()
print sr_parse.parse(sent4)
```

If you like, try making a recursive descent parser with the groucho grammar and observe the trees. (Note that we were careful not to include rewrite rules such as VP -> VP PP, because that would involve an infinite recursion in the recursive descent parser.)

If you want to work on grammar development, the NLTK also provides a function that will load a grammar from a file, so that you can keep your grammar rules in a text file.

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'
""")
```

Now we can make a dependency parser and test it.

```
pdp = nltk.ProjectiveDependencyParser(groucho_dep_grammar)
trees = pdp.parse(sent4)
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.

Annotated syntax trees

From Penn Treebank, we can view the syntax trees of the sentences. 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.

```
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 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 12 – 20 words. Run it in the Stanford parser demo window.

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

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

1. Draw this parse tree 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.