NLP Lab Session Week 7
October 9, 2013

Parsers with simple grammars 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.

Labweek7examples.py

Open an IDLE window. Use the File-> Open to open the labweek7examples.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.

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.

See section 8.4 for a description of Recursive Descent parsing, Shift Reduce parsing and Chart Parsing, including the chart data structures, called Well-Formed Substring Tables in NLTK.

We already looked at these two in class.
nltk.app.rdparser()
Note that in the recursive descent parser you can use Edit Text to parse different sentences, but I couldn't get Edit Grammar to work.

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 with Context Free Grammars

In NLTK, the parsers that are provided all need a grammar to operate, so they are limited by what we can write down as grammars. 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.)

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

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

```
>>> sent2list = "John saw the man in the park with a telescope".split()
>>> for tree in rd_parser.nbest_parse(sent2list):
        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 or doesn't include any backtracking.

```
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.

We will look at the Recursive grammar in Section 8.3 of the NLTK book. Note that it has examples of direct recursion, e.g. in the rule/production

Nom -> Adj Nom

It also has indirect recursion as in the two rules

S -> NP VP
VP -> V S

so that a sentence can occur as a sub-tree in the parse tree of another sentence.

To demonstrate further development of this grammar, suppose that we want to be able to parse sentences like "Book that flight". This was an example that we saw in the lecture slides that used an example from the textbook for a flight grammar subset of English. For this grammar, we need for sentences to be just a Verb Phrase, and we need the three words to be included where "book" is a verb, "that" is a determiner, and "flight" is a noun.

```
>>> flight_grammar = nltk.parse_cfg("""
  S -> NP VP | VP
  VP -> V NP | V NP PP
```

```
  PP -> P NP
  V -> "saw" | "ate" | "walked" | "shot" | "book"
  NP -> "John" | "Mary" | "Bob" | "I" | Det N | Det N PP
  Det -> "a" | "an" | "the" | "my" | "that"
  N -> "man" | "dog" | "cat" | "telescope" | "park" | "elephant" | "pajamas" | "flight"
  P -> "in" | "on" | "by" | "with"
  """)
```

We can redefine the recursive descent parser with this grammar and use it on our sentence.

```
>>> rd_parser = nltk.RecursiveDescentParser(flight_grammar)
>>> sent5list = 'book that flight'.split()
>>> for tree in rd_parser.nbest_parse(sent5list):
        print tree
```

For developing grammars, we added words that we needed directly to the grammars. In practice, we would use a developed lexicon of words with their possible POS tags.

Running Parsers with Dependency Grammars

Next, we look at parsers built from dependency grammars. These grammars show the relationships between individual words. Ideally, we would like to have labeled relationships, but the NLTK dependency grammars have just unlabeled relationships. This is described in Section 8.5 of the NLTK book. Here is a grammar for the groucho example.

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

We can try this out on our ambiguous sentence and look at the trees that it gets. The parser for dependency grammars in NLTK will only parse projective sentences, that is, sentences where the dependencies are non-crossing.

```
print groucho_dep_grammar
pdp = nltk.ProjectiveDependencyParser(groucho_dep_grammar)
glist = 'I shot an elephant in my pajamas'.split()
trees = pdp.parse(glist)
for tree in trees:
    print tree
```

This example dependency grammar shows why people do not develop grammars directly because you have to represent all words that can be involved in a relationship. Instead, we would learn the possible dependencies from a corpus, and using some rules for unknown words.

Probabilistic Context-Free Grammars and Subcategories of Verbs

In this section, we give examples of two different ideas. The first is the idea of subcategories of verbs. Some of the subcategories in English are:

transitive verbs, such as 'saw' and 'chased', require an NP direct object
    The cat saw the dog.
    The dog chased the squirrel.
intransitive verbs do not take any object
    The dog barked.
dative verbs have two objects, expressed in grammar as either two objects or a direct object and a prepositional phrase
    He gave John the book.
    He gave a dog to a man.
sentential verbs are followed by a sentential construct
    He said that a dog barked.

In addition, there may be optional modifers, such as adverbs, and auxiliary verbs for some verb tenses, that we won't go into here.
    The squirrel was really frightened.
    The man really saw a bear.
    The man really thought the bear was angry.

In our next grammar example, we will split some of the verb rules into subcategories.

The other idea that we're going to demonstrate here is that of the probabilistic grammar. In these grammars, each rule is associated with the probability that the left-hand-side symbol is rewritten using that particular rule. The probabilities for each non-terminal symbol must add up to 1.

```
prob_grammar = nltk.parse_pcfg("""
  S -> NP VP [0.9]| VP  [0.1]
  VP -> TranV NP [0.4]
  VP -> InV  [0.3]
  VP -> DatV NP PP  [0.3]
  PP -> P NP   [1.0]
  TranV -> "saw" [0.2] | "ate" [0.2] | "walked" [0.2] | "shot" [0.2] | "book" [0.2]
  InV -> "ate" [0.5] | "walked" [0.5]
  DatV -> "gave"   [1.0]
```

```
  NP -> "John" [0.05]| "Mary" [0.05] | "Bob" [0.05] | "I" [0.05] | Det N [0.4] | Det N PP
[0.4]
  Det -> "a" [0.2] | "an" [0.2] | "the" [0.2] | "my" [0.2] | "that" [0.2]
  N -> "man" [0.2] | "dog" [0.2] | "cat" [0.2] | "telescope" [0.2] | "flight" [0.2]
  P -> "in" [0.2] | "on" [0.2] | "by" [0.2] | "with" [0.2] | "through" [0.2]
  """)
```

The NLTK provides a parser called ViterbiParser to parse using probabilistic CFGs:

```
viterbi_parser = nltk.ViterbiParser(prob_grammar)
print viterbi_parser.parse(['John', 'saw', 'a', 'telescope'])
```

Exercise

Starting with the flight grammar, add the CFG rules to parse one or more of the following sentences. You may work in groups. (Note that I have left the "." off the end of the sentence.)

I prefer a flight through Houston
Jack walked with the dog
I want to book that flight
John gave the dog a bone

Post your revised pattern to the Discussion in the iLMS for Week 7 with the sentence(s) that it parses.