NLP Lab Session Week 6
October 5, 2017

Context Free Grammars and Parsers with simple grammars in NLTK

Getting Started

For this lab session download the examples: LabWeek6CFGparse.txt and put it in your class folder for copy/pasting examples. Start your Python interpreter session.

Parsing demos

Recall that we mentioned the NLTK parsing demos during the lectures, and these are described in Chapter 8 of the NLTK book, section 8.4. In lecture, we looked at the parsing demo for the recursive descent parser, which is a top-down, back-tracking parser. [In my experience, these do not run on some versions of Mac OS, but do run in the labs.]

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

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.

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

The third shows a chart parser in top-down strategy (1); it also has strategies for bottom-up, bottom-up left corner and stepping. Section 8.4 also has a description of Chart Parsing, including the chart data structures, called Well-Formed Substring Tables in NLTK. Here is one way to run the 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 focus is on understanding how grammars represent language using classical algorithms and not on details of modern parsing algorithms using Treebanks. The parsers that are provided all need a grammar to operate, so they are limited by what we can write down as grammars. [For short examples of all the parsers, see the HOWTO parse module document: http://www.nltk.org/howto/parse.html] 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, similar to the one in the NLTK book, except that I separated the proper nouns:

```
>>> grammar = nltk.CFG.fromstring("""
S -> NP VP
VP -> V NP | V NP PP
PP -> P NP
V -> "saw" | "ate" | "walked"
NP -> Prop | Det N | Det N PP
Prop -> "John" | "Mary" | "Bob"
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 test it on a short sentence. The recursive descent parser is described in the NLTK book in section 8.4. Unlike the demo recursive descent parser, this one does backtracking, finds all the parse trees and prints them.

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

When we define example sentences to test our parsers, we can use the normal NLTK function nltk.word_tokenize() for tokenization, but a shorter way to tokenize a string with just words and no special symbols 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()
>>> trees = rd_parser.parse(sentlist)
```

The parser returns a generator for the list of all trees that it found. The NLTK has a type called nltk.tree.Tree for each tree. In order to see the type, we first convert the generator to a list, and then we can look at individual elements.

```
>>> treelist = list(trees)
>>> type(treelist[0])
>>> 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. Here we just iterate over the generator to print all the trees.

```
>>> sent2list = "John saw the man in the park with a telescope".split()
>>> for tree in rd_parser.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. In this grammar, we added “I” to the proper nouns (or we could have added another category for pronouns) and the common nouns “elephant” and “pajamas”.

```
groucho_grammar = nltk.CFG.fromstring("""
  S -> NP VP
  VP -> V NP | V NP PP
  PP -> P NP
  V -> "saw" | "ate" | "walked" | "shot"
  NP -> Prop | Det N | Det N PP
  Prop -> "John" | "Mary" | "Bob" | "I"
  Det -> "a" | "an" | "the" | "my"
  N -> "man" | "dog" | "cat" | "telescope" | "park" | "elephant" | "pajamas"
  P -> "in" | "on" | "by" | "with"
  """)
>>> sent4list = "I shot an elephant in my pajamas".split()
>>> rd_parser = nltk.RecursiveDescentParser(groucho_grammar)
>>> for tree in rd_parser.parse(sent4list):
    print (tree)
```

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.CFG.fromstring("""
  S -> NP VP | VP
  VP -> V NP | V NP PP
  PP -> P NP
  V -> "saw" | "ate" | "walked" | "shot" | "book"
  NP -> Prop | Det N | Det N PP
  Prop -> "John" | "Mary" | "Bob" | "I"
  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.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.

Note that the NLTK has shift-reduce parsers as well, but as we noted in the parser demo, it doesn't have backtracking so it doesn't always find parse trees. The shift-reduce parser is also further described in section 8.4 of the NLTK book.

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.

Dependency Grammars

NLTK also allows you to write dependency grammars, which just have to 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. DependencyGrammar.fromstring ("""
  '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. We will use the parser for dependency grammars in NLTK that will only parse projective sentences, that is, sentences where the dependencies are non-crossing. (There is another parser NonProjectiveDependencyParser.)

```
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)
```

Here are the trees it produces:

```
(shot I (elephant an (in (pajamas my))))
(shot I (elephant an) (in (pajamas my)))
```

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.
  The bear should have chased the squirrel.

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. Note that I put in a rule to allow dative verbs to have a NP PP, but I omitted the case of NP NP.

```
prob_grammar = nltk.PCFG.fromstring("""
  S -> NP VP [0.9]| VP  [0.1]
  VP -> TranV NP [0.3]
  VP -> InV  [0.3]
  VP -> DatV NP PP  [0.4]
  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" [0.2] | "ate" [0.2] | "saw" [0.2] | "walked" [0.2] | "shot" [0.2]
  NP -> Prop [0.2]| Det N [0.4] | Det N PP [0.4]
  Prop -> "John" [0.25]| "Mary" [0.25] | "Bob" [0.25] | "I" [0.25]
  Det -> "a" [0.2] | "an" [0.2] | "the" [0.2] | "my" [0.2] | "that" [0.2]
  N -> "man" [0.15] | "dog" [0.15] | "cat" [0.15] | "park" [0.15] | "telescope" [0.1] | "flight"
[0.1] | "elephant" [0.1] | "pajamas" [0.1]
  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)
for tree in viterbi_parser.parse(['John', 'saw', 'a', 'telescope']):
   print (tree)

for tree in viterbi_parser.parse(sent2list):
   print (tree)

for tree in viterbi_parser.parse(sent4list):
   print (tree)
```

Exercise

Starting with the **flight grammar**, add the CFG rules to parse at least three of the following sentences. (Note that I have left the "." off the end of each sentence.) First try using the rd_parser defined from the flight grammar to parse the sentence and then add words and/or add rules to the grammar. The first sentence is the easiest and the last sentence is probably the hardest.

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

You may want to use the Stanford demo parser to get ideas of parse structure, but note that their parse structures may be more complex than we need for our examples.

Post your revised grammar under the Assignments tab in Blackboard for Lab Week 6, along with the example parses for each sentence.