NLP Lab Session Week 7
February 27, 2013

Parsers with simple grammars in NLTK and Revisiting POS tagging

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.

We already looked at these two in class.

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

But note that in the recursive descent parser you can use Edit Text to parse different sentences and even Edit Grammar to add or change grammar rules.

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, 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.

Revisiting POS taggers

From the Penn Treebank, we get almost 4,000 sentences of POS tagged data.

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

>>> treebank_tagged = treebank.tagged_sents()
>>> treebank_tagtext = treebank.tagged_words()
>>> treebank_text = treebank.words()
```

When we look at the first 50 entries of treebank_tagtext, we see that each item in the list is a (word, tag) pair.

```
>>> treebank_tagtext[:50]
```

Last time, we saw that NLTK has a structure called the Conditional Frequency Distribution, which is a nested dictionary. It allows you to map a (word, tag) pair to a frequency. NLTK calls the first set of keys the “conditions”, which index a regular frequency distribution with keys and values. Before using it in the findtags function, let’s look at the conditional distribution itself. We’ll make a conditional frequency distribution for all the words and tags.

```
cfd = nltk.ConditionalFreqDist((tag, word) for (word, tag) in treebank_tagtext)
```

To show the structure of this conditional frequency distribution, we look at the parts.

```
>>> cfd.keys()[:30]
['PRP$', 'VBG', 'VBD', '``', 'VBN', 'POS', "''", 'VBP', 'WDT', 'JJ', 'WP', 'VBZ', 'DT', '#',
'RP', '$', 'NN', 'FW', ',', '.', 'TO', 'PRP', 'RB', '-LRB-', ':', 'NNS', 'NNP', 'VB', 'WRB', 'CC']
```

Using the function cfd.conditions() is the same as keys().
Each key is a dictionary from words to frequencies

```
>>> NNdict = cfd['NN']
>>> NNdict.keys()[:30]
['%', 'company', 'year', 'market', 'trading', 'stock', 'president', 'program', 'share',
'government', 'business', 'price', 'index', 'time', 'yesterday', 'investment', 'week', 'issue',
'profit', 'interest', 'money', 'country', 'debt', 'chairman', 'number', 'plan', 'month', 'cash',
'group', 'vice']
>>> NNdict['%']
445
>>> NNdict['company']
260
```

Now we define the function that finds the most frequent words that are tagged with any of the noun tags, those prefixed with NN.

```
# find the most frequent words in Penn Treebank that have one of the noun tags
def findtags(tag_prefix, tagged_text):
	cfd = nltk.ConditionalFreqDist((tag, word) for (word, tag) in tagged_text
	                                    if tag.startswith(tag_prefix))
	return dict((tag, cfd[tag].keys()[:20]) for tag in cfd.conditions())
```

This function returns a python dictionary that has the tags beginning with tag_prefix as keys and the top 20 most frequent words with that tag as values.

```
>>> tagdict = findtags('NN', treebank.tagged_words())
>>> for tag in sorted(tagdict):
	print tag, tagdict[tag]
```

POS Tagging

We start by separating the tagged data into a training set and a test set. This allows us to test the tagger's accuracy on similar , but not the same, data that it was trained on. We take the first 90% of the data for the training set, and the remaining 10% for the test set.

```
>>> size = int(len(treebank_tagged) * 0.9)
>>> treebank_train = treebank_tagged[:size]
>>> treebank_test = treebank_tagged[size:]
```

As last time, we use a Bigram tagger that can be trained using 2 tag-word sequences. There will be unknown frequencies in the test data for the bigram tagger, and unknown words for the unigram tagger, so we can use the backoff tagger capability of NLTK to create a combined tagger. This tagger uses bigram frequencies to tag as much as possible. If a word doesn't occur in a bigram, it uses the unigram tagger to tag that word. If the word is unknown to the unigram tagger, then we use the default tagger to tag it as 'NN'.

```
>>> t0 = nltk.DefaultTagger('NN')
>>> t1 = nltk.UnigramTagger(treebank_train, backoff=t0)
>>> t2 = nltk.BigramTagger(treebank_train, backoff=t1)
>>> t2.evaluate(treebank_test)
```

Let's use this tagger to tag some example text. Define some example text, tokenize it, and apply the tagger.

```
>>> text = "A majority of respondents ( 51% ) believe the budget negotiations between
President Obama and congressional Republicans make them feel less confident about
economy, versus just 16% who feel more confident."
>>> tokens = nltk.wordpunct_tokenize(text)
>>> taggedtext = t2.tag(tokens)
>>> taggedtext
```

In order to look at some of the tags in a text, we can use the findtags function from the last section. (Note that a conditional frequency distribution first has a set of keys called the "conditions" that indexes a regular frequency distribution (with keys and values).)

```
>>> tagdictNN = findtags('NN', taggedtext)
#Find the most frequent words in this example that have one of the noun tags
>>> for tag in sorted(tagdictNN):
	print tag, tagdictNN[tag]
```

Exercise:

For this exercise, you are to choose any document, including any of the untagged documents in the Gutenberg corpus, run the bigram POS tagger, and find the top frequency words by POS tag class. As usual, you may work in groups.

(There is one caveat on using the bigram POS tagger on these texts, which is that it was trained on text which had been separated by sentences. So we should really run a sentence detector on our raw text first, and we'll talk about how to do that in the future.)

1. Choose a document, as we did for the word frequencies, and tokenize it. Do not lowercase the words or stem them. For example, if I chose emma, the first document in the Gutenberg corpus, I would get the raw text and tokenize it with wordpunct, ending up with something named emmatokens. (For details on getting a Gutenberg file, check LabWeek3examples.py.)

2. Now run the bigram pos tagger - it was named t2 in our development – and save the results in a variable (don't try to print the whole thing). For example, I would save mine in a variable called emmatagged.

3. Use the findtags function to define 4 dictionaries for nouns (prefix 'N'), verbs (prefix 'V'), adjectives (prefix 'J') and adverbs (prefix 'R'). Display these dictionaries.

4. Choose the results of one of your dictionaries and post it to the discussion for today's lab. Include the names of everyone in your group in the discussion title. The discussion entry should include the name of your document and what type of words you are giving.