Homework #4: CMPT-413

Reading: NLTK Tutorial (*http://nltk.sf.net/docs.html*); Chapter 5 (Sections 5.1 to 5.4.2) and Chapter 7

Distributed on Feb 21; due on Mar 7

Anoop Sarkar – anoop@cs.sfu.ca

Only submit answers for questions marked with †.

(1) Once we have some text that has been tagged with part of speech labels, we can *chunk* words together into non-overlapping spans of text. Each chunk corresponds to some meaningful unit, e.g. we can find all chunks that are *noun phrases*. Here is an example sentence from the Wall Street Journal where the noun phrases are marked up with brackets:

> [The/DT market/NN] for/IN [system-management/NN software/NN] for/IN [Digital/NNP 's/POS hardware/NN] is/VBZ fragmented/JJ enough/RB that/IN [a/DT giant/NN] such/JJ as/IN [Computer/NNP Associates/NNPS] should/MD do/VB well/RB there/RB ./.

In this question, we will use regular expressions on the part of speech tags to identify chunks. First, we need to put the input sentence into the right format for nltk_lite:

```
from nltk_lite import chunk
tagged_text = """
The/DT market/NN for/IN system-management/NN software/NN for/IN
Digital/NNP 's/POS hardware/NN is/VBZ fragmented/JJ enough/RB
that/IN a/DT giant/NN such/JJ as/IN Computer/NNP Associates/NNPS
should/MD do/VB well/RB there/RB ./.
"""
input = chunk.tagstr2tree(tagged_text)
print input
```

Examine the output after executing the above. You will notice that the function `tagstr2tree` automatically puts a S chunk around the entire sentence (it assumes that the input is a full sentence). We are interested only in noun phrase (NP) chunks here, so we can ignore the S chunk.

We can now use the nltk_lite regular expression chunker to identify some NP chunks in the sentence:

```
cp = chunk.Regexp("NP: {<DT><NN>}")
print cp.parse(input)
```

The above code finds the following two NP chunks in the sentence:

```
  (NP: ('The', 'DT') ('market', 'NN'))
  (NP: ('a', 'DT') ('giant', 'NN'))
```

You can provide multiple regexp patterns for identifying NPs (and also provide comment strings) using this syntax:

```
grammar = r"""
NP:
   {<DT>?<JJ>*<NN>}    # chunk determiners, adjectives and nouns
   {<NNP>+}            # chunk sequences of proper nouns
"""
cp = chunk.Regexp(grammar)
```

Note that the order of the patterns is important. You can debug your regexp patterns by using `cp.parse(input, trace=1)` which provides detailed information on the order of the pattern matching.

Provide a program that chunks the example sentence provided and identifies all five of the noun phrase chunks as shown in the marked up example above. Print out the chunked output for the sentence.

(2) † Provide a regular expression based chunker (using the nltk_lite chunker) to identify noun phrase chunks for the CONLL-2000 chunk dataset which contains Wall Street Journal text that has been chunked by human experts. Your chunker should at least have 91% accuracy.

Since you will have to deal with a large variety of noun phrase chunks you will need to generalize your regexp patterns. A *tag pattern* is a sequence of part-of-speech tags delimited using angle brackets, e.g. <DT><JJ><NN>. Tag patterns are the same as the regular expression patterns we have already seen, except for two differences which make them easier to use for chunking. First, angle brackets group their contents into atomic units, so <NN>+ matches one or more repetitions of the tag NN; and <NN|JJ> matches the NN or JJ. Second, the period wildcard operator is constrained not to cross tag delimiters, so that <N.*> matches any single tag starting with N.

To test the accuracy of your chunker use the built-in nltk_lite function:

```
from nltk_lite import chunk
from nltk_lite.corpora import conll2000, extract

cp = chunk.Regexp("NP: {<DT><NN>}")
print chunk.accuracy(cp, conll2000.chunked(files='test', chunk_types=('NP',)))
```

You can compare the output of your chunker with the gold standard to find out which chunks you are missing. For instance, the following code prints the gold standard and then prints the chunker output:

```
from nltk_lite import chunk
from nltk_lite.corpora import conll2000, extract

gold_tree = conll2000.chunked(files='train', chunk_types=('NP',)).next()
print gold_tree
print cp.parse(gold_tree.flatten())
```

Use the following steps in your development process:

a. Write down a regexp chunker using tag patterns that can identify the following examples of noun phrases:

```
another/DT sharp/JJ dive/NN
trade/NN figures/NNS
any/DT new/JJ policy/NN measures/NNS
earlier/JJR stages/NNS
Panamanian/JJ dictator/NN Manuel/NNP Noriega/NNP
his/PRP$ Mansion/NNP House/NNP speech/NN
3/CD %/NN to/TO 4/CD %/NN
more/JJR than/IN 10/CD %/NN
the/DT fastest/JJS developing/VBG trends/NNS
```

b. Write a tag pattern to match noun phrases containing plural head nouns, e.g. `many/JJ types/NNS`, `two/CD weeks/NNS`, `both/DT new/JJ positions/NNS`. Try to do this by generalizing the tag pattern that handled singular noun phrases.

c. Write tag pattern to cover noun phrases that contain gerunds, e.g. `the/DT receiving/VBG end/NN`, `assistant/NN managing/VBG editor/NN`.

d. Write one or more tag patterns to handle coordinated noun phrases, e.g. `July/NNP and/CC August/NNP`, `all/DT your/PRP$ managers/NNS and/CC supervisors/NNS`, `company/NN courts/NNS and/CC adjudicators/NNS`.

e. Compare your output with the gold standard output on some randomly chosen examples from the training data of CoNLL-2000 dataset. See if there are any NP chunks missing in your output and find tag patterns that will include them. Generalize your tag patterns to avoid having one pattern per example.

(3) † Warning: only attempt this question after you have finished Question 2.

The file `genia3.02-small-pos.txt` contains a small amount of text extracted out of bio-medical journals. In this question, we will test how well the chunker you have developed on the Wall Street Journal can deal with text in a completely different domain.

Note that you will now have to deal with the raw text and convert it into a format suitable for use with your chunker.

As you can imagine, the text in this corpus can be very different. Here is a typical example sentence from our bio-medical corpus:

> These/DT findings/NNS should/MD be/VB useful/JJ for/IN therapeutic/JJ strategies/NNS and/CC the/DT development/NN of/IN immunosuppressants/NNS targeting/VBG the/DT CD28/NN costimulatory/NN pathway/NN ./.

But does dealing with a different domain affect your chunker? Run your chunker on this dataset. Depending on the regexp patterns you created for the WSJ text, you may have to tweak your regexp chunker with some additional rules.

Note that we cannot test accuracy on this domain since we do not have human labeled data. We will compare your output with our own chunker on this domain.

(4) † In nltk_lite you can easily represent trees. For instance:

```
from nltk_lite.parse import bracket_parse
sent = '(S (S (NP Kim) (V arrived)) (conj or) (S (NP Dana) (V left)))'
tree = bracket_parse(sent)
print tree[0]
left_tree = tree[0]
print left_tree[0]
```

The above code will print out two constituents of the tree:

```
(S: (NP: 'Kim') (V: 'arrived'))
(NP: 'Kim')
```

Write a program that prints out *all* the constituents of a tree, one per line, using the nltk_lite tree handling functions shown above. For the above input it should produce:

```
(S:
  (S: (NP: 'Kim') (V: 'arrived'))
  (conj: 'or')
  (S: (NP: 'Dana') (V: 'left')))
(S: (NP: 'Kim') (V: 'arrived'))
(NP: 'Kim')
(V: 'arrived')
(conj: 'or')
(S: (NP: 'Dana') (V: 'left'))
(NP: 'Dana')
(V: 'left')
```

(5) Write down two trees, one for each reading of the phrase *old men and women*.

(6) Run the recursive descent parser demo:

```
from nltk_lite.draw import rdparser
rdparser.demo()
```

(7) Chapter 7 of the NLTK tutorial provides a good overview of the grammar development process that can be used to describe the *syntax* of natural language sentences. The notion of a context-free grammar allows us to describe nested constituents unlike a chunking grammar. Based on the ideas provided in Chapter 7 of the NLTK tutorial and the lecture notes, write a context-free grammar that can recognize the following sentences (taken from the NLTK tutorial, Chapter 7):

```
(27a) Jodie won the 100m freestyle
(27b) 'The Age' reported that Jodie won the 100m freestyle
(27c) Sandy said 'The Age' reported that Jodie won the 100m freestyle
(27d) I think Sandy said 'The Age' reported that Jodie won the 100m freestyle
```

Write down your context-free grammar using the following format:

```
productions = '''
S -> NP VP
VP -> V NP | V NP PP
V -> "saw" | "ate"
NP -> "John" | "Mary" | "Bob" | Det N | Det N PP
Det -> "a" | "an" | "the" | "my"
N -> "dog" | "cat" | "cookie" | "park"
PP -> P NP
P -> "in" | "on" | "by" | "with"
'''
```

You can then use your grammar to parse an input sentence. For example, the following code prints out a parse for the sentence *Mary saw Bob* when analyzed using the above grammar.

```
from nltk_lite import parse
from nltk_lite import tokenize
grammar = parse.cfg.parse_grammar(productions)
rd_parser = parse.RecursiveDescent(grammar)
sent = list(tokenize.whitespace("Mary saw Bob"))
for p in rd_parser.get_parse_list(sent):
    print p
```

Print out the parses for the example sentences above using your context-free grammar.

(8) † A Treebank is a corpus of sentences such that each sentence is provided with it's most plausible syntax tree as determined by a human expert. You are provided with a Treebank for sentences from the Air Travel Information Service (ATIS) domain in the file `atis3.treebank`.
From this Treebank, extract a context-free grammar. Provide the context-free grammar as a text file.

(9) † Trim down the number of rules in your context-free grammar, either based on the frequency of the rule or manual inspection or both (this step is necessary to reduce the time taken by the parser). Use your reduced context-free grammar with the nltk_lite recursive descent parser to parse all the sentences in the file `atis.test`. You may need to add lexical rules (e.g. $NNP \rightarrow Miami$) in order to parse some of these sentences. Submit your Python code and a text file containing a list of parse trees, one for each sentence in `atis.test`.