Homework #4: CMPT-413

Anoop Sarkar
http://www.cs.sfu.ca/~anoop

Only submit answers for questions marked with †.

Important! To solve this homework you must read the following chapters of the NLTK Book, available online at *http://www.nltk.org/book*

- Chapter 7. Extracting Information from Text
- Chapter 8. Analyzing Sentence Structure
- Chapter 9. Building Feature Based Grammars

(1) † The following code tells you how to write a simple regexp based chunker using *tag 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.

```
from nltk 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

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

Provide a regular expression based chunker (using the NLTK 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. To load the CONLL-2000 training data:

```
from nltk.corpus import conll2000
train = conll2000.chunked_sents('train.txt', chunk_types=('NP',))

from nltk import chunk
from nltk.corpus import conll2000

cp = chunk.RegexpParser("NP: {<DT><NN>}")
print chunk.accuracy(cp, conll2000.chunked_sents('test.txt', chunk_types=('NP',)))
```

You must obtain 90% or higher on the test data. You can obtain this accuracy (or higher) by using the following algorithm: collect all part of speech tags that are more likely to occur inside NP chunks than outside NP chunks in the training data and then create a regexp that can match one or more of these part of speech tags in any order.

You can improve your accuracy by comparing 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:

```
gold_tree = conll2000.chunked_sents('train.txt', chunk_types=('NP',))[1]
print gold_tree
print cp.parse(gold_tree.flatten())
```

(2) In NLTK you can easily represent trees. For instance:

```
from nltk.tree import Tree
sent = '(S (S (NP Kim) (V arrived)) (conj or) (S (NP Dana) (V left)))'
tree = Tree.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 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')
```

(3) Run the recursive descent parser demo:

```
from nltk.app import rdparser
rdparser()
```

(4) Chapter 8 of the NLTK book 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 8 of the NLTK book and the lecture notes, write a context-free grammar that can recognize the following sentences:

```
(26a) Jodie won the 100m freestyle
(26b) 'The Age' reported that Jodie won the 100m freestyle
(26c) Sandy said 'The Age' reported that Jodie won the 100m freestyle
(26d) 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
NP -> 'John' | 'Mary' | 'Bob' | Det N | Det N PP
VP -> V NP | V NP PP
```

```
V -> 'saw' | 'ate'
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.

```
import nltk

grammar = nltk.parse_cfg(productions)
tokens = 'John saw Mary in the park'.split()
cp = nltk.ChartParser(grammar)
for (c,tree) in enumerate(cp.nbest_parse(tokens)):
    print tree
print "number of parse trees=", c+1
```

Print out the parses for the example sentences above using the context-free grammar you developed to analyze them. Also experiment with the chart parser to parse the same sentence.

```
import nltk

grammar = nltk.parse_cfg(productions)
cp = nltk.ChartParser(grammar)
sent = 'Mary saw Bob'.split()
for p in cp.nbest_parse(sent):
    print p
```

(5) Using `nltk.ChartParser` write down a simple grammar that will produce five trees, one for each reading of the phrase *natural language processing course*.

(6) Write down a grammar that prints out the number of parse trees for the sentences provided in the testcases. The grammar has to be written in such a way that the number of parse trees must match the number provided in the testcases.

Also provide an implementation of the Catalan number function, and use the Catalan function to predict the number of parses reported for the above sentence. You should insert your grammar and the implementation of the catalan function into the following code fragment (the code is a skeleton and will not work without modification):

```
import nltk

# implement the catalan function and insert the right value for the
# parameter n below so that you can predict the total number of parses
# for an input sentence from sys.stdin
print "catalan number =", catalan(n)

grammar = nltk.parse_cfg("""
# write down your grammar here in the usual X -> A B C format
""")
tokens = 'Kafka ate the cookie in the box on the table'.split()
cp = nltk.ChartParser(grammar, parse.TD_STRATEGY)
for (c,tree) in enumerate(cp.nbest_parse(tokens)):
    print tree
print "number of parse trees=", c+1
```

(7) Provide a verbal description (a paraphrase) of a suitable *meaning* for each of the five possible parse trees produced by your grammar in Question (6) for the sentence:

> Kafka ate the cookie in the box on the table.

(8) † Implement the Earley recognition algorithm. Note that you only have to accept or reject an input string based on the input grammar. You do not have to produce the parse trees for a valid input string.
Use the following grammar:

```
import nltk

gram = '''
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"
'''
g = nltk.parse_cfg(gram)
start = g.start()
```

Produce a trace of execution for an input sentence from sys.stdin using your implementation of the Earley algorithm:
The file `earley_setup.py` contains some helpful tips on how to build the data structures required to implement this algorithm.

(9) Using the NLTK functions for feature structures (see Chapter 9 of the NLTK book), provide a Python program that prints out the results of the following unifications:

1. $\left[\textbf{number: sg}\right] \sqcup \left[\textbf{number: sg}\right]$
2. $\left[\textbf{number: sg}\right] \sqcup \left[\textbf{person: 3}\right]$
3. $\begin{bmatrix}\textbf{agreement:} & \boxed{1}\left[\textbf{number: sg}\right] \\ \textbf{subject:} & \left[\textbf{agreement: } \boxed{1}\right]\end{bmatrix} \sqcup \left[\textbf{subject: } \left[\textbf{agreement: } \left[\textbf{person: 3}\right]\right]\right]$

(10) † Consider the following feature-based context-free grammar:

```
% start S
S -> NP[num=?n] VP[num=?n]
NP[num=?n] -> Det[num=?n] N[num=?n]
NP[num=pl] -> N[num=pl]
VP[tense=?t, num=?n] -> TV[tense=?t, num=?n] NP
Det -> 'the'
N[num=sg] -> 'dog'
N[num=pl] -> 'dogs' | 'children'
TV[tense=pres, num=sg] -> 'likes'
TV[tense=pres, num=pl] -> 'like'
TV[tense=past, num=?n] -> 'liked'
```

The notation `?n` represents a variable that is used to denote reentrant feature structures, e.g., in the rule

```
S -> NP[num=?n] VP[num=?n]
```

the num feature of the NP has to be the same value as the num feature of the VP.
The notation can also be used to pass up a value of a feature, e.g., in the rule

```
VP[tense=?t, num=?n] -> TV[tense=?t, num=?n] NP
```

the tense and num features are passed up from the transitive verb TV to the VP.
Save this grammar to the file feat.fcfg, then we can use the following code to parse sentences using the above grammar.

```
import nltk
from nltk.parse import FeatureEarleyChartParser

def parse_sent(cp, sent):
  trees = cp.nbest_parse(sent.split())
  print sent
  if (len(trees) > 0):
    for tree in trees: print tree
  else:
    print "NO PARSES FOUND"

# load a feature-based context-free grammar
g = nltk.parse_fcfg(open('feat.fcfg').read())
cp = FeatureEarleyChartParser(g)

parse_sent(cp, 'the dog likes children')
parse_sent(cp, 'the dogs like children')
parse_sent(cp, 'the dog like children') # should not get any parses
parse_sent(cp, 'the dogs likes children') # should not get any parses
parse_sent(cp, 'the dog liked children')
parse_sent(cp, 'the dogs liked children')
```

Write down a feature-based CFG and save it to a file called spanish.fcfg in your answer directory. Your parser should read this file and parse a sentence provided in sys.stdin. The grammar must appropriately handle the agreement facts in the Spanish noun phrases shown below; *sg* stands for singular, *pl* stands for plural, *masc* stands for masculine gender, *fem* stands for feminine gender. In Spanish, inanimate objects also carry grammatical gender morphology.

1. un cuadro hermos-o.
   a(sg.masc) picture beautiful(sg.masc).
   'a beautiful picture'
2. un-os cuadro-s hermos-os.
   a(pl.masc) picture(pl) beautiful(pl.masc).
   'beautiful pictures'
3. un-a cortina hermos-a.
   a(sg.fem) curtain beautiful(sg.fem).
   'a beautiful curtain'
4. un-as cortina-s hermos-as.
   a(pl.fem) curtain(pl) beautiful(pl.fem).
   'beautiful curtains'

(11) A subcategorization frame for a verb represents the required arguments for the verb, e.g. in the sentence *Zakalwe ate haggis* the verb *eat* has a subcat frame NP. Similarly in the sentence *Diziet gave Zakalwe a weapon* the subcat frame for *give* is NP NP. Assume that we wanted to compactly represent subcategorization frames for verbs using the CFG:

$$\begin{aligned} VP &\rightarrow Verb \\ VP &\rightarrow VP\ X \end{aligned}$$

The non-terminal X can be associated with the category of the various arguments for each verb using a feature structure. For example, for a verb which takes an NP arguments, X is associated with the feature structure: $\left[\text{cat: NP}\right]$. Write down a feature-based CFG that associates with the VP non-terminal an attribute called *subcat* which can be used to compactly represent all of the following subcategorization frames:

1. no arguments
2. NP
3. NP NP
4. NP PP
5. S
6. NP S

Note that the CFG rules should not be duplicated with different feature structures for the different subcategorization frames.

(12) † The `WordNet` database is accessible online at `http://wordnet.princeton.edu/`. Follow the link to *Use WordNet Online* or go directly to:
`http://wordnet.princeton.edu/perl/webwn`

`WordNet` contains information about word *senses*, for example, the different senses of the word *plant* as a manufacturing plant or a flowering plant. For each sense, `WordNet` also has several class hierarchies based on various relations. Once such relation is that of *hypernyms* also known as *this is a kind of* ... relation. It is analogous to a object-oriented class hierarchy over the meanings of English nouns and verbs and is useful in providing varying types of word class information.

For example, the word *pentagon* has 3 senses. The sense of *pentagon* as five-sided polygon has the following hypernyms. The word *line* has 30 senses as a noun (and a further 6 senses as a verb). The sense of *line* as trace of moving point in geometry has the following hypernyms.

Sense3: pentagon
⇒polygon,polygonal-shape
⇒plane-figure,two-dimensional-figure
⇒figure
⇒shape,form
⇒attribute
⇒abstraction, abstract entity

Sense4: line
⇒shape,form
⇒attribute
⇒abstraction, abstract entity

The wordnet module of NLTK can be used to print this type of information:

```
from nltk.corpus import wordnet as wn
from pprint import pprint

hyp = lambda s:s.hypernyms() # useful to print hypernym tree

line = wn.synset('line.n.3')
print line.name + ':', line.definition
pprint(line.tree(hyp))
```

```
for pentagon_sense in wn.synsets('pentagon', 'n'):
    print pentagon_sense.name + ':', pentagon_sense.definition
    pprint(pentagon_sense.tree(hyp))
```

See the file `wordnet_examples.py` for some more examples on how to access Wordnet from NLTK.

The *lowest common ancestor* for these two senses is the hypernym shape, form. A *hypernym path* goes up the hypernym hierarchy from the first word to a common ancestor and then down to the second word. Note that a hypernym path from a node other than the lowest common ancestor will always be equal to or longer than the hypernym path provided by the lowest common anscestor. For the above example, the hypernym path is *pentagon* ⇒ polygon,polygonal-shape ⇒ plane-figure,two-dimensional-figure ⇒ figure ⇒ shape,form ⇒ *line*.

The following NLTK code prints out the distance to the lowest common ancestor across all senses of the two nouns: *pentagon* and *line*:

```
min_distance = -1
for pentagon_sense in wn.synsets('pentagon', 'n'):
    for line_sense in wn.synsets('line', 'n'):
        dist = pentagon_sense.shortest_path_distance(line_sense)
        if min_distance < 0 or dist < min_distance:
            min_distance = dist
print "min distance (pentagon, line) =", min_distance
```

For words other than the ones used in the example above, the code would print a value of `-1` for the minimum distance if the two words have no common ancestors at all.

Provide Python code that extends the NLTK implementation above to print out the synset value of the lowest common ancestor across all senses for pairs of input words (provided in the testcases). You must assume that you only access the noun database in WordNet for the input words.