

Homework #5: CMPT-413

Reading: NLTK Tutorial Chp 8 and 11;
http://nltk.org/doc/en/\parse_featgram}.html;
Distributed on Mar 17; due on Mar 31
Anoop Sarkar – anoop@cs.sfu.ca

Only submit answers for questions marked with †. For the questions marked with a ††; choose one of them and submit its answer.

- (1) † Write down a grammar that derives 5 parse trees for the sentence:

Kafka ate the cookie in the box on the table.

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):

```
from nltk import cfg, parse

# 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 the sentence "Kafka ate the cookie in the box on the table."
print "catalan number =", catalan(n)

grammar = cfg.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 = parse.ChartParser(grammar, parse.TD_STRATEGY)
for (c,tree) in enumerate(cp.nbest_parse(tokens)):
    print tree
print "number of parse trees=", c+1
```

- (2) Provide a verbal description (a paraphrase) of a suitable *meaning* for each of the five possible parse trees produced by your grammar in Question (1) for the sentence:

Kafka ate the cookie in the box on the table.

- (3) Pingala, a linguist who lived circa 5th century B.C., wrote a treatise on Sanskrit prosody called the Chandas Shastra. Virahanka extended this work around the 6th century A.D., providing the number of ways of combining short and long syllables to create a meter of length n . Meter is the recurrence of a pattern of short and long syllables (or stressed and unstressed syllables when applied to English). He found, for example, that there are five ways to construct a meter of length 4: $V_4 = \{LL, SSL, SLS, LSS, SSSS\}$. In general, we can split V_n into two subsets, those starting with L: $\{LL, LSS\}$, and those starting with S: $\{SSL, SLS, SSSS\}$. This provides a recursive definition for computing the number of meters of length n . Virahanka wrote down the following recursive definition for a 6th century Python interpreter:

```
def virahanka(n, lookup = {0:[""], 1:["S"]} ):
    if (not lookup.has_key(n)):
        s = ["S" + prosody for prosody in virahanka(n-1)]
        l = ["L" + prosody for prosody in virahanka(n-2)]
        lookup.setdefault(n, s + l) # insert a new computed value as default
    return lookup[n]
```

Note that Virahanka has used top-down dynamic programming to ensure that if a user calls `virahanka(4)` then the two separate invocations of `virahanka(2)` will only be computed once: the first time `virahanka(2)` is computed the value will be stored in the lookup table and used for future invocations of `virahanka(2)`.

Write a *non-recursive* bottom-up dynamic programming implementation of the `virahanka` function. Provide the following output:

```
virahanka(4) = ['SSSS', 'SSL', 'SLS', 'LSS', 'LL']
```

- (4) † 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:

```
from nltk import cfg

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 = cfg.parse_cfg(gram)
start = g.start()
```

Produce a trace of execution for the input sentence using your implementation of the Earley algorithm:

the cat in the park ate my cookie

The file `earley_setup.py` contains some helpful tips on how to build the data structures required to implement this algorithm. Your program should produce a trace that is identical to the output given in the file `earley-trace.txt`.

- (5) † Using the NLTK functions for feature structures (see Section 11.3 of the NLTK tutorial), provide a Python program that prints out the results of the following unifications:

1. $\left[\text{number: sg} \right] \sqcup \left[\text{number: sg} \right]$
2. $\left[\text{number: sg} \right] \sqcup \left[\text{person: 3} \right]$
3. $\left[\begin{array}{l} \text{agreement: } \left[\text{number: sg} \right] \\ \text{subject: } \left[\text{agreement: } \left[\text{number: sg} \right] \right] \end{array} \right] \sqcup \left[\begin{array}{l} \text{subject: } \left[\text{agreement: } \left[\text{person: 3} \right] \right] \end{array} \right]$

- (6) † 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.

```
from nltk import cfg
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 = cfg.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 that will 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'

Provide the feature-based CFG as a text file, and the Python code that reads in the file and parses the above noun phrases and prints out the trees. Include at least two ungrammatical noun phrases that violate the agreement facts and as a result cannot be parsed.

- (7) 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:

$$VP \rightarrow Verb$$

$$VP \rightarrow VP X$$

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: [cat: NP]. 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.

- (8) †† 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	Sense4: line
⇒polygon,polygonal-shape	⇒shape,form
⇒plane-figure,two-dimensional-figure	⇒attribute
⇒figure	⇒abstraction, abstract entity
⇒shape,form	
⇒attribute	
⇒abstraction, abstract entity	

The wordnet module of NLTK can be used to print out the same information:

```
from nltk.wordnet import *
from pprint import pprint

pentagon = N['pentagon']
print "sense 2 of pentagon"
pprint(pentagon[2].tree(HYPERNYM))
print "sense 3 of line"
line = N['line']
pprint(line[3].tree(HYPERNYM))
```

The *lowest common ancestor* for these two senses is the hypernym *shape, form*. A *hyponym 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 ancestor. 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 words: *pentagon* and *line*:

```
min_distance = -1
for pentagon_sense in pentagon:
    for line_sense in line:
        dist = pentagon_sense.shortest_path_distance(line_sense)
        if min_distance < 0 or dist < min_distance:
            min_distance = dist
print "min distance =", 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 of any two input words. You should provide test cases for the following pairs of words: (a) *pentagon*, *line* (b) *dog*, *cat* (c) *English*, *Tagalog*.

(9) †† **Prepositional phrase attachment ambiguity resolution**

Consider the sentence *Calvin saw the man with the telescope* which has two meanings, one in which Calvin sees a man carrying a telescope and another in which Calvin using a telescope sees the man. This ambiguity between the *noun-attachment* versus the *verb-attachment* is called prepositional phrase attachment ambiguity, or *PP-attachment ambiguity*.

In order to decide which of the two derivations is more likely, a machine needs to know a lot about the world and the context in which the sentence is spoken. Either meaning is plausible for the sentence *Calvin saw a man with the telescope* given the right context. However, in many cases, some of the meanings are more plausible than others even without a particular context. Consider, *Calvin bought a car with anti-lock brakes* which has a more plausible noun-attach reading, while *Calvin bought the car with a low-interest loan* which has a more plausible verb-attach meaning.

We can write a program that can *learn* which attachment is more plausible. In order to keep things simple, we will only use the verb, the final word (as a *head* word) of the noun phrase complement of the verb, the preposition and final word of the noun phrase complement of the preposition. For the sentence *Calvin saw a man with the telescope* we will use the words: *saw, man, with, telescope* in order to predict which attachment is more plausible. We will also consider only the disambiguation between noun vs. verb attachment for sentences or phrases with a single PP, we do not consider the more complex case with more than one PP.

Here is some example NLTK code to read the training dataset for PP-attachment:

```
from nltk.corpus.reader.ppattach import PPAttachmentCorpusReader
from itertools import islice
data = PPAttachmentCorpusReader('/usr/share/nltk/data/corpora/ppattach',
                                ['training', 'devset', 'test'], '')
for item in islice(data.tuples(['training']), 5): print item
```

which produces:

```
('0', 'join', 'board', 'as', 'director', 'V')
('1', 'is', 'chairman', 'of', 'N.V.', 'N')
('2', 'named', 'director', 'of', 'conglomerate', 'N')
('3', 'caused', 'percentage', 'of', 'deaths', 'N')
('5', 'using', 'crocitolite', 'in', 'filters', 'V')
```

Notice that in each case, we have human annotation of which attachment is more plausible. It is convenient to name each component of the training data tuples: $(\#, v, n1, p, n2, attach)$.

- Learn a model from the training data to predict verb attachment, $\Pr(V | p)$. Note that $\Pr(N | p) = 1 - \Pr(V | p)$. This model is a simple baseline model useful for comparison with the more sophisticated methods below.
- Learn a model from the training data to predict verb attachment, $\Pr(V | v, n1, p, n2)$. Note that $\Pr(N | v, n1, p, n2) = 1 - \Pr(V | v, n1, p, n2)$. Make sure you smooth this probability to deal with unseen tuples. You can use the devset to optimize parameters used for smoothing.
- Write an evaluation script that reads the devset and the test set and prints out the accuracy of your model. Accuracy gives us a measure of how well the model can predict the *attach* value. When developing your model, and improving it, only check accuracy on the devset.
- One of the key problems in predicting attachment for unseen tuples is that all four words in the tuple rarely co-occur. Use Wordnet based similarity (see Question 8 for background on Wordnet) to allow your model to generalize to new tuples. For example, the path similarity between two words is given by the following code:

```
dog = N['dog'][0]
cat = N['cat'][0]
print "dog, cat:", dog.path_similarity(cat)
print "dog, line:", dog.path_similarity(line[3])
```

The path similarity value for semantically closely linked words like *dog* (sense 1) and *cat* (sense 1) is 0.2 while the value for more distant words like *dog* (sense 1) and *line* (sense 4) is 0.0769. This notion of similarity can be used to create a more sophisticated model of PP-attachment. Different methods that compute similarity using Wordnet hypernym paths are discussed in:

<http://marimba.d.umn.edu/similarity/measures.html>

Most of these methods are implemented in NLTK (`wordnet/similarity.py`).

- Provide accuracy of your final model on the test set.