Homework #5: CMPT-413

Due on Apr 5, 2011

Anoop Sarkar – http://www.cs.sfu.ca/~anoop/teaching/cmpt413

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

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

- Chapter 8. Analyzing Sentence Structure
- Chapter 9. Building Feature Based Grammars

(2) 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):

```
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 the sentence "Kafka ate the cookie in the box on the table."
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
```

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

Kafka ate the cookie in the box on the table.

(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:

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

(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.

```
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 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:

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

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.

(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
⇒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 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) † **Competitive Grammar Writing**

This question involves writing or creating weighted context-free grammar files in order to parse English sentences and utterances. The vocabulary is fixed but the task is not trivial: you can choose to either write a weighted grammar by hand or you can use the various methods we have learned about in this course to automatically infer a weighted grammar from a suitably chosen set of data. One important thing to keep in mind is that you can solve this question without writing any code, but you probably will need to write some code to get a high ranking.

You will need to think about the following aspects of grammar development:

Linguistic Analysis You will need to consider different types of *linguistic phenomena* like *wh*-questions, agreement, adjunct clauses, embedded sentences, etc. The NLTK book contains a guide to some basic grammar development, but covering a large number of phenomena, even obscure ones like clefts, will help improve your rank in this question (check the detailed rules below to see why). Your previous homework contained some initial steps towards this kind of grammar development. The homework directory contains some sample grammars (explained in detail below) with default part of speech categories and weights. You can change or modify the provided grammars, especially the weights, as long as you do not change the vocabulary: *increase or decrease of the vocabulary is not allowed*. The grammar must be in **extended Chomsky Normal Form** (eCNF): rules are always of the form $A \rightarrow B\ C$, $A \rightarrow B$, or $A \rightarrow a$ where A, B, C are non-terminals and a is a terminal symbol (word in the vocabulary).

Parameter Tuning You can change the weights associated with rules. The weights can be anything you want, and should reflect the relative importance of the rules. This is especially true since in this question we will also be generating sentences from the grammars you write. So giving large weights to recursive rules can lead to the generation of very long sentences, or even a run-away generative process. You should

also be aware that you may need to change the weights associated with the rules in the default grammar files provided in the homework directory.

Quantitative Evaluation This question will involve a very rigorous quantitative evaluation of your submitted grammar. Keep the details of the evaluation in mind as you develop your grammars. The goal is to get the highest evaluation score and to make it harder for others to reach your score. The evaluation is complicated because it tries to avoid some kinds of cheating to improve the ranking, so there is a whole section below devoted to explaining the evaluation procedure.

The Data All the data you need for grammar development is provided to you in the following files.

`S1.gr` The default main grammar file which contains a weighted context-free grammar in extended Chomsky Normal Form (eCNF).

$$\begin{array}{rrcl} 1 & \mathit{S1} & \rightarrow & \mathit{NP\ VP} \\ 1 & \mathit{S1} & \rightarrow & \mathit{NP\ \_VP} \\ 1 & \mathit{\_VP} & \rightarrow & \mathit{VP\ Punc} \\ 20 & \mathit{NP} & \rightarrow & \mathit{Det\ Nbar} \\ 1 & \mathit{NP} & \rightarrow & \mathit{Proper} \end{array}$$

Notice the non-terminal *_VP* which is used to keep the grammar in eCNF. The probability of a particular rule is obtained by normalizing the weights for each left-hand side non-terminal in the grammar. For example, for rule $NP \rightarrow Det\ Nbar$ the probability is $20/(20+1)$.

`S2.gr` The default backoff grammar file. This grammar file ensures that any input constructed using the allowed vocabulary will be accepted by the parser (although it may accept it with low probability depending on the weights). The files `S1.gr` and `S2.gr` are connected via the following rules in `S1.gr`:

$$\begin{array}{rrcl} 99 & \mathit{TOP} & \rightarrow & \mathit{S1} \\ 1 & \mathit{TOP} & \rightarrow & \mathit{S2} \\ 1 & \mathit{S2} & \rightarrow & \mathit{Misc} \end{array}$$

Notice how the grammar in `S1.gr` is highly weighted (99 times out of 100) so that `S2.gr` is used as a backoff grammar. You may want to consider optimizing this weight to improve your evaluation score. *Misc* expands to vocabulary items defined in `Vocab.gr`.

One way to improve the weights for the rules in `S2.gr` is to use Hidden Markov Model learning, but there are many other ways to obtain a good backoff grammar.

`Vocab.gr` The vocabulary file containing rules of the type $A \rightarrow a$ where A is a part of speech tag and a is a word that will appear in the input sentences. Many words are assigned to the default part of speech *Misc*, so you will probably want to re-assign these words to more useful part of speech tags in order to aid your grammar development.

`allowed_words.txt` This file contains all the words allowed in the evaluation. You should make sure that your grammar does not generate any words not in this file. It does not specify the part of speech for each word, so you can choose to model the ambiguity of words in terms of part of speech in your `Vocab.gr` file.

`cgw-devset.txt` This file contains example sentences that you can use as a starting point for your grammar development. Only the first two sentences of this file can be parsed using the default `S1.gr` grammar. The rest are parsed with the backoff `S2.gr` grammar. You can augment this data with any other data you can find on the web (but make sure you do not expand the vocabulary).

`unseen.tags` Used to deal with unknown words. You should not have to use this file during parsing, but the parser provided to you can optionally use this file in order to deal with unknown words in the input. Since the vocabulary is fixed for this question you do not need to use this file, but it is provided in case you want to try the parser on your own data.

The Parser and Generator You are given a parser that takes sentences as input and produces parse trees and also a generator which generates a random sample of sentences from the weighted grammar. Parsing and generating will be useful steps in your grammar development strategy. You can learn the various options for running the parser and generator using the following command. The parser has several options to speed up parsing, such as beam size and pruning. Most likely you will not need to use those options (unless your grammars are huge).

```
$ python2.6 pcfg_parse_gen.py -h
```

Parsing input: The parser provided to you reads in the grammar files and a set of input sentences. It prints out the single most probable parse tree for each sentence (using the weights assigned to each rule in the input context-free grammar). The parser also reports the negative cross-entropy score for the whole set of sentences. Assume the parser gets a text of n sentences to parse: $s_1, s_2, \ldots, s_n$ and we write $|s_i|$ to denote the length of each sentence s_i. The probability assigned to each sentence by the parser is $P(s_1), P(s_2), \ldots, P(s_n)$. The negative cross entropy is the average log probability score (bits per word) and is defined as follows:

$$\text{score}(s_1, \ldots, s_n) = \frac{\log P(s_1) + \log P(s_2) + \ldots + \log P(s_n)}{|s_1| + |s_2| + \ldots + |s_n|}$$

We keep the value as negative cross entropy so that higher scores are better. For example, running the parser with the default grammar files on the development set of sentences `cgw-devset.txt` gives the following output (ignoring the output parse trees for each input sentence):

```
$ python2.6 pcfg_parse_gen.py -i -g "*.gr" < cgw-devset.txt
#loading grammar files: S1.gr, S2.gr, Vocab.gr
#reading grammar file: S1.gr
#reading grammar file: S2.gr
#reading grammar file: Vocab.gr
#skipping comment line in input: # this is the devset for hw5 q9
#-cross entropy (bits/word): -10.0557
```

Generating output: In order to aid your grammar development you can also generate sentences from the weighted grammar to test if your grammar is producing grammatical sentences with high probability. The following command samples 20 sentences from the `S1.gr,Vocab.gr` grammar files. Always sample only from these two files and ignore the backoff grammar (as you will see from the evaluation procedure).

```
$ python2.6 pcfg_parse_gen.py -o 20 -g S1.gr,Vocab.gr
#loading grammar files: S1.gr, Vocab.gr
#reading grammar file: S1.gr
#reading grammar file: Vocab.gr

every pound covers this swallow          the castle is the sovereign
no quest covers a weight                 the king has this sun
Uther Pendragon rides any quest          that swallow has a king
the chalice carries no corner .          another story rides no story
any castle rides no weight               this defeater carries that sovereign
Sir Lancelot carries the land .          each quest on no winter carries the sovereign .
a castle is each land                    another king has no coconut through another husk .
every quest has any fruit .              a king rides another winter
no king carries the weight               that castle carries no castle
that corner has every coconut            every horse covers the husk .
```

During the time that the homework is assigned to you, you are allowed to share samples of your output with others in the class (the more students share samples the better it is for everyone). To avoid flooding the mailing list with samples from your grammars, please use an online pastebin, e.g. the site

pastebin.mozilla.org, to create a pastebin with your samples and mail the link to the mailing list. You can edit the pastebin with additional samples as you develop better grammars but please do not email the mailing list each time you update the pastebin. For example, the above set of 20 sentences sampled from the default grammar is available at *http://pastebin.mozilla.org/1162241*.

You should modify your grammar to improve the cross entropy score assigned to samples from the grammars written by others. **Important:** You cannot share your grammars, only the samples from your grammars. Sharing grammars will be treated as plagiarism.

The Evaluation For this question you should submit your grammar files `"*.gr"` which must contain at least one rule with the start symbol `TOP` and your submission must have at least one file called `S1.gr`. *Please do not put the grammar files into subdirectories. We want to run an automatic script on your grammar files, so please keep them at the top directory level in your submission.* Based on the `"*.gr"` you will submit as your solution to this question, we will use the following steps in order to provide a ranking to each submission.

Step 1: Sample sentences We will sample 20 sentences from the `S1.gr` and `Vocab.gr` file from each submission using the following command (this example uses the default grammar files):

```
$ python2.6 pcfg_parse_gen.py -o 20 -g S1.gr,Vocab.gr
#loading grammar files: S1.gr, Vocab.gr
#reading grammar file: S1.gr
#reading grammar file: Vocab.gr
```

If your submitted grammar files cause an error in the sentence generator then you will receive zero marks for this question. We might have to delete some lines from your output for misbehaving incomplete submissions.

In order to score a high rank, it is important to note that the `S2.gr` file is **not** used to sample sentences.

Step 2: Parse open test set We will concatenate the 20 sentence samples obtained from all the submissions into a single file called `cgw-testset.txt`. We will parse this file with the parser using your grammar files (including `S2.gr`). The score obtained on `cgw-testset.txt` will be part of the rank given to your grammar submission.

Step 3: Parse with test set During testing your submission we will parse a file called `cgw-holygrail.txt` using your `S1.gr` and `Vocab.gr` grammar files. The file `cgw-holygrail.txt` contains previously unseen sentences from the same domain as the file `cgw-devset.txt` (the sentences use the words given in `allowed_words.txt`). The score obtained on `cgw-holygrail.txt` will be part of the rank given to your grammar submission, but you will not have access to this file during your grammar development process. The reason to include this test is to penalize submissions that use the following trick to get a high ranking: put a very small weight on `S1.gr` and put a high weight on `S2.gr` and then your grammar will get a score for the sentences in `cgw-testset.txt` mainly using your `S2.gr`. With this weighting the trick is to use ungrammatical sentences in `S1.gr` which is used for sampling sentences for others. This trick will not work since we use `S1.gr` to parse the unseen text file `cgw-holygrail.txt` and the ranking depends on this score as well.

Step 4: Rank each submission Each submission will be ranked using the following formula:

$$\text{rank} = 0.5 \times \text{score}(\texttt{cgw-testset.txt}) + 0.5 \times \text{score}(\texttt{cgw-holygrail.txt})$$

We will look at your submission to assign a grade for this question – especially the effort made in `S1.gr` and in `S2.gr`. You can submit a text file called `cgw-readme.txt` that includes your strategy for obtaining a high rank and the process you used to write or automatically produce `S1.gr` and `S2.gr`.

Further Reading This question is an adaptation of the idea presented in the following paper:

Jason Eisner and Noah A. Smith. Competitive Grammar Writing. In *Proceedings of the ACL Workshop on Issues in Teaching Computational Linguistics*, pages 97-105, Columbus, OH, June 2008. *http://aclweb.org/anthology/W/W08/W08-0212.pdf*

You can read this paper for a discussion on how to get the best score on this question, but be aware that some of the details have been changed, so follow the instructions provided here.

(10) † Provide a Python file `exec.py` that will execute each of your programs. Run your programs with different test cases (especially the ones provided in the homework as examples). Please provide verbose descriptions to explain how your programs work (when necessary).