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 6. Learning to Classify Text
- Chapter 7. Extracting Information from Text
- Chapter 8. Analyzing Sentence Structure

(1) **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 import ppattach
print ppattach.attachments('training')[0]
print ppattach.attachments('devset')[0]
print ppattach.attachments('test')[0]
```

which produces:

```
PPAttachment(sent='0', verb='join', noun1='board', prep='as',
    noun2='director', attachment='V')
PPAttachment(sent='40000', verb='set', noun1='stage', prep='for',
    noun2='increase', attachment='N')
PPAttachment(sent='48000', verb='prepare', noun1='dinner', prep='for',
    noun2='family', attachment='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: $(\#, v, n1, p, n2, attach)$.

a. Learn a simple baseline model from the training data to predict verb attachment, $\Pr(V \mid p)$. Note that $\Pr(N \mid p) = 1 - \Pr(V \mid p)$. This model is a simple baseline model useful for comparison with the more sophisticated methods below. Provide the accuracy on the dev and test set. While developing your code only evaluate on the dev set to avoid tuning your code to the test set.

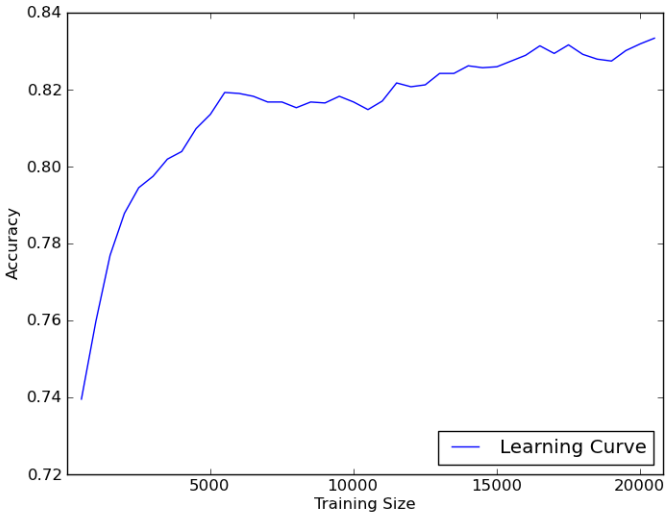

Figure 1: Learning curve for the PP attachment task.

b. † Learn a model from the training data to predict verb attachment, $\Pr(V \mid v, n1, p, n2)$. Note that $\Pr(N \mid v, n1, p, n2) = 1 - \Pr(V \mid v, n1, p, n2)$. If you use a generative model of $P(V, v, n1, p, n2)$ make sure you smooth this probability to deal with unseen tuples using the dev set. A generative model for this problem is discussed in the lecture notes and you can use `nltk.NaiveBayesClassifier` to implement it (see Chapter 6 of the NLTK book).

   Provide the accuracy on the dev and test set. While developing your code only evaluate on the dev set to avoid tuning your code to the test set.

c. Provide the PNG file of the learning curve for your classifier (plot accuracy on the dev set on the y-axis and size of training data on the x-axis in increments of a 1000 labeled examples). The learning curve using the nltk maxent classifer with the IIS algorithm for optimization is given in Fig. 1. Your learning curve could look very different if you use a different learning method.

(2) 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())
```

(3) 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
```

(4) † **Competitive Grammar Writing**

This homework involves writing or creating weighted context-free grammars in order to parse English sentences and utterances. The vocabulary is fixed. 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 will need to write some code to learn a weighted grammar that will obtain a high ranking. This homework has the following goals:

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). The homework directory contains TreeBank data which you can use to create your own grammar. 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. There is a whole section below devoted to explaining the evaluation procedure.

The Data Initial versions of the grammar files you will submit are provided to you:

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

1	*S1*	→	*NP VP*
1	*S1*	→	*NP _VP*
1	*_VP*	→	*VP Punc*
20	*NP*	→	*Det Nbar*
1	*NP*	→	*Proper*

The non-terminal *_VP* 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 grammars in files `S1.gr` and `S2.gr` are connected via the following rules in `S1.gr`:

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

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 generates sentences using exactly the words 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).

```
python 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):

```
python pcfg_parse_gen.py -i -g "*.gr" < example_sentences.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 the parse trees ...

#-cross entropy (bits/word): -10.0502
```

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).

```
python 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 .
```

The Evaluation For this question you should submit your grammar files `S1.gr, S2.gr, Vocab.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.* The evaluation will be done as follows:

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):

```
python 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 will rely on all the students in this class to edit the final set to remove any ungrammatical sentences that are obviously put there to unfairly manipulate the evaluation score.

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 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: Score each submission Each submission will be scored using the following formula:

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

Your Submission You will need to commit the following files into your hw4/answer directory using svn.

S1.gr Your main grammar file in extended Chomsky Normal Form (eCNF).

S2.gr Your backoff grammar file.

Vocab.gr The vocabulary file. Make sure you have not deleted any words that appear in the allowed_words.txt file or made them unreachable by changing the original rules in Vocab.gr.

cgw-readme.txt A text file that describes *in detail* your approach to solving this homework. This file will form an important part of your grade, so make sure you convince the reader that you did something worthwhile.

You should immediately copy the .gr and cgw-readme.txt files above to a hw4/answer directory in your svn repository. As you modify your grammars, you should commit new versions often. We will evaluate the last commit (before the deadline expires) but we will also take into consideration the frequency and how early you commit your grammars to svn. As we pull your grammars we will make public addition sentences sampled from the submitted grammars. 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). 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.

We will look at your submission to assign a grade for this question – especially the documentation you have provided in cgw-readme.txt, the effort made in creating S1.gr and in S2.gr and the evaluation score for your submission (as computed in Step 4 above).

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 to get additional ideas on how to do better on this task, but be aware that some of the details have been changed, so follow the instructions provided here.