CS 517 - Natural Language Processing

Programming Assignment #4

Due date: 05/21/14 16:00 hrs

Overview

[Taken from Jason Eisner and Noah Smith's paper. I strongly encourage you to read this paper along with this instruction!]

Grammars are of interest to both linguists and engineers. They are a formal way of writing down how a language works.

- For linguists, the goal is to write down the true grammar of the language -- the one that is subconsciously used by the language's speakers.
- For engineers, the goal is to apply the grammar to some task. This requires algorithms that make use of grammars: for example, **generation** and **parsing** of sentences.

In this assignment, you will work in a team to design a grammar for as much of English as possible. Nowadays, grammars are constructed somewhat automatically. But by building one the old-fashioned manual way, you'll have to grapple directly with linguistic phenomena, grammars, and probabilities.

You do not have to write any code for this exercise!

System

Your system will consist of *three* sub-grammars:

- S1: A probabilistic context-free grammar, that is supposed to generate **all** and **only** English sentences
- S2: A probabilistic context-free grammar, that generates **all word strings**. This will be used as a **default fallback**, if a sentence is not parsable using S1.
- Vocab: List of **all possible and allowed** lexicons with parts of speech.

If you could design S1 perfectly, then you wouldn't need S2. But English is amazingly complicated, and you only have a few hours. So S2 will serve as your backoff model. It will be able to handle any English sentences that S1 can't.

These grammar files are written with file extension as `.gr`. All lines beginning with the # symbol are considered comments. Empty lines are ignored and all other lines should define a rule of the form:

```
weight <tab>  parent <tab> child1 <space> child2 ...
```

Let's take an example (given in the provided grammar):

```
99    START      S1
1     START      S2
1     S1         NP         VP    .
20    NP         Det        Nbar
1     NP         Proper
…
```

So, probability of `START->S1` is `99/100`; same for `START->S2` is `1/100`. Similarly, probability of `NP->Det Nbar` is `1/21` and `NP->Proper` is `20/21`. It is hard to make sure sum of all probabilities for rules from a given non-terminal as 1.0 and hence this mechanism has been chosen. Given code is smart enough to compute probabilities from weights.

Also, note that, you are allowed write rules having more than 2 children - something like:

```
55    VP    VerbT      Det   Nbar
```

Given tool is smart enough to break these into binary rules and generate parse tree of given sentences.

Vocab

In the file Vocab.gr, we are giving you the set of terminal symbols (words), embedded in rules of the form Tag -> word. Note that the vocabulary is closed. There will be no unknown words in the test sentences, and you are not allowed to add any words (terminal symbols) to the grammar.

S1

We are giving you a simple little S1 to start with. It generates a subset of real English. As noted, we've also given you a set of Tag -> word rules, but you might find that the tags aren't useful when trying to extend S1 into a bigger English grammar. So you are free to create new tags for word classes that you find convenient or use the words directly in the rules, if you find it advantageous. We tried to keep the vocabulary relatively small to make it easier for you to do this.

You will almost certainly want to change the tags in rules of the form `Misc` -> word. But be careful: you don't want to change `Misc -> goes` to `VerbT -> goes`, since `goes` doesn't behave like other `VerbT`'s. In particular, you want your `S1` to generate `Guinevere has the`

`chalice .` but not `Guinevere goes the chalice .`, which is ungrammatical. This is why you may want to invent some new tags.

S2

The goal of S2 is to enforce the intuition that every string of words should have some (possibly miniscule) probability. You can view it as a type of smoothing of the probability model. There are a number of ways to enforce the requirement that no string have zero probability under S2. Just note that your score will become infinitely bad if you ever give zero probability to a sentence.

Our method of enforcing this requirement is to use a grammar that is effectively a bigram (or finite-state) model. Suppose we only have two tag types, A and B. The set of rules we would allow in S2 would be:

```
S2 -> _A
S2 -> _B
S2 ->
_A -> A
_A -> A _A
_A -> A _B
_B -> B
_B -> B _A
_B -> B _B
```

This grammar can generate any sequence of As and Bs, and there is no ambiguity in parsing with it: there is always a single, right-branching parse.

Placing your bets

Two rules your grammar must include are `START -> S1` and `START -> S2`. The relative weight of these determines how likely it is that S1 (with start symbol S1) or S2 (with start symbol S2) would be selected in generating a sentence, and how costly it is to choose one or the other when parsing a sentence. Choosing the relative weight of these two rules is a gamble. If you are over-confident in your "real" English grammar (S1), and you weight it too highly, then you risk assigning very low probability to sentences which S1 cannot generate (since the parser will have to resort to your S2 to get a parse, which gives every sentence a low score).

On the other hand, if you weight S2 too highly, then you will probably do a poor job of predicting the test set sentences, since S2 will not make any sentences very likely (it accepts everything, so probability mass is spread very thin across the space of word strings).

Steps

- Use tools (described below) to generate parse trees

- See if parse trees are valid. For given grammar, it will be really random.
    - Fix those random branching
        - reduce weight for corresponding rule OR
        - introduce better rule
            - Note that you can directly refer lexicons in S1, e.g.
              `S1->either SENT or SENT`
              Here `either` and `or` are already defined as lexicon in Vocab.gr, but `SENT` is not. Hence tool will look for a rule that defined `SENT->`*`something`*.
            - Feel free to define `NON_TERMINALS` as many as you want.
            - Note that, even punctuation marks like (`.,;` etc) can be part of rules.
    - Validate your change by re-generating the tree
- Iterate
- Once you got some reduction in cross-entropy for given sentences, try generating some random sentences (`randsent`, see below) from your grammer to see if that make *some sense*. Note that, it won't be perfect. But more it generates better sentence, it will be better in evaluating unknown sentences.

Tools

We've provided you with three basic tools for developing your PCFG. All these tools are pre-compiled linux binaries and perl script. So, your development environment should be a linux machine with perl installed.

PCFG Parser

This program takes in a sentence file and a sequence of grammar files and parses each of the sentences with the grammar. It will print cross-entropy computed over all of these sentences. Because your grade will be a function of your perplexity (which is 2^cross-entropy) on the dev and test data sets, this is going to be the key tool for evaluating your grammar. To invoke the PCFGParser call:

```
cat Vocab.gr S1.gr S2.gr | ./parse -i examples.sen -C
```

to print the parse trees treebank format use the -t option:

```
cat Vocab.gr S1.gr S2.gr | ./parse -i examples.sen | ./prettyprint
```

to print only score, NOT the parse tree:

```
cat Vocab.gr S1.gr S2.gr | ./parse -i examples.sen -nC
```

PCFG Generator

This program takes a sequence of grammar files and samples sentences from the distribution of sentences described by your PCFG. This can be useful for finding glaring errors in your grammar, or undesirable biases. To generate random set of 20 sentences, you can just call:

```
cat Vocab.gr S1.gr S2.gr | ./randsent -n 20
```

and to print the parse trees along with the actual sentences, use the `-t` option (first):

```
cat Vocab.gr S1.gr S2.gr | ./randsent -n 20 -t
```

and to `prettyprint` the parse trees use:

```
cat Vocab.gr S1.gr S2.gr | ./randsent -n 20 -t | ./prettyprint
```

Validate Grammar

Before submission, make sure that you haven't created any non-terminals which never generate a terminal. While you won't be explicitly penalized for such mistakes, they will only hurt your perplexity because they will hold out probability for symbols which never actually occur in the dev or test set.

Evaluation

To evaluate the precision of your S1 grammar, we will first generate a test set of sentences from it using randsent. We will then make a grammaticality judgement about each sentence, and your score will be the fraction of sentences judged to be grammatical. To judge the grammaticality of the sentences, we will be using the best tool available: human speakers (yourselves!). We'll elicit grammaticality judgments from each of you on a set of sentences (which might have been generated by your grammar or someone else's -- so be honest!).

The second score evaluates your full grammar (S1+S2) and is much more important. For this, we will measure perplexity to evaluate how well your PCFG does on an unseen set of test sentences. A **low perplexity is good**; it means that your model is unsurprised (not very perplexed) by the test sentences. You will be able to parse all of the sentences (because S2 will accept anything, albeit with low probability). The score will be the cross-entropy of your model (as perplexity = 2^cross-entropy) with respect to both eval and test set.

If the test set of sentences is {s1, s2, s3, ...}, then your model's cross-entropy is defined as

(-log(p(s1)) - log(p(s2)) - log(p(s3)), ...) / (|s1| + |s2| + |s3| + ...)
where |si| is the number of words in the ith sentence. Note that
-log(1)=0, -log(1/2)=1, -log(1/4)=2, -log(1/8)=3, ... -log(0)=infinity
So a high cross-entropy corresponds to a low probability and vice-versa. You will be severely penalized for probabilities close to zero.

Note, p(s) denotes the probability that a randomly generated sentence is exactly s. Often your grammar will have many different ways to generate s, corresponding to many different trees. In that case, p(s) is the total probability of all those trees. However, for convenience, we approximate this with the probability of the single most likely tree, because that is what our parser happens to find. So we are really only measuring approximate cross-entropy.

Hint: Remember the part about weighting the START -> S1 and START -> S2 rules? Just updating those weights from their defaults helps cross-entropy to change a lot. Given grammar evaluates to cross-entropy of 9.54751. Spending couple of hours, I could come up with a cross-entropy of 4.47767. Think about different ways we speak English - asking a question; answering it; making a passive statement; etc. and try to formulate a grammar for each of them. Then tune their weights depending on how frequently they are used in common. Creating different kinds of non-terminals also helps.

Submission

To submit, just upload your S1.gr file to blackboard.

Questions for Discussion

- What were some of the linguistic phenomena you handled, and how?
- If s is a sentence, and p1(s) and p2(s) denote the respective probabilities that S1 and S2 generate s, then what is the probability that the whole grammar generates s?
- How could you improve the probabilities assigned by S2, without losing the guarantee that S2 can generate any string of words?
- What would happen if you didn't have S2?
- Is there any other way to combine S1 and S2 besides the one we recommended?
- Suppose you strategically decided to make your grammar devote most of its probability to a really unusual sentence. Who would that hurt more - your team or the other teams?
- Suppose you were leading a 6-week (or 6-year) team effort to build a grammar for English. How would you organize the effort? How would you track your progress? What kinds of tools or other resources would you want to develop?
- How does your answer to the previous question change if the language is not English? Does it matter what family of language it is? What if you and your teammates don't speak it?

- Your grammar scores well when it finds a parse of high probability. But S1 might produce several different parses of a sentence, dividing the probability among them. Is the scoring method fair?

Honor Code

I encourage students to discuss the programming assignments including specific algorithms and data structures required for the assignments. However, students should not share any source code for solution.

One of the most important principles of modern natural language processing is the use of a true unseen test set. Therefore, whenever we give a problem with training data, development data, and test data, you may not tamper with the submission process or scripts so as to attempt to look at the test data.

Code exists on the web for many problems including some that we may pose in problem sets or assignments. Students are expected to come up with the answers on their own, rather than extracting them from code on the web. This also means that we ask that you do not share your solutions to any of the homework - programming assignments, or problem sets - with any other students. This includes any sort of sharing, whether face-to-face, by email, uploading onto public sites, etc. Doing so will drastically detract from the learning experience of your fellow students.