Homework 5: Hand Writing a PCFG [starter code]

Due: Tuesday April 24 11:59pm Pacific Daylight Time

For this assignment (adapted from Noah Smith and Jason Eisner's homework referenced below) you will be writing a probabilistic context free grammar (PCFG) for a small subset of English. In contrast to the other assignments so far, you won't be doing any traditional programming. Instead, you will be crafting your PCFG as a set of weighted rules written in simple TSV files. We've provided you with a set of tools which, given these TSV files, can evaluate the performance of your PCFG. You will be given one set of example sentences to develop with (dev.sen) and your PCFG will be tested on a hidden set of sentences at submission time.

Also, you are required to have Java working on your system because a significant amount of the starter code is written in Java. However, you do not need to worry about the Java part because we provide the jar file to run the program(s).

Your system, will consist of two sub-grammars:

- S1: A weighted context-free grammar that is supposed to generate all and only English sentences
- S2: A weighted context-free grammar that generates all word strings

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.

Your task is to write the grammar rules and also choose the weight for each rule. The weights allow you to express your knowledge of which English phenomena are common and which ones are rare. By giving a low weight to a rule (or to S2 itself), you express a belief that it doesn't happen very often in English.

PCFG Details

More formally, a probabilistic context free grammar consists of:

- A set of non-terminal symbols
- A set of terminal symbols
- A set of rewrite or derivation rules, each with an associated probability
- A start symbol

For natural language PCFGs, we think of the start symbol as indicating "sentence" (in this case it will be START), and the terminal symbols as the words.

A derivation rule gives one way to rewrite a non-terminal symbol into a sequence of non-terminal

symbols and terminal symbols. For example, S -> NP VP says that an S (perhaps indicating a declarative sentence) can be rewritten as an NP (noun phrase) followed by a VP (verb phrase). For the system we've provided you with, the rules need to be written one per line in a plain text file with the following syntax:

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

The same format is used represent both terminal and non-terminal rules. So for example, a multi-word terminal like the name Uther Pendragon can just be represented by the line:

```
1 <tab> NN <tab> Uther <space> Pendragon
```

This can be confusing if you look at the list of allowed words and try to come up with your own sentences because some of the allowed words only occur in these types of rules. So if you had constructed a sentence in which Uther is not followed by Pendragon, the starter grammar would not be able to parse the sentence. You can even give rules which yield a combination of terminals and non-terminals like:

```
1 <tab> PP <tab> through <space> the <space> THROUGH-NN
```

The parser will basically treat any symbols for which the grammar contains no rewrite rules as terminals and throw an error if it finds any such terminals not in the allowed words list. Also note that lines beginning with the # symbol are considered comments. You can give any of the utilities a set of files containing such rules and they will merge the rules in each file into a single grammar.

In a probabilistic CFG, each rule would have some probability associated with it, and the probability of a derivation for a sentence would be the product of all the rules that went into the derivation. We don't want you to worry about making probabilities sum up to one, so you can use any positive number as a weight. We renormalize them for you such that the weights for all the rules which rewrite a given non-terminal form a valid probability distribution.

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.

We have given equal weights to all the Tag -> word rules, but you are free to change the weights if you like. You are also free to add, remove, or change these rules, as long as you don't add any new words.

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; we give you one to start with, and you are free to change it. 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). Of course, you can invest some effort in trying to make S2 a better n-gram model, but that's a tedious task and a risky investment.

Tools

We've provided you with three basic tools for developing your PCFG. All of the tools revolve around flat tab separated grammar files (ending in .gr) described above and have a relatively simple command line interface.

PCFG Parser

The main way to see what your grammar does is to parse sentences with it. You can use our parser to look at a sentence and figure out whether your grammar could have generated it -- and how, and with what probability. We say that your grammar predicts a sentence well if (according to the parser) your grammar has a comparatively high probability of generating exactly that sentence. 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 out the log probability for that parse and the maximum probability parse tree (with the command line option `-t`). It also computes the perplexity of your grammar on the given sentence file. Because your grade will be a function of your perplexity on the dev and test data sets, this is going to be the key tool for evaluating your grammar. To invoke the PCFG Parser call:

```
$ java -jar pcfg.jar parse dev.sen *.gr
```

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

```
$ java -jar pcfg.jar parse -t dev.sen *.gr
```

PCFG Generator

Another way to see what your grammar does is to generate random sentences from it. For our purposes, generation is just repeated symbol expansion. To expand a symbol such as NP, our sentence generator will randomly choose one of your grammar's NP -> ... rules, with probability proportional to the rule's weight. This program takes a sequence of grammar files and performs a given number of repeated expansions on the START symbol. This can be useful for finding glaring errors in your grammar, or undesirable biases. To generate the default number of sentences, 20, you can just call:

```
$ java -jar pcfg.jar generate *.gr
```

or to generate an arbitrary number of sentences (say, 100), use the -n option:

```
$ java -jar pcfg.jar generate -n 100 *.gr
```

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

```
$ java -jar pcfg.jar generate -t -n 100 *.gr
```

Validate Grammar

This program checks the terminals of your grammar against a hard-coded list of allowed words (also given in the allowed words file). This is useful for making 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. It also makes sure that you have some rule which generates every word in the list of allowed words. The starter grammar files already satisfy this, but this will show you if you accidentally change things for the worse. This utility operates sort of like the unix diff where the first file is implicitly the list of allowed words. Specifically, it will print words in either set difference marked with the following convention:

```
'<' denote allowed words not in grammar
'>' denote words in grammar which are not allowed
```

To invoke the tool, call:

```
$ java -jar pcfg.jar validate *.gr
```

Grading

For this assignment we will use the measure of 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. More formally, if the test set of sentences is $\{s1, s2, s3, \ldots\}$, then your model's perplexity is defined as

$$2^{\frac{-\log_2(p(s1))-\log_2(p(s2))-\log_2(p(s3)),\ldots}{|s1|+|s2|+|s3|+\ldots}}$$

where $|si|$ is the number of words in the ith sentence. Note that

$$-\log_2(1) = 0, -\log_2(1/2) = 1, -\log_2(1/4) = 2, -\log_2(1/8) = 3, \ldots -\log_2(0) = \infty$$

So a high perplexity 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.

It is especially important to note that weight of START -> S1 and START -> S2 are very important for determining the final perplexity of your grammar.

Submitting

In order to submit, run

```
$ java -jar pcfg.jar submit *.gr
```

You will be prompted with the options to submit for either parts (the development set given to you and

a test set of sentences that are hidden to you).

Acknowledgements

We'd like to thank Jason Eisner and Noah Smith for the inspiration for this assignment. We've reproduced several of their original resources in their exact form. Their version of this assignment can be found at: paper, source