Competitive Grammar Writing

Computational Linguistics
Linguistics 409

April 2, 2013

Introduction

In this exercise, you'll write/improve a PCFG for a small subset of English. This won't involve any traditional programming. Instead, you will create your PCFG as a set of weighted rules and write them as tab-separated values. You are provided with tools that can evaluate the performance of your PCFG.

Your grammar should be sophisticated enough to parse a variety of English sentences, but it should not produce any ungrammatical output. Since this is a probabilistic CFG, your grammar should give more probability mass to likely sentences and less probability mass to unlikely ones. You'll use a sample set of sentences (`dev.sen`) to develop your grammar and test how much it can cover.

Of course, this is *competitive* grammar writing. Your grammar should be able to generate sentences that other groups' grammars can't handle. And likewise, it should be prepared to handle as many grammatical sentences from other teams' grammars as possible.

Setup

We'll be forming teams in class on Wednesday (April 3). Each team needs at least one computer. You can also work in parallel on the same grammar. Once you've formed a group and signed-onto the clear machines, you can grab the starter code:

```
mkdir ~/cgw
cd ~/cgw
cp -rf ~km21/ling409/cgw/* .
more README.TXT
```

The language

As mentioned, you'll create a grammar to describe a subset of English: **your sentences must use the words listed in the allowed_words file. You are not allowed to add new words to the vocabulary**. Any sentence that can be produced with these words and judged grammatical by native speakers is fair game. If you take a look through the file, you'll see hundreds of words from various linguistic categories[1]. The more of these words you can accurately handle, the better your grammar will be. You are encouraged to add other part of speech tags for the existing words (e.g. *left* can be a verb or an adjective).

Creating a PCFG

PCFG overview

A probabilistic context-free grammar consists of:

1. A set of non-terminal symbols
2. A set of terminal symbols
3. A set of rewrite or derivation rules, each with an associated probability
4. A start symbol

For the natural language PCFGs in our class, we have been thinking 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).

Files

To create your PCFG, you will be creating and editing grammar files. These all end in the suffix .gr. You begin with three default files:

S1.gr This is a starter grammar that contains a few simple rules. It generates real English sentences, but it's very limited.

S2.gr This is a weighted context-free grammar that generates all possible sentences. If you could design S1 perfectly, then you wouldn't need S2. But since English is complicated and time is short, S2 will serve as your backoff model. (For details, see the Appendix.)

[1] All capitalized words are proper nouns.

Vocab.gr This gives a part-of-speech tag to every word in allowed_words. Feel free to change these tags or create new ones (but, again, not to add or remove allowed_words). 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 VerbTs. 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.

Derivation rules

All derivation rules have the same structure:

```
<weight>  <parent>  <child1>  <child2>   <child3>  ...
```

`<weight>` is an integer value. 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, you express a belief that it doesn't occur very often in English.

In a probabilistic CFG, each rule has some probability associated with it, and the probability of a derivation for a sentence is the product of all the rules that went into the derivation. I don't want you to worry about making probabilities sum up to one in this exercise, though, so you can use any positive number as a weight. The code renormalizes them for you so that the weights for all the rules which rewrite a given non-terminal form a valid probability distribution (you're welcome).

`<parent>` is a non-terminal symbol. All children (e.g. `<child1>`) are either terminals or non-terminals. 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 comments and are ignored. 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.

You can change these files however you like, and you can create new ones, too. However, this can't be stressed enough, you must not create new vocabulary words (terminals).

Hint: If you have multiple laptops in your group you can create more grammar files which you will eventually concatenate into a single group grammar.

Weighting S1 and S2

Two rules your grammar must include are `START -> S1` and `START -> S2`. (By default, these rules are in S1.gr.) The relative weight of these determines how likely it is that S1.gr (with start symbol S1) or S2.gr (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). But 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.

Testing your PCFG

I've provided you with three tools for developing your PCFG. First, you can test how well your PCFG can parse other sentences. Second, you can test how well it can generate sentences of its own. Finally, you can also verify that your grammar produces only the words listed in allowed_words.

Parsing sentences

```
java -jar pcfg.jar parse <sentences> *.gr
```

where `<sentences>` is a sentence file.

This command 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 maximum probability parse tree and the log probability for that parse. It also computes the perplexity of your grammar on the given sentence file. This is going to be the key measure for evaluating your grammar. You will want to lower the perplexity:

$$2^{\frac{-log_2(p(s1))-log_2(p(s2))-log_2(p(s3)),\ldots}{|s1|+|s2|+|s3|+\ldots}} \quad (1)$$

where p(s1) is the probability that your grammar assigns to sentence 1. The starter files include a dev set (dev.sen) that contains a variety of sentences. You will want to try to improve the log probabilities of these sentences with your grammar:

```
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
```

You can also visualize your tree outputs by running this command (which just replaces normal parentheses with square ones)

```
java -jar pcfg.jar parse -t dev.sen *.gr | sed -e 's/(/[/g' | sed -e 's/)/]/g'
```

and copying the resulting trees into: `http://ironcreek.net/phpsyntaxtree/`

Generating sentences

```
java -jar pcfg.jar generate -n 20 *.gr
```

where 20 is the number of sentences to generate (you can change it).

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).

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.

Validating your rules

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

This command checks the terminals of your grammar against a hard-coded list of allowed words (cleverly 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. Such mistakes will 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.

Appendix: S2.gr

The goal of S2 is to enforce the intuition that every string of words should have some (possibly minuscule) 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; I 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.