CS 224N
Competitive Grammar Writing

1 Setup Instructions:

Log into your home directory in corn/myth.

```
cd
mkdir -p cs224n
cd cs224n
cp -r /afs/ir/class/cs224n/pa-cgw .
cd pa-cgw
bash
export PATH=$PATH:~/cs224n/pa-cgw
```

Note: the path variable needs to be reset every time you log into myth/corn.
For csh users:

```
setenv PATH $PATH:~/cs224n/pa-cgw
```

2 Introduction

The competitive grammar writing exercise is meant to be an introduction to hand-written context free grammars. In this exercise you will write grammar rules and manually assign weights to these rules. Using your grammar, you will be able to generate sentences from a fixed vocabulary and parse sentences containing words from this vocabulary.

Your system, which is in competition with those of the other teams, will consist of two sub-grammars:

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

The S1 grammar is where you add the rules for your grammar. This grammar should ideally be able to parse and assign non-zero probabilities to all syntactically correct English sentences that can be generated from our vocabulary. The S2 grammar should be able to assign some small probability mass to all possible word strings possible from our vocabulary. 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 fallback grammar. It will be able to handle any English sentences that S1 can't.

One 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.

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.

Another 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.

You will have to decide how much effort to expend on designing S1 and S2, and how much you will trust S1 relative to S2. Your goal is to describe English accurately. This means that your grammar (especially the S1 subgrammar) should be able to generate lots of syntactic constructions that show up in English. Moreover, it should have a higher probability of generating common sentences, and a low or zero probability of generating ungrammatical sentences.

The goal of this exercise is to build a grammar that produces only grammatically correct sentences. For example, it shouldn't accept 'a man run' or 'the man honest'. Because in English you need to say 'a man is running' (or 'a man runs') or 'the man is honest'. However, your grammar should accept grammatically correct but semantically ridiculous sentences, such as 'the castle drinks temperate coconuts that have successfully carried Arthur'. (Note that we dont worry about capitalization at the beginning of the sentence.)

3 Model

A weighted 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 weight
- A start symbol

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

A word about weights: 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.

3.1 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.

3.2 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.

3.3 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 (even if it's a crummy one generated by another team)!

3.4 Placing Your Bets

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 generated by a team whose grammar is more extensive (since the parser will have to resort to your S2 to get a parse).

On the other hand, if you weight S2 too highly, then you will probably do a poor job of predicting the other teams' 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.

4 Grammar Syntax

Rules are written in the following format:

```
WEIGHT  RULE
```

Rules are written with the left and right hand sides separated by "->". For example:

```
1 S -> NP VP
```

Anything written inside parentheses is considered optional. This is equivalent to the '?' operator in regex. For example, the two rules

```
S -> NP VP
```

```
S -> NP
```

can be concisely written as

```
S -> NP (VP)
```

To indicate an 'or' relationship, use curly brackets with the pipe symbol as delimiter. For example:

```
S -> { NP VP | NP }.
```

This expands to

```
S -> NP VP
```

```
S -> NP
```

'e' is the special symbol that indicates null string. To indicate that sentences can sometimes be blank, you can create the following rule:

```
S -> e
```

Note that in the case of a `WEIGHT RULE` combination such as `1 S -> NP (VP)`, the weight of 1 is given to each the expanded rules `S -> NP VP` and `S -> NP`.

5 Getting Started

The first thing you will probably want to do is add some more part of speech categories by splitting up the 'Misc' category in `Vocab.gr`. For example, you might put in an 'Adj' category. Once you have a new part of speech, you'll want to add one or more rules to the S1 grammar which uses it. But, ***importantly*** you'll also want to add it to the back-off S2 grammar. Just add it to the disjunction in the Markov rule and the S2 grammar will remain able to generate any sequence of nonterminals. Since we are working with a fixed vocabulary, you should not change the terminal vocabulary of the system.

Don't forget to read the comments in each of these files:

- S1.gr
- S2.gr (Back-off grammar)
- Vocab.gr

If you are working in teams, your partner(s) can add rules in a new file. The script considers any file in the pa-cgw directory that ends in `.gr` as a part of your grammar. This might be a more efficient way to work in teams on this exercise.

6 Commands

1. To generate sentences from the grammar you currently have:

   ```
   ./cs224n-generate.sh -n 20 -t
   ```

   Options:

   - t → Print the parse tree.
   - -n → Number of sentences to be generated.
   - -s → In case you want to use a separate start symbol (to test a part of the grammar). The default is START.

   Pipe the output to `prettyprint` to see a nice view of the parse (as nice as we could do, right now)

   ```
   ./cs224n-generate.sh -n 20 -t | prettyprint
   ```

2. To parse the set of sentences in the dev set using your grammar and calculate cross-entropy (see definition in the next section):

   ```
   ./cs224n-parse.sh -i dev.sen -C
   ```

   Options:

   - `-i` → Input sentence file, one sentence per line.

- `-C` → To calculate cross-entropy.
- `-n` → To suppress tree output, in case you just want to see the cross-entropy value.
- `-s` → In case you want to use a separate start symbol (to test a part of the grammar). The default is START.

Again, you can pipe the output to `prettyprint`:

```
./cs224n-parse.sh -i dev.sen | prettyprint
```

3. To make sure your grammar isn't generating new terms not in the vocabulary (this will come in handy when you introduce a new non-terminal symbol and forget to write its expansion rule):

   ```
   ./cs224n-check.sh
   ```

   This script internally calls the binary `check-for-new-terms`. Note that this binary is called automatically whenever you make a call `cs224n-generate.sh` or `cs224n-parse.sh`. Also, note that this will not check for grammars that generate infinitely long sentences.

7 Evaluation

We will see how well your grammar predicts the output of the other teams' grammars (i.e., their randomly generated S1 sentences, excluding any ungrammatical ones). Really we should see how well your grammar predicts some naturally occurring English, like tomorrow's newspaper, but it's more fun to pit the teams against one another. The cross-entropy (or log-perplexity — in effect, the recall of a probability model) is a measure of whether the grammar gives high or low probability to the dev set sample of sentences. It is defined as:

$$2^{\frac{-\log_2 p(s_1)-\log_2 p(s_2)-\log_2 p(s_3)\ldots}{|s_1|+|s_2|+|s_3|\ldots}},$$

where $p(s_1)$ is the maximum probability that your grammar assigns to sentence 1. A smaller number is better.

You need to guess what the other teams are doing and keep up with them. Your grammar must be able to generate their sentences, and indeed do so with high probability. In practice you want your S1 to generate their sentences; although your fallback grammar S2 can generate all possible strings, it consequently has low probability of generating any particular string. So you will try to make S1 extensive enough to handle anything that the other teams can throw at you – and extensive enough that you are throwing tricky sentences back at the other teams!

8 Submission

You will submit your files using a Unix script that we've prepared. To submit your program, first put your files in one directory on Farmshare.
Then submit:

```
cd /path/to/my/submission/directory
/afs/ir/class/cs224n/bin/submit
```