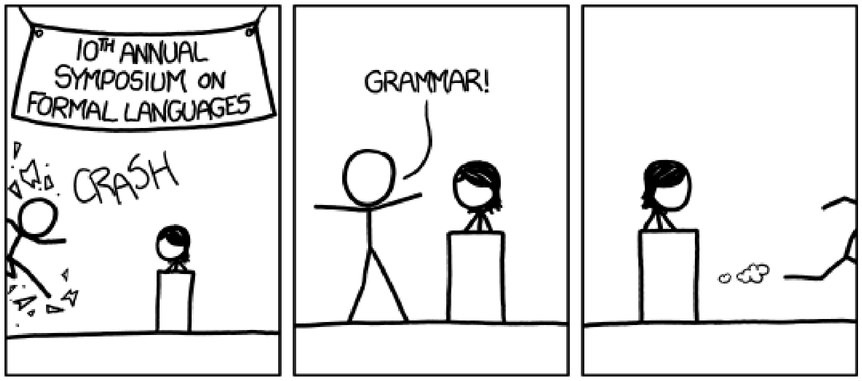
CS159 - Assignment 3

Due: Wednesday, September 23 @ 11:59pm (any time zone)



`http://xkcd.com/1090/`

In this assignment you will construct a probabilistic context free grammar (PCFG) from a set of parsed sentences (a treebank). The goal of this assignment is to familiarize yourself with PCFGs and also to get you comfortable with traversing and manipulating trees and rules.

For this assignment, I am giving you a fair amount of flexibility for how you implement it. You may use whatever programming language you like, however, I have provided you with some starter code in Java for reading in and representing parse trees and PCFGs that will make your life easier if you choose to use Java.

You may (and I would encourage you to) work with a partner on this assignment. If you do, you must both be there when either of you are working on the project and you should only be coding on one computer (i.e. pair programming). If you would like a partner, but don't have one, contact me asap and I will try and pair you up.

As always, read through this entire handout before starting!

Code and Data

`http://www.cs.pomona.edu/classes/cs159/assignments/assign3-starter.zip`

Contains a number of directories/files that you will need for this assignment. The `code` directory has starter code in Java that may be useful if you decide to do this assignment in Java.

Code

- `ParseTree`: This class represents a parse tree. If you pass the constructor a line from one of the parse tree files it will build a `ParseTree` object. A `ParseTree` object is defined recursively with nested `ParseTree` objects to represent the tree. The methods are commented and should hopefully be self-explanatory.
- `GrammarRule`: This is one possible representation of a PCFG rule that includes the left-hand side, the right-hand side list and the probability. I have also implemented the `hashCode` and `equals` method so you may uses these in hashtables, which will likely be useful. Again, the methods are commented and should be self-explanatory.

If there is any question about what the particular classes or methods do, please feel free to talk to me.

Data

The other two directories contain data files which will be discussed below.

Part 1: Constructing a PCFG

The first part of this assignment is to learn a set of PCFG rules from a treebank. Given a set of trees you should first count all of the occurrences of each CFG rule. This should feel somewhat similar to counting n-grams from the last assignment. Once you have all of the occurrence counts, you can then calculate the probabilities associated with each rule and print them out.

There are many ways to do this, but I would encourage you to use a similar data structure setup to the last assignment. When you calculate the probabilities you will be normalizing by the left hand side of the rules and so it will make sense to group these together. I suggest a hash of hashes again, but you may use whatever works for you. The data sets will be large enough that you must be fairly efficient about it. The process takes less than a minute on my laptop to generate rules for the largest data set I'm giving you.

Your output PCFG rules should have the following format. For non-lexical rules (i.e. non-terminal rules):

```
LHS -> RHS1 RHS2 ... RHSN[tab][tab]PROBABILITY
```

where `[tab]` is an actual tab. For lexical rules (i.e. terminal rules, which happen with the RHS is a word):

```
LHS -> * word[tab][tab]PROBABILITY
```

We make the distinction between lexical and non-lexical rules in case we find a lexical word that happens also to be a non-terminal. All of the rules will go in the same file.

For testing purposes, the `example` directory included with the starter contains a very simple treebank file called `example.parsed` with only four parsed sentences. So that you can check to see if you're on the right track, I have also provided two output PCFG files based on this treebank. For this part of the assignment, `example.pcfg` contains the learned PCFGs. When you run your program on this data, you should get the same set of PCFGs (though not necessarily in the same order). This file can also serve as a concrete example of what the formatting of the rules.

Once you've got it working on the small data set, run it on the larger data sets in the `data` directory. I have provided you with two different sized treebanks. `short.parsed` contains 100 sentences to get you warmed up. If that works ok, `simple.parsed` contains around 137K parsed sentences (in fact, the same sentences that we played with for our n-gram language model).

You will hand in your generated grammar for both of these input files. See the section below on what to hand in.

Part 2: Binarizing your PCFG

Training on a large treebank will result in many grammar rules that have more than two symbols on the right-hand side. Because many parsing algorithms such as CKY require rules with just two symbols on the right hand side, a common task is to binarize the grammar.

To binarize a rule, start with the leftmost pair of symbols and replace them with a *new* non-terminal, not already in your grammar. For this assignment, we'll use `X` followed by some number to represent these new non-terminals. Then, create a new PCFG rule from `X` to those two symbols with probability 1.0. Continue this process until the rule is binary. Specifically, given a rule:

A -> B C γ $\qquad$ p

where γ is one or more symbols and p is the probability, we can replace it with two rules:

X -> B C $\qquad$ 1.0
A -> X γ $\qquad$ p

For example, binarizing the rule

```
A -> B C D E F          0.4
```

we would obtain the following new rules

```
X1 -> B C        1.0
X2 -> X1 D       1.0
X3 -> X2 E       1.0
A -> X3 F        0.4
```

There are many ways to binarize a grammar, however, for this assignment, make sure that you binarize as shown above starting with the leftmost pair of non-terminals and incrementally adding one symbol at a time moving right. Intermediate non-terminals should be `X1, X2, X3, ...` To keep things simple, we will binarize each grammar rule independently of any other rules (unless you do the extra credit–see below). This should make it simpler.

You may either write the binarization process as an independent step that takes as input a PCFG and outputs a binary PCFG or you may also build it into your grammar extraction code. If you do this second option, make sure that you learn a normal PCFG first and then binarize the grammar (i.e. don't binarize it on the fly) and also make sure that you can generate grammars in either form normal and binarized.

In the `example` directory I have also included a file `example.binary.pcfg` which includes a binarized version of the example grammar generated. Note that depending on the order in which you process the rules, you may have different `X1`, `X2` and `X3` rules, however, you should have three new non-terminals and they should have the right-hand sides seen in my version.

Once you've got it working, generate binary grammars for the larger parsed files (`short.parsed` and `simple.parsed`).

Extra Credit

There are two possible ways of extending this assignment to get extra credit. Each are worth 1.5 points (for a total of 3 possible extra credit points)

1. Better binarization: The way I've described the binarization process above is wasteful. If at some point during binarization we generate two intermediary rules with the same right hand side we will generate two different rules, for example

   ```
   A -> B C D         0.5
   E -> B C F         0.25
   ```

   would be binarized as:

   ```
   X1 -> B C        1.0
   X2 -> B C        1.0
   A -> X1 D        0.5
   E -> X2 F        0.25
   ```

   A better binarization, which accounted for rule sharing, would be:

   ```
   X1 -> B C        1.0
   A -> X1 D        0.5
   E -> X1 F        0.25
   ```

   For extra credit, implement this rule sharing. The key is to remember the right-hand side of each newly introduced `X`-rule and not generate a new one if you find the same right-hand side again. Notice that you should NOT try and share with the original rule set since they will have probabilities that are not 1.0.

   If you decide to do this, include the original binarization file as well as the binarization with sharing, which you should call `simple.binary.shared.pcfg`.

2. Generating sentences: We've seen in class that you can generate sentences using an n-gram model. You can also probabilistically generate sentences using a PCFG. Write some code to generate random sentences (text only) based on your grammar by starting at `S` and probabilistically generating the tree until you get to the words at the leaves. If you decide to do this, include a file called `example.random.sents` that contains 10 randomly generated sentences based on the grammar from `example.parsed`.

Hints and Advice

- You won't need to write a lot of code for this assignment. However, you should spend some time thinking about what data you need and how you are going to store it. Spending 20-30 minutes upfront will save you a lot of time down the road.
- Recursion is your friend! Parse trees are recursive structures. For most of these methods, using recursion will make your life much easier and your code simpler. For example, look at the `toString` method of the `ParseTree` class.
- As with any large scale data processing task, start small and then work your way up. It will be impossible to tell if you're doing the right thing if you only test on the larger treebanks.
- The `Scanner` class can sometimes be unpredictable on text data with special character (like accented words). For this assignment, I'd advise you to user a `BufferedReader` to read the data (if you use Java).
- ***A note about console output in Eclipse:*** Eclipse has a limited buffer size for the console. If you try and use System.out.println and then copy and paste your results from the console, you will get a truncated version of the answer for the larger grammars.

  You can either use something like `PrintWriter` to explicitly write to a file or, you can also redirect STDOUT (i.e., where System.out.println sends its output to). To do this in Eclipse:

  - Select Run->Run configurations...
  - Click on the class with your main method on the left
  - Click the "Common" tab
  - Check the box next to "File:" and then put a filename (full path) where you would like the output to be redirected to.
  - Click Run
  - Note: you only have to do this process once. Each time after this it will rewrite that file each time you run your program.

When you're done

If you worked with a partner, only one of you should submit and make sure to tag your partner when submitting. You will be submitted both your code and your output (see below). For the code, just submit the source files (e.g., the .java or the .py files).

What to submit

You should submit your full working code. If the naming of the files isn't obvious about where various things are being done, please include a `README` file explaining your organization.

For your `output` you should have the following (make sure you name them exactly this):

- `short.pcfg`: the PCFG grammar for `short.parsed`
- `simple.pcfg`: the PCFG grammar for `simple.parsed`
- `short.binary.pcfg`: the binarized PCFG grammar for `short.parsed`
- `simple.binary.pcfg`: the binarized PCFG grammar for `simple.parsed`
- If you did extra credit, then put the output in this directory as well

Please make sure to follow the folder outline and file naming conventions above.

Commenting and code style

Your code should be commented appropriately (though you don't need to go overboard). The most important things:

- Your name (or names) and the assignment number should be at the top of each file you modify.
- Each class and method should have appropriate JavaDoc comments (docstrings if you do it in Python or whatever is appropriate for the language–come ask me if you don't know).
- If anything is complicated, put a short note in there to help me out if there are any issues.

This is a non-trivial assignment and it can get complicated, which makes code style and comments very important so that I can understand what you did. For this reason, you will lose points for poorly commented or poorly organized code.

Grading

Part	points
PCFG	15
Binarization	15
style/commenting	3
extra credit	3
total	33 + 3 extra