LSA 354

Programming Assignment: A Treebank Parser

Due Tue, 24 July 2007, 5pm

Thanks to Dan Klein for the original assignment.

1 Setup

To do this assignment, you can use the Stanford teaching computer clusters. (You can also just copy the code/data to your own machine, providing you have java (and perhaps ant) installed. But don't forget to delete the LDC treebank data at the end of the Institute.) You need to ssh to a machine with one of the names: elaine, myth, bramble, hedge, vine. (Stanford machines only allow ssh logins which encrypt your password over the network. If you have linux or a Mac, ssh is built in: open a Terminal window and type ssh myth.stanford.edu. On Windows, you need an ssh program. You could get SecureCRT from http://ess.stanford.edu/.

The code for this assignment is at: `/afs/ir/class/cs224n/pa3/java`. The data for this assignment is in `/afs/ir/class/cs224n/pa3/data/parser`. Copy the starter code to your local directory, and make sure you can compile it. The code compiles under JDK 1.5, which is the version installed on the Leland machines.

To ease compilation, we've installed ant in the class bin/ directory. ant is similar in function to the Unix make command, but ant is smarter, is tailored to Java, and uses XML configuration files. When you invoke ant, it looks in the current directory for a file called build.xml which contains project-specific compilation instructions. The java/ directory contains a build.xml file suitable for this assignment (and a symlink to the ant executable). Thus, to copy the source files and compile them with ant, you can use the following sequence of commands:

```
cd ~
mkdir -p lsa354/pa
cd lsa354/pa
cp -r /afs/ir/class/cs224n/pa3/java .
```

```
cd java
./ant
```

If you don't want to use ant, you are welcome to write a Makefile, or for a simple project like this one, you can just do

```
cd ~/lsa354/pa/java/
mkdir classes/
javac -d classes src/*/*/*.java
```

Once you've compiled the code successfully, you need to make sure you can run it. In order to execute the compiled code, Java needs to know where to find your compiled class files. As is standard for Java, this is achieved by setting the CLASSPATH environment variable. If you have compiled with ant, your class files are in java/classes, and the following commands will do the trick. Type printenv CLASSPATH. If nothing is printed, your CLASSPATH is empty and you can set it as follows:

```
setenv CLASSPATH ./classes
```

Otherwise, if something was printed out, enter the following to append to the variable:

```
setenv CLASSPATH ${CLASSPATH}:./classes
```

To look at Penn Treebank data in general, you can cd to:

```
/afs/ir/data/linguistic-data/Treebank/3/parsed/mrg
```

Providing you're registered for the class, you should be able to access this directory. (You should also have signed the LDC data use form!)

Spend some time looking through the principal source file for this assignment:

```
java/src/cs224n/assignments/PCFGParserTester.java
```

Make sure you can run the main method of the `PCFGParserTester` class. You can specify the path to the data directory with the `-path` value. Use the `-data` value to specify which dataset to use. Running:

```
java -server -mx500m cs224n.assignments.PCFGParserTester
-path /usr/class/cs224n/pa3/data/parser -data miniTest
```

will train and test your parser on a few sentences from a toy grammar. Running:

```
java -server -mx500m cs224n.assignments.PCFGParserTester
-path /usr/class/cs2224n/pa3/data/parser -data ptb
```

will train and test your parser on the WSJ section of the Penn Treebank dataset.

2 Building A PCFG Parser

In this project, you will build a broad-coverage probabilistic parser. You're free to build a beam-decoded shift-reduce parser, or implement any other general (P)CFG parsing solution you find interesting and manageable. However, the bulk of the support code assumes that you will be building a parser where the grammar rules have been pre-transformed into exclusively unary and binary grammar rules. The easiest thing to build would be a probabilisitic generalized CKY parser. Another good thing to attempt to build is an agenda-driven PCFG parser.

At the beginning of the main method in `PCFGParserTester` some training and test trees are read in. Currently, the training trees are used to construct a `BaselineParser` that implements the `Parser` interface (which only has one method: `getBestParse()`). The parser is then used to predict trees for the sentences in the test set. For each test sentence the parse given by your parser is evaluated by comparing the constituents it generates with the constituents in the hand-parsed version. From this, precision, recall, and the F1 score are calculated.

This baseline parser is really quite poor—it takes a sentence, tags each word with its most likely tag (i.e., a unigram tagger), then looks for occurrences of the tag sequence in the training set. If it finds an exact match, it answers with the training parse of the matching training sentence. If no match is found, it constructs a right-branching tree, with nodes' labels chosen independently, conditioned only on the length of the span of a node. If this sounds like a strange (and terrible) way to parse, it should, and you're going to provide a better solution.

You should familiarize yourself with these basic classes:

`ling.Tree`	CFG tree structures (pretty-print with `ling.Trees.PennTreeRenderer`)
`Lexicon`	Pre-terminal productions and probabilities
`Grammar, UnaryRule, BinaryRule`	CFG rules and accessors

`Tree` is a linguistic tree class that you will use no matter what kind of parser you implement. `Lexicon` in a minimal lexicon, but it handles rare and unknown words adequately for the present purposes. You can use it to determine the pre-terminal productions for your parser if you like. `Grammar` is a class you can use to learn a PCFG from the training trees. Since the training set is hand-parsed this learning is very easy. We simply set

$$\hat{P}(N^j \rightarrow \zeta) = \frac{C(N^j \rightarrow \zeta)}{\sum_\gamma C(N^j \rightarrow \gamma)}$$

where $C(N^j \rightarrow \gamma)$ is the count observed for that production in the data set. While you could consider smoothing rule rewrite probabilities, it is sufficient for this assignment to just work with unsmoothed MLE probabilities for rules. (Doing anything else makes things rather more complex and slow, since every rewrite will have a nonzero probability, so

you should definitely get things working with an unsmoothed grammar before considering adding smoothing!) `UnaryRule, BinaryRule` are simply the classes the grammar uses to store these learned productions. They each bear the frequency estimated probabilities from the training set.

(If you build an agenda-driven PCFG parser, you will also find `util.PriorityQueue` useful. It is a fast priority queue implementation, but it does not support a promotion / increasePriority operation. If you want to increase the priority of an edge in the queue, you need to add that edge a second time, with the new higher priority, and be aware that whenever you pop an edge off of the queue, it might be a duplicate "dead" edge. We made this design decision because, in our experience, highly-tuned agenda-driven parsers spend a large fraction of their time dealing with the overhead that queues supporting promotion incur. But you are of course welcome to change the priority queue if you'd really like it to support promotion.)

Although it is not required, we strongly recommend that you get your parser working on the `miniTest` dataset before you attempt the `treebank` dataset. The `miniTest` data set consists of 3 training setences and 1 test sentence from a toy grammar. There are just enough productions in the training set for the test sentence to have an ambiguity due to PP-attachment. There are unary, binary, and ternary grammar rules in in training sentences.

As we discussed in class, most parsers require grammars in which the rules are at most binary branching. You can binarize and unbinarize trees using the `TreeAnnotations` class. The implementation we give you binarizes the trees in a way that doesn't generalize the grammar at all. You should run some trees through the binarization process to get the idea of what's going on. If you annotate/binarize the training trees, you should be able to construct a `Grammar` out of them using the constructor provided.

Your first job is to build a parser using this grammar. For the `miniTest` dataset your parser should match the given parse of the test sentence exactly. Once you've got this working you can move on to the `treebank` dataset.

Scan through a few of the training trees in the `treebank` dataset to get a sense of the range of inputs. Something you'll notice is that the grammar has relatively few non-terminal symbols (27 plus part-of-speech tags) but thousands of rules, many ternary-branching or longer. Currently, sections 2 through 21 of the Treebank are read in as training data, section 24 is used as validation data, and section 22 is the test data. (You can look in the data directory if you're curious about the native format of these files.) At the moment, only the training and test set are used, but you are welcome to use the validation set too if you see a use for it. The static integer `MAX_LENGTH` determines the maximum length of sentences to test on (it does not affect the training set). You can lower `MAX_LENGTH` for preliminary experiments, but your final parser should work on sentences of at least length 20 in a reasonable time (5 seconds per 20-word sentence is achievable with some optimization).

Once you have a parser working on the Treebank, your next task is improve upon the supplied grammar by adding 2nd-order vertical markovization. This means using parent annotation symbols like `NP^S` to indicate a subject noun phrase instead of just `NP`. You can

test your new grammar on the `miniTest` data set if you want, though the results won't be very interesting. When you test it on the Treebank the results should be pretty impressive: a 3–4% improvement over the a parser using the original grammar.

At this point an F1 performance of 80% is probably achievable (but don't worry too much about this exact figure—it's just a ballpark). With a little more work it may be possible to improve upon that by 2–5%.

2.1 Coding Tips

Whenever you run the java VM, you should invoke it with as much memory as you need, and in server mode:

```
java -server -mx500m package.ClassName
```

On many machines, you'll get much faster performance than just running with no options.

If your parser is running very slowly, run the VM with the `-Xprof` command line option. This will result in a flat profile being output after your process completes. If you see that your program is spending a lot of time in hash map, hash code, or equals methods, you might be able to speed up your computation substantially by backing your sets, maps, and counters with `IdentityHashMap` instead of `HashMap`. This requires the use of something like an `Interner` for canonicalization. Ask around if you're not sure what that would entail, some people in the class already know this trick.

2.2 The Requirements

To summarize you are expected to implement at least the following:

- A PCFG parser, probably a CKY parser or an agenda-driven parser, that can parse sentences of at least length 20 in a reasonable amount of time (definitely under a minute!).
- An improved grammar with 2nd order vertical markovization.

Anything beyond that is optional. Some extensions are described in the next section.

3 Further Investigations

If you have a parser which, given a test sentence, returns a most-likely parse in a reasonable amount of time, then you have done enough to get full credit. If you are interested in extra-credit there are several ways in which you can embellish your parser:

- One choice is to focus on the grammar, using additional annotation and binarization techniques like horizontal and vertical markovization to improve the accuracy of your parser. The supplied grammar is equivalent to a 1st-order vertical process with an infinite-order horizontal process. You are required to implement 2nd-order vertical markovization, but you could also try 3rd-order vertical markovization or horizontal markovization, meaning forgetful binarization (symbols like `@VP-> ...NP_PP` which omit details of the history, instead of `@VP->_VBD_RB_NP_PP` which record the entire history).

- Another choice is to focus on the efficiency of the parser. You could compare a beam method with exact methods like CKY or the agenda-driven method, or implement a pruning technique like an A* heuristic or a figure-of-merit and compare it to the basic agenda method (this may require substantial effort). Or you could study the trade-offs of pre-tagging the sentence before parsing, rather than letting the parser do the part-of-speech tagging.

- A final choice is to investigate some other aspect of parsing that can be easily tested in your existing code. For example, you might investigate whether some compact subset of a grammar gives nearly the same accuracy. Or if you're interested in cognitive issues, you could study whether correct trees really do tend to have bounded stack depths in one direction or the other (and whether incorrect trees perhaps have different profiles). Or you could do an error analysis of the mistakes your parser makes.

4 Write Up

Your write up should include the following:

- A brief description of your implementation including any important design decisions you made.

- An evaluation of successes and failures of your parser. You don't have to do a detailed data analysis, but you should give a few examples to illustrate any errors your parser makes often and describe its performance in general.

- A discussion of further improvements you might make to deal with some of the observed errors.

- An explanation of any extensions you implemented and an analysis of whether or not they improved performance.

You should turn in a hard copy of your write up. There is no required length but you can probably do it in 4 pages or less. We only need one copy of the write up per group.

5 Submitting the assignment

Submission of the assignment should be by email. Send a message to `manning@cs.stanford.edu` with an attached file archive, which includes the program source code and the write-up.

Make sure that you include all the source code for your programs. I will run the programs and they should run without problems. Please don't include large data files in your submission. Your code doesn't have to be beautiful but I should be able to scan it and figure out what you did without too much pain.