Homework #5: CKY Parsing

Deadline: 11:59pm Sunday 25 November 2013

Please present a report with all your answers, explanations, and sample images or plots. Submit also a soft copy of the source code and binaries used to generate these results. Please note that copying of any results or source code will result in ZERO credit for the whole homework.

Acknowledgment: This homework is adapted from Chris Manning and Dan Jurafsky's Coursera NLP class from 2012.

For this project, you will build a probabilistic parser by implementing the CKY parser. We will be using the Manually Annotated Sub-Corpus (MASC) from the American National Corpus (ANC): http://www.anc.org/MASC/Home.html. The data is provided in the attachment file.

The Code

At the beginning of the main method in `PCFGParserTester.py` some and test trees are read in. Currently, the training trees are used to construct a baseline parser (aptly named `BaselineParser`) that implements the `Parser` interface (which only has two methods: `train` and `get_best_parse`. 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

(For those wondering about `ling.Trees.PennTreeRenderer`, we will not be reading in Penn Treebank data but this Renderer is a good format for displaying any trees, the MASC included.) `Tree` is a linguistic tree class that you will use no matter what kind of parser you implement. `Lexicon` is 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:

$$P(N_j \rightarrow \zeta) = C(N_j \rightarrow \zeta) / \sum_{\gamma} C(N_j \rightarrow \gamma)$$

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 and BinaryRule are simply the classes the grammar uses to store these learned productions. They each bear the frequency estimated probabilities from the training set. Although it is not required, we strongly recommend that you get your parser working on the `miniTest` dataset before you attempt the treebank datasets. The `miniTest` data set consists of 3 training sentences 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 training sentences.

As discussed in the lectures, 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.

Running the Code

Make sure you can run the main method of the `PCFGParserTester` class. You can either run it from the data directory, or pass that directory in as the --path value. Use the --data value to specify which dataset to use (`miniTest` or `masc`). In Java, you will need to compile the project first, which you can do by calling ./ant from thejava/ directory. Running:

```
python PCFGParserTester.py --path ../data/ --data miniTest
```

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

```
python PCFGParserTester.py --path ../data/ --data masc
```

will train and test your parser on the MASC dataset.

Notice we also provide you with run scripts in the python/ directory that have the commands required to run the program.

Your Task

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 MASC dataset.

Scan through a few of the training trees in the MASC dataset to get a sense of the range of inputs. Something you'll notice is that the grammar has relatively few non-terminal symbols but thousands of rules, many ternary-branching or longer. Currently there are 38 MASC train files and 11 test files. (You can look in the data directory if you're curious about the native format of these files.) 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.

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 will be minimal but substantive (a 2% improvement over the parser using the original grammar). At this point an F1 performance in the mid-70% range is achievable.

All in all, there are only 3 places in the code (all in `PCFGParserTester`) that you will have to fill in your own code. Those places are marked with the usual `TODO` markers.

Evaulation

For this assignment we will use your average F1 score to evaluate the correctness of your CKY parser, although in essence you ought to know from the output on the development set (devtest) whether your parser is implemented correctly.

We will grade your assignment based on the F1 you achieve on the development set (devtest).

Requirements

You are required to implement a CKY parser and a vertical markovization of the grammar to achieve an average F1 score of over 75%.

Please submit your code and report in one zip file, named **CMPN463.HW03.firstname.lastname.zip**. For example, if your name is Mohamed Aly, your file should be named **CMPN463.HW03.Mohamed.Aly.zip**.

Grading

1 pts: report and submission file name
2 pts: working CKY parser on miniTest set
2 pts: working CKY parser on MASC devtest
3 pts: working vertical markovization
2 pts: F1 score > 75%