PCYK Parser

Bruce Spang

November 11, 2013

1 Implementation

I implemented a probabilistic CYK parser. The implementation was very similar to the non-probabilistic version.

A major optimization I made was to only consider rules that it was possible to produce for each cell. I indexed the nonproducer rules by their right-hand side, and took all pairs of nonterminals in the children of each cell and looked up the nonterminal they produced. Since there were generally many fewer pairs of nonterminals than rules in the grammar, this made the parser much faster. Before this optimization, it took about 12 seconds to parse 5 sentences with a grammar of about 500 nonterminals. After this optimization, it took about 2 seconds to parse about 100 sentences with the same grammar.

I trained a grammar on the first 3800 trees from the penn treebank included in NLTK. I converted each tree in the treebank into chomsky normal form using NLTK, and counted the number of times each parent had some children. I took the counts and normalized them with log probabilities and add-two laplacian smoothing.

There are two notable details on training the grammar.

By default, NLTK's CNF conversion doesn't eliminate unary rules. This leads to really weird behavior in the CYK algorithm, including failing silently when given with grammar with unary rules. There is a separate function for removing unary rules from a tree.

Whenever the CNF conversion encounters a nonterminal with more than two children, it generates a new nonterminal based on the children's names. Because this is done recursively, it leads to lots of infrequently occuring nonterminals. This increases the grammar size and slows down the CYK parser. NLTK supports order-n markov smoothing, which reduces the number of nodes included in each nonterminal name. This significantly reduces the size of the grammar.

2 Evaluation

I tested the parser on the final 114 sentences from NLTK's penn treebank. The parser sort of worked. See Appendix A for an example parse.

The parser does very well on the training sentences. While unimpressive, this at least suggests that the implementation is correct.

Since I'm doing simple maximum likelihood estimation for part of speech tagging, the parser completely rejects sentences with words it hasn't seen before. This means that it rejects the majority of sentences. Fortunately, it is possible to fix this by using a better part of speech tagger.

When looking at sentences with words it has seen before, the parser manages to get large chunks of the tree right. (See Appendix A). However, it often chooses the wrong start symbol (there are multiple in the penn treebank), or adds a node it shouldn't and then gets hopelessly on the wrong track.

NLTK's ChunkScore reports a precision of 0.413, a recall of 0.413, and an f-measure of 0.413. While I don't believe that those are the actual numbers (since it seems unlikely that precision and recall would be the same), I don't imagine that the actual numbers would be much better.

3 Future Work

It would be great to use a proper part-of-speech tagger to tag the terminals instead of doing maximum likelihood estimation. I'd expect that using a proper tagger would significantly improve the parser's accuracy.

It'd be good to figure out what's wrong with NLTK's ChunkScore algorithm, and get more accurate precision and recall numbers.

The parser is currently horrendously slow. With a grammar of 2157 terminals, it takes 6.7 minutes to parse about 100 sentences from the penn treebank.

A good start would be finding a way to simplify the grammar. I'd like to generate a non-CNF grammar from the penn treebank to start, and then convert the grammar to CNF instead of each individual tree to CNF. I expect that this would reduce the number of nonterminals. If that wasn't enough, I could reduce the number nonterminals in the treebank (e.g. by using N instead of NN, NNP, etc...).

If simplifying the grammar didn't work, there's still a lot of room for optimizing the actual code. I fully expect that I'm generating huge amounts of garbage and doing all sorts of other suboptimal things. It probably wouldn't be too hard to just rewrite the core algorithm in C.

Appendix A: Sample Parses

Sentence

"In other commodity markets yesterday:"

Gold Standard

```
(FRAG
  (PP-LOC
    (IN In)
    (NP (JJ other) (NP|<NN> (NN commodity) (NNS markets))))
  (FRAG|<NP-TMP> (NP-TMP+NN yesterday) (: :)))
```

Parse

```
[FRAG-HLN:-32.77447
  [PP-LOC:-27.92448
    [IN:-3.94681
      In]
    [NP:-23.74094
      [JJ:-4.22286
        other]
      [NP|<NN>:-15.03343
        [NN:-8.30715
          commodity]
        [NNS:-5.09638
          markets]]]]
  [FRAG-HLN|<NP-TMP>:-3.05823
    [NP-TMP+NN:-0.55005
      yesterday]
    [::-1.40957
      :]]]
```

Appendix B: Source Code

See `https://github.com/brucespang/pcyk` for the source code and sample output.