CS224N Homework 2

Jiyue Wang
jiyue@stanford.edu

Ye Tian
yetian1@stanford.edu

1 Implementation

This section mainly explains the details of how the PCFG parser is implemented

1.1 PCFG Parser

We mainly implemented the `train` and `getBestParse` method for the PCFG Parser as the `Parser` interface enforces. The `train` phase is easy, we first annotate and binarize each training tree and then fee the list of binarized tree to `Lexicon` and `Grammar` constructor and save them as private class variable in the `PCFGParser` class.

`getBestParse` is the actual implementation of an improved CKY algorithm described in the course video. There are several points we need to consider:

1.1.1 How to store subproblem results

As many other dynamic programming algorithm, CKY needs to store the result of subproblems. In our problem, the subproblem could be indexed by `i, j, TAG`. It is natual to build a table (for `i, j`), but we choose to build a hashmap `probTable` in order to save memory. The key of the hashmap is an encoded string of `i, j` and the value is a hashmap of `<TAG, probability>`.

1.1.2 How to recover the best parsing tree

We choose to store the information of best parsing TAG to recover the best result instead of traversing the `probTable`. The main consideration here is to save time. Without extra information stored, we need to traverse the tree and get the best splitting index and TAG. Thus we choose to store this information in another hashmap `bestTag`. We also defined a new child class `TagInfo` to encapsulate this information. The best parse tree is generated by `buildTree` in a recursively way.

1.2 Markovization

Markovization is a way to give context to the PCFG model. In the original PCFG model, all the rules are assumed to be independent and the tags are coarse. We thus use vertical markovization to encode more prior context for the tag. On the other hand, we already have `TreeAnnotations.binarizeTree`, which is actually a infinite horizontal Markovization. This will introduce too many subtags which might cause the sparsity problem. Thus, in `TreeAnnotations.annotateTree`, we first did a vertical markovization on the original tree, then used the lostless binarization method and finally prune the tags by horizontal markovization.

1.2.1 Horizontal Markovization

As we will call the `binarizeTree` to binarize the tree before we do markovization, we assume here the input tree has been binarized. We implemented `horizontolMarkovAnnotate` from top to bottom in a recursive way to keep the code clean and it modifies the tags in place to save memory.

1.2.2 Vertical Markovization

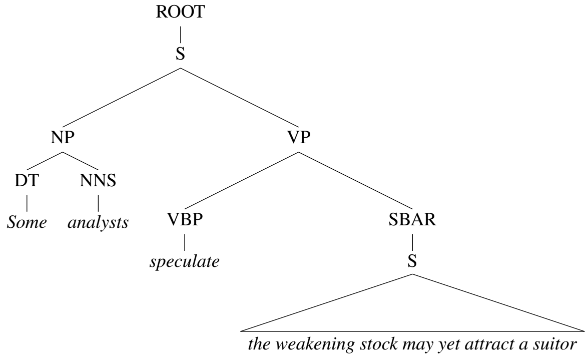
Like `horizontolMarkovAnnotate`, we also implemented `verticalMarkovAnnotate` in a recursive way, but from bottom to top.

2 Error Analysis

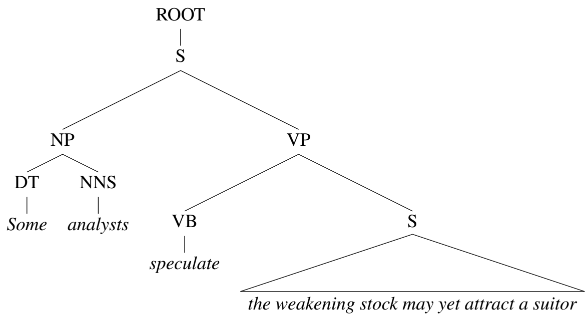
We have a 78.34 F1 score for the naive PCFG parser and a better 81.86 F1 score for the markovized parser. We will analyze the perfomance of the parser by some examples. (In order to show better picture, we deleted some punctuations and unimportant parts.)

2.1 Example 1

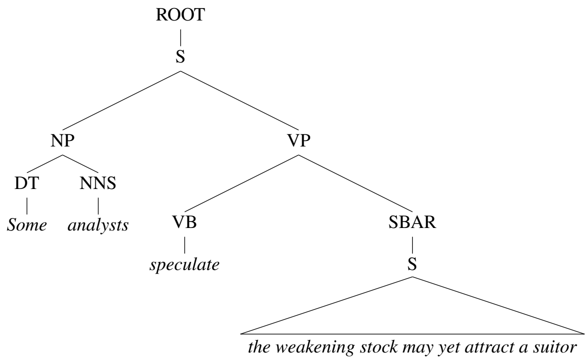
Gold



PCFG



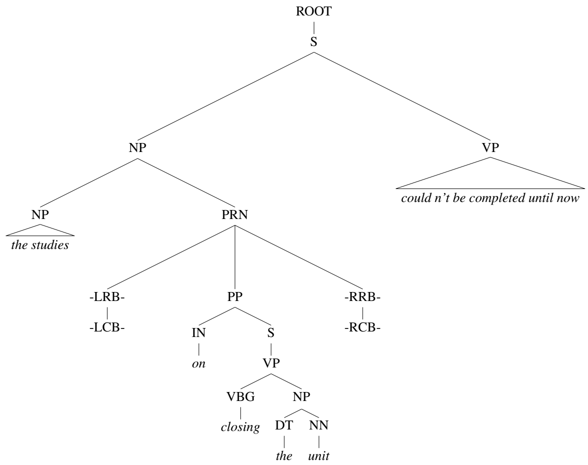
Vertical Markovization



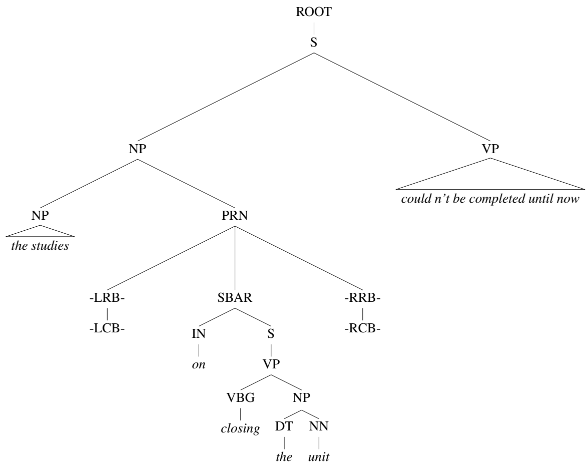
Horizontal Markovization got the same result as pure PCFG. And vertical + horizontal Markovization got the same result as vertical Markovization. In this example, PCFG missed SBAR since adding another layer of SBAR $\rightarrow$ S only decreases the score. But this guess lost context information. This is because of the Lack of Sensitivity to Structural Preferences. Vertical Markovization settles this problem.

2.2 Example 2

Gold



PCFG



PCFG mistakenly assign (SBAR (IN on) to (PP (IN on). This is because the ambiguity of the Treebank PoS tag. IN can be prepositions like on, in, etc., but can also be subordinating conjunctions like if. Perhaps in more cases IN is subordinating conjunction and thus a child of SBAR. It's a lack of Sensitivity to Lexical Information . In this example, Markovization doesn't make any difference. It is because Markovization also does not add any lexical information.

3 Improvement

There is much room for improvement after we add the markovization. If we still go for the structural annotation way, we could try to refine the tags as we did in the competitive gramma writing.

3.1 Split `IN` tag

As we have mentioned earlier, PoS tags in the treebank is quite coarse. All these words (about 177) are tagged as `IN` while, FOR, For, AS, At, As, By, ON, unless, OF, On, Of, IF, onto, IN, If, In, Up, So, Around, de, en, Unless, by, at, as, of, on, if, in, up, v., so, whether, nearest, inside, Against, into, over, before, Through, until, across, Without, Until, Unlike, Despite, Whereas, underneath, under, toward, Off, Out, after, about, above, etc.

If we could split the tags into more categories, say prepositions like ‘in’, ‘on’, ‘for’ and subordinating conjunctions like ‘unless’, ‘if’, etc. we should have some improvement. This can be easily implementedy by storing a map from word to new tags and learn the probabilty based on the new tags. We may even set up some threshold to decide if we should replace the old tag with the new one to avoid data sparsity.

3.2 Latent-annotated PCFGs

Using latent-annotated PCFG, we could easily refine other BIG tags like ‘NP’. According to the paper by Petrov and Klein (2006, 2007), this could get a comparable and even better accuracy than lexicalized model. However, this could be much harder than the first improvement as it involves learning subcategories.