CS224n Problem Assignment 2

Enhao Gong, Yuming Kuang

October 22, 2014

1. Design decisions

1.1 PCFGParser

We essentially followed the algorithm in CKY Parsing videos. Some design decisions we made in the parser are the following:

- Data structure: For the *score* and *back* matrices, we use the following data structure.

```
ArrayList< ArrayList< Map<String, Double> > > score
ArrayList< ArrayList< Map<String, Triplet<Integer, String, String> > > > back
```

For *score*, it's an $(n+1) \times (n+1)$ matrix (n is the sentence length), in which each element is a *map* recording the *tag* and corresponding *score*. Similarly for *back*, whose *map*'s value type is *Triplet*, recording split position, left child tag and right child tag. Here we use *Triplet* of $(-1, child\_tag, "")$ to represent unary rules.

- Optimization 1: When using elements (begin, split) and (split, end) to populate element (begin, end), we have to use all the binary rules whose left child is in (begin, split) and right child in (split, end). The way we implemented here is for each left child tag in (begin, split), we use *getBinaryRulesByLeftChild* to find binary rules corresponding to this left child, then for each binary rule, check whether its right child tag is in (split, end). The advantage of this implementation is that we avoid searching all binary rules, instead we only look at a subset of rules using the left child tag information. Moreover, check right child tag only take $O(1)$ time, so we also avoid iterating the right child tags in (split, end).

- Optimization 2: For the process of adding unary rules, the naive algorithm includes looping the tag set many time which might be time consuming. We observed that only the tags newly inserted or score updated can recursively introduce new tags, so it might be helpful to remember them. Thus, we introduces a queue for these tags:

```
LinkedList< Triplet<String, String, Double> > addTag
```

We first loop all the tags in the tag set once, and push all the new update introduced by unary rules into *addTag* in the form of $(parent\_tag, child\_tag, new\_score)$. Then we only use *addTag* to do update. Each time we pop the front element of *addTag*, check whether this update results in a higher score. If so, do the update to *parent_tag*, and treat *parent_tag* as a new *child_tag* to see if it can introduce new possible updates, push those updates into *addTag*. If not, just continue. When the queue *addTag* is empty, all updates are done. The advantage of this optimization is that we avoid waste time looping all the tag set many times, instead we only check tags that are possible to make new updates.

- Result Tree: At time, we need to use *score* and *back* to construct the best tree parse. Instead of using recursion, we used an iterative way, essentially breadth-first-search, which is faster and memory saving. To do this, we introduce a queue

```
Queue< Triplet<Tree<String>, Integer, Integer> > queue
```

whose element is a $Triplet$ representing current node with label, and the position of corresponding element in *score* and *back*. To start, we push the root node with label 'ROOT' and position $(0, n)$. Then each time, we pop the front element of *queue*, use the node's label tag to search in *back* for its children nodes. One child node for unary rules and two children nodes for binary rules. Having found the children nodes' information, we construct the children nodes, set their labels, put them into their parent node's children list. Finally we push all children nodes into the *queue*. When we arrive at a word, it's a terminal node and no children node is added into the queue. When the *queue* becomes empty, we finish recovering the result tree, and can do unannotation and return.

1.2 Vertical Markovization

In order to add vertical markovization, new annotation methods were created in TreeAnnotations.java. Recursive tree visiting was used to process each node on the tree and add the parent annotation symbols to the node, for example as mentioned in the assignment adding NP to NP^S.

- New Method Function

Firstly, we designed a new method to recursivedly visit nodes and process the annotation symbols. The method take the Tree structure and parent annotation symbol (start with empty string for root) and return a processed Tree structure.

```
    private static Tree<String> myMarkovizeTree(Tree<String> tree, String labelParent) {...}
```

Then we parse the 'augmentedly' annotated Tree into binarization further used in Training.

2 Result and Error Analysis

2.1 Results

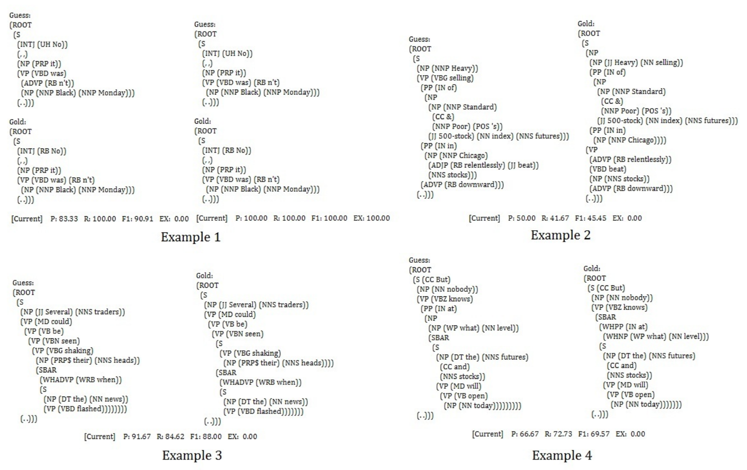
The parsing result of logless parser and 2nd vertical markovization parser is shown in Table 1. The $F1$ performance of logless parser is 78.34%, which is relatively high. And by adding 2nd vertical markovization we are able to improve $\sim$ 4.2% to reach 82.50%.

	P	R	F1	EX
logless parser	81.31	75.58	78.34	20.65
2nd vertical markovization	83.71	81.33	82.50	32.26

Table 1: Results of logless parser and 2nd vertical markovization parser

2.2 Error Analysis

Figure 1 shows some example of our parsing results. Example 1 is a comparison between logless and vertical markovization. Example 2-4 are all taken from result of 2nd vertical markovization parser.



```
Guess:
(ROOT
  (S
    (INTJ (UH No))
    (, ,)
    (NP (PRP it))
    (VP (VBD was)
      (ADVP (RB n't))
      (NP (NNP Black) (NNP Monday)))
    (. .)))

Gold:
(ROOT
  (S
    (INTJ (RB No))
    (, ,)
    (NP (PRP it))
    (VP (VBD was) (RB n't)
      (NP (NNP Black) (NNP Monday)))
    (. .)))

[Current]   P: 83.33   R: 100.00   F1: 90.91   EX:  0.00
```

```
Guess:
(ROOT
  (S
    (INTJ (UH No))
    (, ,)
    (NP (PRP it))
    (VP (VBD was) (RB n't)
      (NP (NNP Black) (NNP Monday)))
    (. .)))

Gold:
(ROOT
  (S
    (INTJ (RB No))
    (, ,)
    (NP (PRP it))
    (VP (VBD was) (RB n't)
      (NP (NNP Black) (NNP Monday)))
    (. .)))

[Current]   P: 100.00   R: 100.00   F1: 100.00   EX: 100.00
```

Example 1

```
Guess:
(ROOT
  (S
    (NP (NNP Heavy))
    (VP (VBG selling)
      (PP (IN of)
        (NP
          (NP (NNP Standard)
            (CC &)
            (NNP Poor) (POS 's))
          (JJ 500-stock) (NN index) (NNS futures)))
      (PP (IN in)
        (NP (NNP Chicago)
          (ADJP (RB relentlessly) (JJ beat))
          (NNS stocks)))
      (ADVP (RB downward)))
    (. .)))

Gold:
(ROOT
  (S
    (NP
      (NP (JJ Heavy) (NN selling))
      (PP (IN of)
        (NP
          (NP (NNP Standard)
            (CC &)
            (NNP Poor) (POS 's))
          (JJ 500-stock) (NN index) (NNS futures)))
      (PP (IN in)
        (NP (NNP Chicago))))
    (VP
      (ADVP (RB relentlessly))
      (VBD beat)
      (NP (NNS stocks))
      (ADVP (RB downward)))
    (. .)))

[Current]   P: 50.00   R: 41.67   F1: 45.45   EX:  0.00
```

Example 2

```
Guess:
(ROOT
  (S
    (NP (JJ Several) (NNS traders))
    (VP (MD could)
      (VP (VB be)
        (VP (VBN seen)
          (VP (VBG shaking)
            (NP (PRP$ their) (NNS heads))
            (SBAR
              (WHADVP (WRB when))
              (S
                (NP (DT the) (NN news))
                (VP (VBD flashed)))))))))
    (. .)))

Gold:
(ROOT
  (S
    (NP (JJ Several) (NNS traders))
    (VP (MD could)
      (VP (VB be)
        (VP (VBN seen)
          (S
            (VP (VBG shaking)
              (NP (PRP$ their) (NNS heads))))
          (SBAR
            (WHADVP (WRB when))
            (S
              (NP (DT the) (NN news))
              (VP (VBD flashed)))))))
    (. .)))

[Current]   P: 91.67   R: 84.62   F1: 88.00   EX:  0.00
```

Example 3

```
Guess:
(ROOT
  (S (CC But)
    (NP (NN nobody))
    (VP (VBZ knows)
      (PP (IN at)
        (NP
          (NP (WP what) (NN level))
          (SBAR
            (S
              (NP (DT the) (NNS futures)
                (CC and)
                (NNS stocks))
              (VP (MD will)
                (VP (VB open)
                  (NP (NN today))))))))))
    (. .)))

Gold:
(ROOT
  (S (CC But)
    (NP (NN nobody))
    (VP (VBZ knows)
      (SBAR
        (WHPP (IN at)
          (WHNP (WP what) (NN level)))
        (S
          (NP (DT the) (NNS futures)
            (CC and)
            (NNS stocks))
          (VP (MD will)
            (VP (VB open)
              (NP (NN today)))))))
    (. .)))

[Current]   P: 66.67   R: 72.73   F1: 69.57   EX:  0.00
```

Example 4

Figure 1: Parsing result examples

- By adding 2nd vertical markovization, our parser can make use of the information of parent tag to improve performance. In Example 1, 'was n't' is parsed correctly in vertical markovization parser (on the right) because it can see that 'n't' is more likely to be tagged with VDB^RB rather than ADVP^RB.
- In longer sentences, especially sentences with several verbs, our parser is sometimes not able to determine the finite form verb. Example 2 is an example that there are two possible verbs 'selling' and 'beat'. Our parser tagged 'selling' as the verb possibly because it's more frequent that the verb is right after the subject noun. However in this example the true verb is 'beat'.
- Our parser is reluctant to parse short clause, because introducing an extra unary clause rule lower the score. In Example 3, 'shaking their heads' is a clause, but our parser parses it as VP, because there's no subject in this clause and making it a clause lower the score.
- Cases with low frequency is hard to parse. As in Example 4, 'at what level' is a WHPP, however out parser simply treats it as a PP, because PP is more common than WHPP.

3. Further Improvements

- **Lexicalization:** If we can lexicalize, the word 'selling' in Example 2 is a hint for NP and the word 'what' in Example 4 may suggest the WHPP.
- **Imprved the Efficiency in Vertical Markovization**

To improve the efficiency of the code, we combined the member functions, including **MarkovizeTree, binarizeTree, binarizeTreeHelper** in the TreeAnnotations class. The final function generalized is:

```
mySecondMarkovizeTree(Tree<String> tree, String labelParent)
{//include markovization and binarization..}
```

- **Added additional parameter for parser**

To further support different markovization level (both vertical and horizontal), we redesigned the Parser to support an additiaonl parameter:

```
[-mode 1]
```

The default value is using 2nd order vertical markovization. This parameter is used in multiple modified functions.

```
public void train(List<Tree<String>> trainTrees,int mode)
{//in PCFGParser.java}
```

In PCFGParserTester.java:

```
int mode_run = Integer.parseInt(options.get("-mode"));
//0 for basic without vertical markovization, 1 for second order markov,
//2 for third order markov, 3 for third order vertical + 2nd order horizontal
parser.train(trainTrees,mode_run);
```

4. Extra Credit

We implemented improved 2nd/3rd order Vertical Markovization and 2nd Horizontal Markovization.

```
if (mode_input==0) {  binarizeTree(myMarkovizeTree(unAnnotatedTree,"")); }
if (mode_input==1) { return (mySecondMarkovizeTree(unAnnotatedTree,"")); }
if (mode_input>=2) { return (myThirdMarkovizeTree(unAnnotatedTree,"","")); }
```

The main function for 3rd order tree is:

```
private static Tree<String> myThirdMarkovizeTree
(Tree<String> tree, String labelParent, String labelGrandParent) {...}
```

In binarizeHorizontalTreeHelper, the annotation string was parsed to reduce the memory in horizontal markovization. As an example,

```
private static Tree<String> binarizeHorizontalTreeHelper
(Tree<String> tree, int numChildrenGenerated, String intermediateLabel) {...}
```

This function was used when mode_run is 3. Example of the parsed markovation is : @VP->VP_VBG_PP_NP can be parsed (find the position of '->' and last two '_')

The results for different Vertical and Horizontal Markovization is:

	P	R	F1	EX
logless parser	81.31	75.58	78.34	20.65
2nd vertical markovization	83.71	81.33	82.50	32.26
3rd vertical	81.31	75.58	80.24	20.65
3rd vertical+2nd horizontal	81.31	75.58	80.34	20.65

Table 2: Results of parsers with different markovization level

Conclusion

- We successfully implemented PCFG CKY parser
- Basic logless parser achieved 78.34% F1 and 2nd vertical markovization implementation improved it by more than 4%.
- Implemented 3rd order vertical markovization and 2nd order horizontal markovization
- Evaluated and investigated other improvements