CS224N Tree Parsing

Jiayuan Ma
jiayuanm@stanford.edu

Xincheng Zhang
xinchen2@stanford.edu

October 24, 2013

1 Implementation Detail

1.1 CKY parsing

Following the pseudo code given in the lecture notes, we implemented the CKY parsing algorithm. Instead of allocating a single large three dimensional array for dynamic programming, we store our dynamic programming results and backtracking information in two single dimensional arrays. In addition, we use a two dimensional array `scoreIdx` to record the mapping from `(begin, end)` pair to the index `scoreIdx(begin, end)` of these two single dimensional flat arrays.

The following are the definitions of data structures used in our implementation of CKY algorithm. As explained before, `scoreTable` and `backTable` are two single dimensional arrays used in dynamic programming and backtracking. `scoreIdx` is used to translate from two dimensional `(begin, end)` pairs into the singleton indices of `scoreTable` and `backTable`.

```
// Single dimensional array for the score
private List<Map<String, Double>> scoreTable;
// Single dimensional array for the backtrack information
private List<Map<String, Triplet<Integer, String, String>>> backTable;
// Two dimensional array for translating pairs into indices
private int [][] scoreIdx
```

During dynamic programming, it is observed that we do NOT need to delete any elements from arrays. Therefore, we chose `ArrayList` and `HashMap` in our implementation.

- `ArrayList` supports $O(1)$ random element accesses and amortized $O(1)$ element insertions. Since we know in advance there will be exactly $(N+1)N/2$ elements in the array, where $N = \Big(1 + \#\text{of words in the sentence}\Big)$. We can allocate $(N+1)N/2$ slots ahead of time to avoid the overhead of memory copying and reallocation.
- `HashMap` stores Nonterminal $\Longrightarrow$ probability mappings for a specific word span of `(begin, end)`. Given a decent hash function, `HashMap` supports $O(1)$ element accesses and insertions, which is good enough in our application.

For backtracking, we use `backTable` to store necessary information needed to construct parse tree after dynamic programming. For each entry of `HashMap` in `backTable`, we store a mapping from a nonterminal to a triplet, which contains splitting point, left subtree labels, and right subtree labels. For unary rules, we marked splitting point of the triplets as -1 so that we know we have only a single child to expand. During backtracking, if we want to access additional information with respect to a particular `(begin, end)` pair, we simply retrieve the corresponding `HashMap` by using the translation stored in `scoreIdx`.

Finally, the translation mapping `scoreIdx` is built on-the-fly. When we are filling the "grid" in CKY algorithm, we simply append the score and backtrack information to the corresponding array, and record their locations in `scoreIdx` for future look-ups. A sample code snippet is given as follows.

```
scoreIdx[begin][end] = scoreTable.size();
scoreTable.add(score);
backTable.add(back);
```

1.2 Unlexicalized Parsing (Extra Credit)

The plain CKY parsing is not enough for some complicated cases. Sometimes in different contexts a word may serve as different purposes. For example, a verb may be used as an adjective. In this case, we want to split a verb phrase (VP) tag to several different ones based on the context. The idea of *vertical markovization*, using tags from parent or even ancestor to split a tag, solves this problem. Therefore, we can learn more fine-grained rules for different contexts. However, vertical markovization creates too many different rules for CKY algoithm and we use *horizontal markovizations* to reduce rules so that we can strike a balance of computational overhead.

- **vertical markovization** We implemented a vertical markovization method which can incorporate parent annotations of any order in the tree. It recursively append parent labels to child and cut those labels which is beyond the scope.
- **horizontal markovization** We implemented the horizontal markovization by modifying the `binarizeTree` method. The original `binarizeTree` is a loseless binarization. We add a parameter to control how many tags can be preserved in the generated intermediate label.

2 Performance Evaluation

Our experiment results are aggregated in Table 1, all of which are averaged performance measures on 155 sentences (with the default max length 20) from Treebank data set. We also timed our implementation (in Table 2) of vanilla CKY and CKY with markovizations on a 2.3Ghz MacPro using 800MB memory on the same data set.
In general, adding markovization (whether vertical or horizontal) helps to boost the performance, which is expected. Another general observation is that vertical markovization seems to give a higher performance gain than horizontal markovization.
In addition to the required 2nd order vertical markovization, we also implemented horizontal markovization as described in the implementation section. It didn't increase the accuracy of parsing

Algorithm	Precision	Recall	F1 score	EX score
Vanilla CKY	81.31	75.58	78.34	20.65
CKY + 2nd-order vertical markovization	83.71	81.33	82.50	32.26
CKY + 2nd-order vertical & 2nd-order horizontal markovization	**84.59**	82.63	83.60	33.55
CKY + 2nd-order vertical & (distant) horizontal markovization	83.82	**85.06**	**84.44**	29.03
CKY + 3rd-order vertical & 2nd-order horizontal	84.02	84.47	84.24	**37.42**

Table 1: Aggregated results of CKY and its variants with markovization on Treebank dataset

Algorithm	Running time for 155 sentences (secs)
Vanilla CKY	660.91
CKY with markovization	2194.90

Table 2: Running time of CKY with/without markovization on 155 sentences with at most 20 words

a lot when compared to vertical markovization. On the other hand, adding horizontal markovization did not have a significant running time gain as expected. We think this is due to that the number of grammar rules that have 4 or 5 nonterminals is comparatively small, which makes no big difference. We also tested the 3rd order vertical markovization since our implementation of vertical markovization can take an arbitrary parameter to control the number of orders we want. Interestingly, 3rd order vertical markovization gave an exceptionally high EX score.

While we were implementing the horizontal markovization, we mistakenly preserved the farthest away tags instead of the most recent tags.[1] For example, `NP->VP->ADJP->NNP` was annotated as `NP->...->NNP` instead of `...->ADJP->NNP`. Surprisingly, the implementation with this particular bug actually produced the highest recall and F1 score (see the 4th row of Table 1). Our guess is that this bug worked so well is because it took into account some long-range dependencies which were missing from the classical horizontal markovizations and helped to parse some sentences with long-range dependencies.

2.1 Error Analysis

Vanilla CKY parsing performs well in relative simple sentences. The parsing trees generated from those sentences tend to have both small depth and breadth. The following is an example.

```
(ROOT
  (S
    (PP (IN At)
      (NP
        (NP (DT the) (NN end))
        (PP (IN of)
          (NP (DT the) (NN day)))))
    (, ,)
```

[1]We followed this StackOverflow link here: `http://stackoverflow.com/questions/12884411/horizontal-markovization`.

```
    (NP
      (QP (CD 251.2) (CD million))
      (NNS shares))
    (VP (VBD were)
      (VP (VBN traded)))
    (. .)))
```

In this example, we demonstrate a case where an adverb and a past participle verb appears together being recognized as an adjective phrase because this is a common phenomenon in English sentences. However in this example, since begun is inside a verbal phrases, it should also be recognized as a verb instead.

```
Gold Tree:
(ROOT
  (S
    (NP (DT The) (NN finger-pointing))
    (VP (VBZ has)
      (ADVP (RB already))
      (VP (VBN begun)))
    (. .)))

Vanilla CKY Results:
(ROOT
  (S
    (NP (DT The) (NN finger-pointing))
    (VP (VBZ has)
      (ADJP (RB already) (VBN begun)))
    (. .)))
```

This is the portion of parsing tree for “has already begun” before the unbinarization of CKY parsing.

```
(VP (VBZ has)
        (@VP->_VBZ
          (ADJP (RB already)
            (@ADJP->_RB (VBN begun)))))
```

Now we introduce the vertical markovization to fix this problem, and here is the portion of new parsing tree for “has already begun”.

```
(VP^S (VBZ^VP has)
        (@VP^S->_VBZ^VP
           (ADVP^VP (RB^ADVP already))
            (@VP^S->_VBZ^VP_ADVP^VP
             (VP^VP (VBN^VP begun)))))
```

We observe that by splitting the RB tag and VBN tag by annotate them with their parent and ancestor, we successfully parsed it with correct tag since we learn a separate rule for RB and VBN in this situation.

Another observed problem is that sometimes the highest score parsing may not lead to a sentence (label “S”).

```
Guess:
(ROOT
  (PRN (-LRB- -LRB-)
    (S
      (NP (NNP See))
      (VP (VBD related)
        (NP
          (NP (NN story))
          (: :) (`` ``)
          (NP
            (NP (NNP Fed) (NNP Ready))
            (TO to)
            (NP (NNP Inject) (NNP Big) (NNPS Funds)))
          ('' '') (: --)
          (NP (NNP WSJ) (NNP Oct.) (CD 16) (, ,) (CD 1989)))))
    (-RRB- -RRB-)))
```

2.2 Possible Improvement

The biggest issue of vertical markovization is that it will introduce too many tags and it will make the learned grammar overfit to the training data. Though horizontal markovization has the potential to merge the tags back, it may be better that we only add some specific vertical markovization to a manually defined subset of tags, for example, verbal phrase has a higher chance to be interpreted differently while an adverb can hardly be recognized as something else.