CS224N Fall 2014 Programming Assignment 2

SUNet ID: tzhang54, jiajihu
Name: Tong Zhang, Jiaji Hu

1 Implementing a CKY Parser

1.1 Algorithm and Naive Implementation

To implement the CKY Parser, we followed the pseudocode introduced in the videos. The main idea of the algorithm is that we build up the parse in a bottom-up manner. During the process, we store intermediate results to speed up the process. As introduced in the videos, the time complexity of the CKY algorithm is $O(n^3)$.

We started with a naive implementation of the algorithm. Note that the key design choices in the implementation is what kind of data structure we choose to store the intermediate results. In particular, the two tables `score` and `back`.

From the very start, it was clear that we cannot use a 3-d array to store these tables, as the third dimension will be very large but sparse, due to the large number of distinct words in the treebank, and we will quickly run out of memory. Therefore, we chose to use a Hashmap data structure. We chose `HashMap<Triplet<Integer, Integer, String>, Double>` for `score`, and `HashMap<Triplet<Integer, Integer, String>, Triplet<Integer, String, String>>` for `back`. Also, to keep track of which tags had non-zero values in a block, we kept a `HashMap<Pair<Integer, Integer>, Set<String>>` called `tagDict` to get that set of tags.

Looking back, this setup was highly inefficient. Using the optimizations described below, we were able to dramatically improve our speed performance, resulting in a setup that could parse the test set in less than 3 minutes, averaging one parse per second, and handle sentences of length 20 in 2 seconds.

1.2 Optimizations

1.2.1 Data Structures

First, we came across the `CounterMap` data structure provided in the code library. We realized that it was quite convenient for our use case. Using a `CounterMap<Pair<Integer, Integer>, String>` for `score`, we were able to get rid of `tagDict`, since the information was already in the Map.

Next, we looked into using `IdentityHashMap` to further improve the speed of operations in `score`. Since a data structure that incorporated `IdentityHashMap` into `CounterMap` was not provided, we wrote our own `IndentityCounterMap`, which used an `IdentityHashMap` in the first step, which is `Pair<Integer, Integer>` to `String`. We didn't use it for the `String` to `Double` step because we figured that we would then need to get canonical representations for `String`s, which might be computationally heavy and also heavy on memory. In practice, doing that for Strings did negatively impact our performance.

For the `back` table, we also wanted to incorporate `IdentityHashMap`. We found out through trials that hashing and getting back `Triplet`s was slightly slower than our current solution (and also that there was a terrible bug in the skeleton code for `Triplet` that causes countless unnecessary hash collisions). We had the idea that since we were already using an `Interner<Pair<Integer, Integer>>` for `score`, we might as well utilize that also for `back`. Therefore, we came up with a similar data structure to `IdentityCounterMap`, and named it `IdentityTripletMap`. We implemented that class ourselves.

Using these data structures, we were able to dramatically improve our runtimes. In addition, we were able to shave a bit more off our runtime by doing code flow analysis.

1.2.2 Code Analysis

We carefully analyzed our code to see if we could save a bit of computation here and there. For example, by moving our interning towards the top of the loops, we could make sure that we didn't repeat that work.

Also, we looked at the `while(added)` loop given by the pseudocode in the videos. We see that the loop goes over all `A,B` in `nonterms` and `A->B in grammar`. We realized that on the second iteration of the loop, we really only need to look at the items newly added in the previous iteration. Therefore, if we just keep track, we can look at much fewer candidates for the unary rule in subsequent iterations. Implementing something to that effect actually sped up the runtime by around 10%.

There were many similar tweaks to the code. By intuition and also trial and error, we worked out what data structures and what code flow allowed the parser to run faster.

2 Adding Vertical Markovization

2.1 2nd Order Vertical Markovization Implementation

We implemented vertical markovization to the unannotated tree in a recursive manner. Each node was passed in with the label of the parent node, and if the node is not a leaf, we update the new label to `childLabel ^parentLabel`. For `ROOT`, the parent label was an empty string.

2.2 Markovization Effect Analysis

As reported in the results section, adding 2nd order vertical markovization improved F1 score by more than **4%** from 77.57 to 81.64. The number of exact match cases also increased by more than 10%.

This improvement in results verified the points of integrating vertical markovization addressed in the *Accurate Unlexicalized Parsing* paper. In the basic (1st order vertical) grammar, the cateogry symbols are too coarse to adequately render the expansions independent of the contexts. It is hard to differentiate various expansions from the same category without integrating ancestor labels.

```
Gold:
(ROOT
  (SINV (‘‘ ‘‘)
    (S
      (NP (PRP It))
      (VP (VBD screwed)
        (NP (NNS things))
        (PRT (RP up))))
    (, ,) (’’ ’’)
    (VP (VBD said))
    (NP (CD one) (JJ major) (NN specialist))
    (. .)))
```

```
Guess: Basic
(ROOT
  (SINV (‘‘ ‘‘)
    (S
      (NP (PRP It))
      (VP (VBN screwed)
        (NP (NNS things))
        (PRT (RP up))))
    (, ,) (’’ ’’)
    (VP (VBD said)
      (NP (CD one)))
    (NP (JJ major) (NN specialist))
    (. .)))
```

```
Guess: Markov V2
(ROOT
  (SINV (‘‘ ‘‘)
    (S
      (NP (PRP It))
      (VP (VBG screwed)
        (NP (NNS things))
        (PRT (RP up))))
    (, ,) (’’ ’’)
    (VP (VBD said))
    (NP (CD one) (JJ major) (NN specialist))
    (. .)))
```

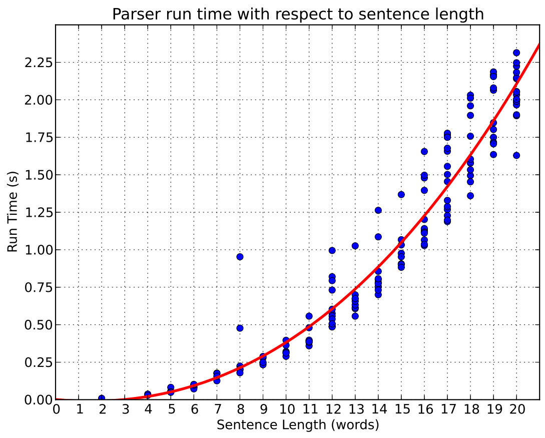
In the example above, the basic parser got 70.59 F1 score, but 2nd order VM got a perfect match. This was because 2nd order VM was able to caputre the vertical structure of the second sentece ”said one major specialist” and treat ”one major specialist” as a single `NP`. In contrast, 1st order VM was not able to differentiate `NP` and `NP^VP` and thus treated ”one” as `NP^VP` by mistake.

3 Results

3.1 Run Time

Algorithm	Total Runtime (s)	Avg. Runtime (s)	Length 20 (s)
1st Order Vertical	160.32	1.03	2.04
2nd Order Vertical	253.54	1.64	3.50
3rd Order Vertical	344.83	2.22	4.58
1st V + 1st H	373.05	2.41	5.02
2nd V + 1st H	352.20	2.27	4.68

Below is the graph for runtime versus sentence length for the basic parser with 1st order vertical markovization. Since the runtime should be $O(n^3)$, we fitted a third order polynomial curve to the data points. As shown below, the curve fits quite well.



3.2 Parsing Score

Algorithm	Precision	Recall	F_1	Exact Match
1st Order Vertical	80.76	74.63	77.57	20.65
2nd Order Vertical	83.64	79.73	81.64	31.61
3rd Order Vertical	82.13	82.28	82.20	36.13
1st V + 1st H	80.68	79.96	80.32	29.68
2nd V + 1st H	84,94	85.60	**85.27**	34.19

4 Extra Credit

4.1 Higher Order Vertical And Horizontal Markovization

4.1.1 3rd Order Vertical Markovization

We implemented the 3rd order vertical markovization in a similar manner as the 2nd order one. At each recursive call, the grandparent node label was also passed in addtion to the parent node label. The label for the current node was updated to
`childLabel ^parentLabel ^grandparentLabel`.

4.1.2 Horizontal Markovization

We also implemented 1st order and 2nd order horizontal markovizations. For n-th order horizontal markovization, we kept track of the previous n sibling labels as the context. Horizontal contexts were marked with `^@`.

4.1.3 Combination And Results

We tried different combinations of vertical and horizontal markovizations and reported the results in the section above. As expected, 3rd order vertical markovization improved results from 2nd order vertical. Adding horizontal markovization improved results in general. The combination of 2nd V and 1st H gave the best F1 score of **85.27**.

However, we were not able to use 1st order horizonal with 3rd order vertical, or 2nd order horizontal at all. In those cases, the grammar became too complicated and sparse. We were not able to parse some of the test data because the grammars were never seen in training. If provided with more training data, we would expect to be able to run combinations of higher order markovizations.

5 Collaboration

Jiaji focused on the parser implementation and optimization. Tong was focused on markovization implementation and testing. We wrote the report together.