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 retrieve 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 gt 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 an `Interner` for `String`, which might be computationally heavy and especially heavy on memory.

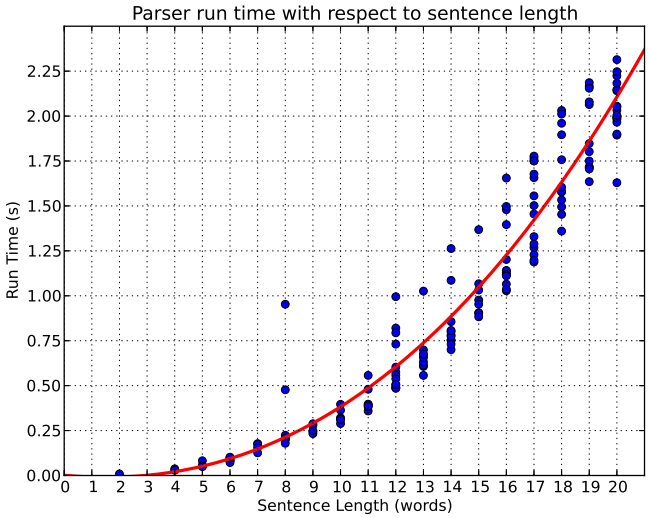
1.2.2 Code analysis

20%

1.3 Results

1.3.1 Run Time

Algorithm	Total Runtime (s)	Avg. Runtime (s)	Length 20 (s)
Basic	160.32	1.03	2.04



1.3.2 Parsing Score

Algorithm	Precision	Recall	F_1	Exact Match
Basic	80.76	74.63	77.57	20.65

2 Adding Vertical Markovization

3 Extra Credit