**Stanford CS224n – Project 2**

**By Patrick Manion**

This project explored an implementation of the CKY for training a probabilistic context-free grammar parser using the Wall Street Journal section of the Penn Treebank. The model was built on trees 200 through 2199 and then tested on a holdout of trees 2300 through 2319.

**1. Implementation Details**

There were several key decisions in the implementation of the CKY algorithm.

One of the biggest decisions was how to store the probability scores for each row, column, and tag. This matrix is generally very sparse as only the upper-right triangle of the matrix is filled and, even for these cells, the majority of tags have zero probability. I initially tried storing these in a series of Java Maps to save memory. However, the numerous look-ups and updates in the map ended up consuming a large amount of CPU time (25-50%). As a result, I also tried storing the values in a full matrix of double[][][], which resulted in a significant runtime decrease. Speed was more of a constraint in this project than memory, so the doubles array was used in the final implementation.

Another key implementation decision was how to work through expanding unary rules. The pseudo-code of CKY suggests exhaustively checking every possible tag combination, checking if they exist in the grammar, and calculating the resulting probability (if possible) until no cells are updated. This ended up being the primary bottleneck in the code as even simple word tags like Noun were often expanded to over 5,000 tags after accounting for all unary rules. A massive speed-up was found by keeping a set of the tags that were updated in the last loop and only checking those for in the subsequent loop. All tags not added to this set either had no matching rules or the rules that applied did not have a better probability than the existing path and thus need no further evaluation. Storing this set requires additional memory, but the performance benefit appears to justify its use in all but the most memory-constrained applications.

Another important decision was how to find the binary rules to evaluate. After some experimentation, the fastest approach appeared to be getting all rules matching the left child (using a map created during training), all rules matching the right child (using a map created during training), and then keeping only rules that appeared in both sets. This is again a trade-off between the additional memory needed to keep the maps and the speed boost provided.

A final implementation choice was how to reconstruct the optimal tree after the CKY algorithm completes. While it can be reconstructed using the probabilities, this adds additional runtime and processing complexity at the end, and so three integer matrices were used to store the best split value, best left child tag id, and best right child tag id for every parent row, column, and tag. These were dynamically updated every time a new best path was discovered during the CKY algorithm, and they allow the optimal tree to be reconstructed almost instantly with a simple recursive algorithm.

**2. Performance**

Two primary models were evaluated as part of the project. The first is a simple PCFG with no parent information but keeping track of all prior sibling tags (infinite horizontal Markovization). The second model adds 2nd order vertical Markovization so each tag retains its parent relationship (e.g. S🡪NP becomes S^ROOT🡪NP^S). The results on the test data are shown below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Precision** | **Recall** | **F1** | **EX** |
| **Base PCFG** | 81.44% | 75.70% | 78.46% | 20.65% |
| **+2nd Order Vertical Markov** | 83.71% | 81.33% | 82.50% | 32.26% |

The results show that the base PCFG already does fairly well, but adding 2nd order Markovization provides a significant boost of over 4% in F1.

Looking at the errors, one of the most obvious issues is that seemingly small differences in attachment can cause a huge drop in F1 due to how the metrics are calculated *especially* for very long sentences. For example, the sentence *“Stock-index futures contracts settled at much lower prices than indexes of the stock market itself.”* is parsed almost entirely correctly except for the final word *“itself”*, which is incorrectly labeled as an NP-PRP attached as the object of *“settled”*. This results in an F1 of 59.26% for this sentence. This would hopefully be an example where additional vertical context would help, but the same error exists in both models.

Another example of a common error is long prepositional phrases accidentally being broken up or wrongly attached by the model. For example, the sentence *“The balance is supplied by a host of smaller exporters, such as Australia and Venezuela”* has a long prepositional phrase starting with *“by”*. The base model, which has no vertical context, assumes that *“by a host”* and *“of smaller exporters”* are separate PP’s attached to *“supplied”* and that *“such as Australia and Venezuela”* is an adjective phrase after a linking verb. Luckily, in this case, the 2nd order Markovization captures the context and correctly nests everything under one prepositional phrase, which improves the F1 from 60.87 in the base model to 86.96.

The base PCFG ran through the test set in 6m 26s while the 2nd Order Markov model ran in 11m 31s, which are well below the guidance limits of 10 minutes and 20 minutes as a result of the optimizations mentioned in Section 1.

**3. Improvement Opportunities**

One key area for improvement is the storage of probabilities. Rather than the double[][][] array approach, small list could be used to store separate double[][] arrays for each row, which could eliminate the wasted space from the lower-left triangle that is never filled. If memory was more of a constraint, the Map approach could also be sped up using custom hashing functions rather than the default Java HashMap.

Another opportunity to improve both the memory required and runtime is to add a search heuristic such as beam search. The probabilities for many cells are so small that it’s very unlikely they will end up being the best at the end, so they could likely be dropped with only a small impact on overall performance. Careful tuning of this approach would be required as it may depend on the specific test dataset.

Another opportunity is to optimize or remove the best tree information stored in the three integer arrays. Similar to the probabilities, more careful data structure can eliminate a lot of wasted space in the lower left triangle. Second, the three numbers could be more carefully stored in a single array using bit-level encodings or even by hashing the numbers into buckets and evaluating the small number of collision at the end. Similarly, it’s also possible to store some subset of the three matrices or even the hash-code of the best rule itself and doing a small number of calculations at the end depending on the specific implementation.

**4. Model Extensions**

Several extensions to the base models were implemented. First, an additional level of 3rd order vertical Markovization was tried, which showed a significant improvement in performance. In addition, a test was conducted on whether to include the pre-terminal tags in the Markovization as this seems a potentially easier task for the parser. The results are shown below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Precision** | **Recall** | **F1** | **EX** |
| **Base PCFG** | 81.44% | 75.70% | 78.46% | 20.65% |
| **2-Vertical** | 83.71% | 81.33% | 82.50% | 32.26% |
| **2-Vertical**  (excluding Pre-Terminals) | 82.34% | 79.61% | 80.95% | 27.10% |
| **3-Vertical** | 83.52% | 84.41% | 83.96% | 37.42% |
| **3-Vertical**  (excluding Pre-Terminals) | 82.85% | 82.16% | 82.50% | 27.10% |

The results show that 3rd order vertical Markovization provides a material boost even above the 2nd order vertical Markovization – boosting F1 by almost 1.5%. The results also indicate that the pre-terminal vertical Markovization is very important, providing almost half the boost from the Base PCFG to the 2nd order vertical Markov model. Interestingly, including the pre-terminals in the vertical Markovization add 1.55% in the 2-Vertical model and 1.46% in the 3-Vertical model, which suggests that most of the benefit from the 3rd order vertical Markovization comes from splits above the pre-terminal. This aligns with the intuition that this is a somewhat easier part of the parsing.

Finally, different settings for the horizontal Markovization were tried comparing the default ∞-Horizontal approach with versions that discarded some of the prior tag information if it became too distant.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Precision** | **Recall** | **F1** | **EX** |
| **Base PCFG** | 81.44% | 75.70% | 78.46% | 20.65% |
| **2-Vertical ∞-Horizontal** | 83.71% | 81.33% | 82.50% | 32.26% |
| **3-Vertical ∞-Horizontal** | 83.52% | 84.41% | 83.96% | 37.42% |
| **2-Vertical 1-Horizontal** | 84.43% | 82.28% | 83.34% | 32.90% |
| **3-Vertical 0-Horizontal** | 83.76% | 84.35% | 84.05% | 35.48% |
| **3-Vertical 1-Horizontal** | 84.14% | 84.88% | 84.51% | 38.06% |
| **3-Vertical 2-Horizontal** | 84.53% | 85.18% | 84.85% | 38.71% |
| **3-Vertical 3-Horizontal** | 84.09% | 84.88% | 84.48% | 38.71% |

The results indicate that reducing the horizontal Markovization provides a material boost of almost 1% for the 3rd order vertical Markov model. Interestingly, the model that completely discarded any horizontal information actually performed slightly better than the infinite Horizontal model. All of these results align with the intuition that more complexity can be better up to a point, but it also requires far more data to get accurate probability estimates. The 3-Vertical 2-Horizontal model seems to balance this the best by capturing context for the word but not so much that probability estimates are noisy.