CS 224N Assignment 2: CKY Parser

John Miller, Catalin Voss [*]

1 Introduction

Given a sentence, our goal in this programming project is to find a parse tree that captures its syntatic structure according to some probabilistic context-free grammar (PCFG).

2 Implementation

In this section, we discuss the training of a statistical treebank parser and implementation of the CKY algorithm for parsing.

2.1 Parse Tree Construction

2.1.1 Training

In the given starter code, we take an empirical approach to learn rules of the grammar directly from annotated treebank data. Efficient parsing algorithms require grammars to consist only of binary and unary rules, so as a preprocessing step we perform lossless binarization on each of the input trees. Then, we learn rules of the grammar and their associated probabilities directly from the annotated input tree.

For some observed rule $A_i \to B_j$ in the treebank, we estimate the probability of this rule using the maximum likelihood estimator

$$\Pr\left(A_i \to B_j\right) = \frac{\text{count}(A_i \to B_j)}{\sum_{j'} \text{count}(A_i \to B_{j'})}.$$

During training, we do not employ any form of smoothing for nonterminal rewrite rules. Smoothed estimates would likely improve accuracy of the parser, but these gains come directly at the expense of training and parsing speed. Since each nonterminal rule already has nonzero probability, unsmoothed estimates are suitable.

On the other hand, for preterminal rules, we perform laplacian smoothing for words that appear less than 10 times in the corpus. This ensures that the grammar is able to parse sentences that contain rare or unknown words.

2.1.2 Vertical Markovization

In order to capture more specific tags in our grammar, we implemented n-order vertical Markovization based on Klein & Manning's 2003 paper "Accurate Unlexicalized Parsing." For each node of every tree in the treebank, we modified the node's label we include the label of each of the $n-1$ ancestors to better encode vertical dependencies in the training data. We chose to implement this as a recursive array-based procedure that annotates the tree bottom-to-top following a Depth-First-Search-like node discovery. This proved to be more efficient and easier to trace than comparable string-based approaches that removed unnecessary annotations in a second pass. Note that this step is performed before binarization so as to preserve the conditioning on the parent node's label in the tree.

[*]Collaboration: John and Catalin implemented and debugged the CKY parser together. Catalin implemented vertical Markovization, and John performed error analysis of the parser and authored most of the writeup.

2.1.3 CKY Parsing Algorithm

After training a PCFG on annotated treebank data, given some new sentence, we use the CKY parsing algorithm to produce the most probable parse. CKY employs a bottom-up dynamic programming strategy. For each subsequence of the sentence ranging, the algorithm maintains a collection of the atoms and their associated probabilities that can generate the subsequence. This is reasonably efficient: for sentence of length n and a grammar of size $|G|$, the algorithm does $O(n^3 \cdot |G|)$ work.

Specifically, the algorithm first considers all sequences of length 1. Using the atoms that generate the sequences of length 1, the algorithm considers all sequences of length 2, and so on. When considering a sequence from i to j, the algorithm considers each possible split k and checks if this subsequence can be generated by a rule $X \to Y\ Z$ for atom Y in range i to k and atom Z in k to j.

The probability associated with each rule is simply the product of the probabilities of the rule and the probabilities of the right hand side. If multiple splits yield the same rule, then we choose the split with higher probability, breaking ties arbitrarily. After adding a rule to the table, we store a backpointer to the atoms and split that produced the rule to allow for fast reconstruction of the tree. After each entry is completely filled, the algorithm rewrites binary rules as applicable unary rules if those yield higher probabilities. Finally, after CKY finds rules that span the entire sentence, we can recursively follow the trail of backpointers to reconstruct the most probable parse.

2.1.4 Data Structures

The main design challenge in implementing the algorithm is the choice of data structure for storing the table and rules kept in each entry.

The set of all spans is dense, requiring $\frac{n(n+1)}{2}$ total entries – one for each possible span i, j. In order to fill in the table for some span, the algorithm requires accesses to each table in each split i, k, j in a tight loop. Therefore, we chose to represent the table itself as list with canonical indexing to allow for fast $O(1)$ lookup of some entry within the table.

For a fixed span, the algorithm requires both iteration over the rules generating the span, as well as the ability to lookup single rules. Consequently, we represented each entry in the table as a map from atoms in the grammar to their associated probabilities to allow for expected time $O(1)$ lookups and easy iteration over elements. The specific choice of map and runtime dependencies are discussed in Section 2.2. For the same reason, and to simplify the code, we use the same datastructure to represent backpointers for each entry.

2.2 Code Optimization

Our initial implementation of the CKY in Java used an `ArrayList` of `HashMaps` to store each cell in the table. After training on the full WSJ section of the Penn Treebank, this naive implementation took an average of 24 seconds to parse sentence of length 20. A code profile revealed that 40% of this time was spent in the `HashMap` methods `get()` and `containsKey()`. To reduce this overhead, we replaced the `HashMap` with an `IdentityHashMap`. This data structure requires cannonicalization of each key so that `==` can be used instead of the more costly `equals()` method. Using the Interner class for canonicalization of keys, the resulting implementation requires an average of 9 seconds to parse sentences of length 20, and a profiler confirms that only 13% of time is spent in comparision methods. Finally, this second code profile revealed the majority of running time was now spent in memory allocations for our map data structures. We were able to achieve an additional 2.4× speedup in overall running time by reusing datastructures throughout CKY parses and reducing

Average Performance on the WSJ evaluated on `corn1`					
	P	R	F1	Ex	Avg. Parse Time (s)
Baseline Grammar	81.31	75.58	78.34	20.65	2.4
Second-Order Markov	83.71	81.33	82.50	32.26	5.05
Third-Order Markov	83.52	84.41	83.96	37.42	8.64
Fourth-Order Markov	81.96	83.46	82.70	31.61	12.2

the allocations to a bare minmum. This brought our overall runing time on the test set from 15m34.943s to 6m47.862s when evaluated on `corn1`.

3 Evaluation

In this section, we perform error analysis on the parser implementation and the vertical markov models described in section 2. After training the PCFG on the WSJ section of the Penn treebank each sentence in the grammar was parsed using the CKY parser. We experiment with modifying our grammar with higher order vertical markovization by modifying the tags on the annotated trees before learning the grammar. As a baseline, we use the grammar generated directly from the annotated trees. The results of this experimentation are presented above. The average parse time is the average over all 155 sentences in the validation and test sets.
Note: To run our evaluations for different orders of markovization, simply change the `markovOrder` flag in the file `TreeAnnotations.java`.

3.1 Analysis

The baseline result is somewhat positively surprising: Our parser achieved an F1 score of almost 80% with the default annotation scheme. This suggests that the broad categorical tags in the default set are sufficient to correctly label most parts of the sentence, especially parts with frequently observed structure. Consequently, the parser achieves high or perfect accuracy on most short, simple sentences in the treebank test set. However, in order to correctly predict more complex structure, these features are insufficient.

For example, the grammar produced by the default annotation scheme consistenely mislabelled sentences that required $SBAR$ tags. The sentence "Diamond Shamrock Offshore Partners said Y" is correctly parsed as as NP for "DSOP" and a VP for "said Y", likely because these are simple and frequently observed rules. However, the VP for "said Y" is transformed via a unary rule to S. This is incorrect, as "said Y" is not a simple declarative, but a clause introduce by a subordinating conjuction, i.e. SBAR. A similar errors occurs in the sentence "The company said improvement ...", where "said X" was parsed as "S X" rather than SBAR. The reason for these errors is that the probabilities are not conditioned on "said" or the structure of the sentence prior to "Y" in "said Y," Therefore, default PCFG cannot capture sufficent context dependencies to derive the correct parse. Note that the second-order Markov annotation corrected both of these errors.

A similar phenomenon is observed with parsing adverbs using the default annotation. For example, in the sentence "The largest, Suburban Propane, was already owned by Quantum," the default grammar parses "already" as an adverb modifying "was" instead of the correct "owned." In the sentence "The finger-pointing has already begun" the default grammar parses "already" as an adjective phrase modifying "begun" instead of an adverb phrase modifying "has." On the other hand, equipped with richer tags and grammar rules, the second-order annotation correctly attaches the adverb in both of these sentences.

The richer PCFG generated by second-order vertical Markovization achieved an as-expected 5% increase over the default annotation. Generally, parses produced using the second-order markov annotation were at least as good as (and often much better than) the parses produced using the default annotation. Namely, nearly all of the sentences that that default grammar parsed accurately were also parsed correctly by the second-order grammar. Additionally, the second-order grammar also corrected many of the local errors that occured frequently in the default case (like those mentioned above). However, this second-order grammar still made several categorical errors.

The second-order model consistently makes errors in prepositional attachment. In the sentence "Most of the stock selling pressure came from Wall Street Professionals, including computer-guided program traders," the prepositional phrase "including...traders" should be attached to "Wall Street Professionals." However, the second-order model incorrectly attaches it to the verb "came."

Additionally, in the sentence "The centers normally are closed for the weekend," the second-order model incorrectly labels "closed ... weekend" as a VP rather than an ADJP modifying "are." This example represents a relatively frequent error that is intrinsic to the model: the probabilities are conditioned only over POS labels, rather than additional word or semantic information. Therefore, while $VP \to VP$ might have a reasonably high probability, treating the Z in "are Z" as a VP is incorrect regardless of the probabilities associated with the VP tag for "are."

As we continued to experiment with higher-order vertical Markovization third-order annotations yielded a modest 1% gain over second-order in terms of F1 score, but a dramatically improved EX score. Finally, fourth-order Markovization actually decreased performance. This is likely because the rules generated by fourth-order are sufficiently sparse in the training data that it is difficult to learn accurate MLE estimates of their probabilities. Another possibility is that the naive vertical conditioning we use to construct the PCFG is not a meaningful signal beyond 3 levels in the parse tree. In both of these cases, as we employ higher-order vertical annotations, the time required to generate a parse increases at a superlinear rate. This is likely because the search space of possible parses and tags increases superlinearly as the order of the Markovization increases.

4 Further Improvements

In light of the errors discussed in the previous section, we highlight several possible avenues for improvement of the statistical parser.

The results in Section 3 clearly demonstrate the usefulness of higher-order Markov annotations. However, the observed decrease in performance as we transitioned from second to third and higher order annotation is likely caused by the sparsity of these rules in the treebank. Rather than completely abandon the idea of annotations of higher order, we postulate that better smoothing of rule probabilities and a more intelligent backoff scheme or interpolation would allow us to take advantage of frequently observed higher-order rules without sacrifing the performance of the first and second order PCFG's.

Many of the observed errors in both the default annotation and the second-order annotation might be removed with head annotation. For example, in the example given in Section 3 where the second order PCFG mis-parsed "are Z", it it likely that an annotated rule for VP_{are} would better model the correct probability distribution for expanding "are" into a VP and a ADJP modifying the VP. More broadly, head annotation allows for more context-specific and richer rules in the grammar that offer an improvement over vanilla default rules in terms of modeling probabilities. Another way to achieve this goal would be to perform more tag splitting and fine-grained labelling of the annotated data.