CS224N PA2: CKY Parser

Andrew Giel
BS Stanford 2015, Computer Science
agiel@stanford.edu

Jonathan NeCamp
BS Stanford 2015, Computer Science
jnecamp@stanford.edu

I. Introduction

Probabilistic Context-Free Grammars (PCFG) allow computational linguists to model language as a combination of symbols, words, rules, and probabilities. With these powerful grammars researchers can both generate grammatically legal sentences and parse existing sentences. This paper focuses on the act of parsing, the fruits of which are tree structures labeling constituency. In this paper we outline our implementation of the Cocke-Kasami-Younger (CKY) algorithm for PCFG parsing, analyze its effectiveness and weaknesses, detail our extension of the seminal algorithm via vertical markovization, and suggest future work.

II. Implementation

Creating an accurate PCFG parser consisted of two parts: creating a grammar composed of rules with corresponding probabilities and writing the CKY algorithm that uses these probabilities to find the best parse.

The grammar was trained via maximum-likelihood estimates performed over annotated parse trees courtesy of the Penntreebank. This training was abstracted away using the *Lexicon* class and *Grammar* class. Once determined, these grammar rules and probabilities were then used to find the best parse.

We implemented the CKY algorithm, as presented in lecture, to find the best parse according to the grammar. At a high-level, CKY is a dynamic programming algorithm that fills in a grid to find the most probable parse and backtraces to construct the corresponding parse tree. The key structure necessary to this algorithm is this grid that encapsulates constituency probabilities over non-terminals for a given position in potential parse trees. The CKY algorithm fills each position of this grid with the possible non-terminals and their accompanying probabilities. For any given position in the grid, only a small number of non-terminal symbols will be possible, as many non-terminal symbols represent parts of speech which have no intersection. Additionally, as the grid is filled, backtrace pointers are stored. Once the grid is filled, the backtrace pointers are recursively explored to construct the most probable parse tree.

III. Design Optimizations

With a brief and high-level description of the CKY algorithm complete, we will now walk through the story of our implementation decisions that ultimately resulted in an efficient and accurate parser.

We first implemented CKY in what we thought to be an "efficiency-conscious" manner, which ultimately was relatively naive as compared to our later efforts. In this first implementation, we followed the CKY algorithm described in lecture closely but with minor tweaks to improve efficiency. An example of one of these improvements is our approach to filling the grid with unary rules. We used the *getUnaryRulesByChild* interface to only grab relevant unary rules for the non-terminals in a given grid cell. This is certainly an improvement over naively iterating through all non-terminals A and B to find relevant unary rules.

An even more salient improvement we identified is during the filling of the "higher" cells in the grid which deal only with constituency and not with pre-terminal to terminal mappings. Once a split s in the span of the potential constituency has been determined, a naive approach would be to again iterate over the set of all nonterminal symbols in cubic time, once for the non-terminal on the left of a grammar rule, and twice more for the non-terminals of the right of a grammar rule (since we are dealing with binary rules). It becomes exceedingly obvious that this naive approach can be improved upon when one notes that the majority of these entries iterated over in cubic time will be impossible and therefore useless to iterate over. More formally, given a binary grammar rule in the form of $A \rightarrow BC$ where A, B, C are all non-terminals, the probability of a constituency of this form at a given position (x, y) building "upward" from lower-level constituencies L and R in the grid is going to be nonzero only if $(P(A \rightarrow BC) \neq 0 \wedge P(B \in L) \neq 0 \wedge P(C \in R) \neq 0)$. We can see that there are three potential sources for sparsity and naively iterating over all possible combinations of A, B, C will result in wasted processor time. Our implementation utilized the *Grammar* class to only iterate over possible constituencies. Our implementation, using the same notation as above, extracts all the non-zero probability constituencies from L and R, which we will denote N_L and N_R. For each of these lower-level constituencies, we find the set of binary-rules from the grammar which correspond to each respective side of the expansion. For example, from N_L, we are able to find all binary-rules of the form $A \rightarrow BC$ where $B \in N_L$. Once we have these rule sets, denoted G_L and G_B respectively, we take the intersection to determine the set of possible constituencies for this location (x, y) in the grid with a split s

$$G_{x,y,s} = G_L \cap G_R$$

Then for each member of $G_{x,y,s}$ we calculate the probability and store it within the grid. Note that this approach is many times faster than that of a naive approach. With a grammar of size g, the naive approach takes $O(g^3)$ time to complete this task. Our approach takes $O(|N_L|+|N_R|+min(|G_L|,|G_R|)) \approx O(g)$ time, as it must iterate over the non-terminal sets for each side of the split, find all rules with matching terminals on each side of the split, and then take the intersection of these rule sets. All of these component operations are in linear time in terms of the size of the grammar, thanks to the child-based indexing of the *Grammar* class and the intersection operation in the Java *Set* interface. This means that the time complexity of our parser is significantly lower than that of a naive approach, lowering a cubic time operation to a linear one. This has potentially enormous efficiency implications, especially when the grammar size g grows as a result of markovization and other techniques.

One area of our implementation, not explicitly described by the CKY algorithm, is handling of the backtrace that constructs the parse tree. To start the backtrace, we initially iterated through all nonterminals in the top-level grid cell and chose the one with the highest probability as the root of the parse tree. This resulted in decent parses, but we saw a drastic improvement in accuracy when manually finding the "ROOT" nonterminal in the top-level cell and using that as the root of the parse tree.

As for the data-structures used in our initial implementation, we used a *CounterMap* that used a string version of the grid cell index as the key to each *Counter* with the non-terminal as the key to this individual *Counter*. Such a structure allowed us to easily store probabilities, as well as iterate through all the non-terminals for a given grid-cell. Furthermore, we used a *HashMap* keyed by a *Triplet* storing the cell index and nonterminal and valued by a *Triplet* containing all the information needed to walk backwards through the best parse.

Method	Runtime	F1	Exact
Unannotated	8m38s	78.10 (R:81.06, P:75.34)	20.65
2^{nd} Markov	16m45s	82.26 (R:81.09, P:83.47)	32.26
VP-Split	17m7s	83.78 (R:84.91, P:82.69)	34.84

Final Parsing Results

After completing this implementation and testing it with the treebank, we obtained an $F1$ of 78.10. Although astonishingly accurate, our first implementation attempt was slow – requiring over 3 hours to parse the 155 test sentences of 20 words or less. Thus, we moved on to tuning our implementation to improve it's runtime efficiency.

CKY relies on all rules being binary or unary in order to ensure n^3 complexity. In order to enforce this, a lossless binarization takes place that splits nodes in the tree with ternary or more rules. One observation we made when running the parser on the *minitest* was that the binarization of tree was making unneccesary modifications to already binary rules and as a result creating intermediate 'split symbols' that were entirely unnecessary. As a result, our grammar grew. In order to prevent this phenomenon and effectively reduce the size of the grammar we changed the binarization method. After modifying binarization (keeping everything else constant), our parser ran in a much improved 50 minutes and 10 seconds over the 155 sentences, with the same average F1 score. This not only reflects the effectiveness of this particular implementation optimization but the impact the size of the grammar has on the runtime of our parser.

Although this was a great improvement, our parser was still too slow and so we continued with finding areas for tightening up our code. When printing out the non-terminals stored in the top-level cell we noticed we were storing over 6,000 symbols. From this, we found that during initialization of bottom-level cells with preterminals, we were adding all preterminals in the lexicon. This included preterminal tags that had 0 probability of being paired with the correspond terminal word. Adding this condition before storing the probability in our *CounterMap* reduced the number of symbols in the final grid cell and dropped the running to 39 minutes 15 seconds.

Optimizing our code further by storing repeatedly fetched values and strings in persistent variables, we dropped this time to 34 minutes and 58 seconds.

After executing the code with *-Xprof* flag, we noticed a large percentage of the running time was spent putting values into a *HashMap* (80%). Since both our grid *CounterMap* and backtrace information are backed by a *HashMap* and both were keyed by cell index and nonterminal, we decided to combine both of these structures into one.

We chose to utilize a class that we built ourselves called *CKYGrid*. This class is implemented by using a *HashMap* of *HashMap*s, with the index in the grid as the first key, and the non-terminal symbol as the second key. This allowed us to cut down our memory usage by assuming any combination of (x, y) and a non-terminal symbol that was absent from the *CKYGrid* had a probability of 0.0. Additionally, this allowed us to very easily access all of the non-zero probability non-terminal symbols for a given location in the grid by accessing the key set of the *HashMap*. Note that we could have simply used a *HashMap* with key (x, y, n) where n is the non-terminal and this would have cut slightly lowered our memory usage compared to our implementation, but would have severely limited our ability to iterate over the set of non-terminal symbols for a given position (x, y). As such, we feel that this usage of the *HashMap* is both memory-efficient and non-limiting in the types of operations we relied on in our initial implementation of CKY algorithm.

As mentioned earlier, a large percentage of our operations were *HashMap.put*. Additionally, our backtrace and grid structures were indexed by the same keys. Thus, combing these structures allowed us to half the number of *HashMap.put* calls we make, and subsequently makes our implementation of CKY parsing very efficient and fast. With this last big change to our implementation we dropped the total running time to 8 minutes and 38 seconds while maintaining the same F1 score.

IV. Vertical Markovization

With an efficient parser, we decided to work on improving its accuracy. One area for consdiration are the independence assumptions made on the PCFG, namely that the material in a given node is independent of the surrounding material. In an effort to weaken this assumption and improve the parser's accuracy, we added 2^{nd} order vertical markovization. 2^{nd} order vertical markovization weakens the independence assumption by providing information to a given node about the parent label. This is implemented by concatenating a given node's label with the parent label before training. By performing this annotation prior to binarization and then training the grammar, we got a 4.16 increase in F1. This shows that by providing more information about the parent of a given node to the parser, we do in fact weaken the independence assumptions and as a result increase the accuracy. As a downside, adding more annotations increases the size of our grammar which has a palpable effect on the run-time – with 2^{nd} order vertical markovization it takes nearly double the time.

V. VP-Split (Extra Credit)

Note: This can be run with the flag:

-parser cs224n.assignment.ExtraCreditParser

As with the motivation behind 2^{nd} order vertical markovization, VP-Split is a way of weakening the independence assumptions by providing more information to certain nodes. More specifically, VP tags are annotated with the associated verb preterminal tag found when descending to its children. By providing extra information about the type of verb at the base of the tree (e.g. gerund, infinitive, finite, etc), the parser can more accurately predict a better grammar rule at higher levels. Verb phrases are at the heart of the meaning of a sentence, and supplying the parser with additional information in order to correctly encapsulate verb phrases in particular can supplement the parse of the overall sentence. With VP-Split annotations, we saw a 1.52 increase in F1. Once again, the more detailed annotations increased the size of the grammar and increased the total run-time.

VI. Analysis

Overall, our implementation of the CKY Parser performed very well. Even without additional annotation of nonterminals, our CKY Parser was able to achieve 78% F1, perfectly parsing more than one in five. As expected, with further annotation in the form of 2^{nd} order vertical Markovization and VP-Split our F1 scores rose to 82% and 83.75% respectively. These annotation changes reflect the information made available to the parser when we weaken the independence assumptions of Context Free Grammars which have been shown to be problematic. It should be noted that although these annotations improve the the effectiveness of the algorithm, they result in much larger grammars and subsequently result in a larger search space and slower running times. Nonetheless, VP-Split demonstrated remarkable performance, most notably achieving perfect parses on nearly 35% of the test sentences.

Although it may be fun to celebrate the effectiveness of our parser, analysis of the errors made will illuminate the weaknesses of our implementation and areas for future improvement and research.

One interesting example that demonstrates some of the flaws of our parser is the sentence: *But now prices have nose-dived and Quantum's profit is plummeting*. Below, both our parse (Guess) and the actual parse (Gold) are given in Penntreebank form.

One of the most interesting aspects of this parse from the perspective of CKY is the conjunction *and* in the middle of the sentence. We can see that the correct parse at the highest level is in the form S → CC ADVP S CC S while our parse gives S → CC ADVP NP VP. Notice that both of these parses require an expansion that is much larger than binary, one quarternary and one quinary. This poses problems for the CKY algorithm, as binary or less rules are required to ensure cubic runtime. As such, binarization occurs.

```
Guess:
(ROOT
  (S (CC But)
    (ADVP (RB now))
    (NP (NNS prices))
    (VP (VBP have)
      (SBAR
        (S
          (NP
            (NP (NNP nose-dived)
              (CC and)
              (NNP Quantum) (POS 's))
            (NN profit))
          (VP (VBZ is)
            (VP (VBG plummeting))))))
    (. .)))

Gold:
(ROOT
  (S (CC But)
    (ADVP (RB now))
    (S
      (NP (NNS prices))
      (VP (VBP have)
        (VP (VBN nose-dived))))
    (CC and)
    (S
      (NP
        (NP (NNP Quantum) (POS 's))
        (NN profit))
      (VP (VBZ is)
        (VP (VBG plummeting))))
    (. .)))
P: 50.00   R: 45.45   F1: 47.62   EX:  0.00
```

Even if the *Grammar* had seen sentences of similar form in training, the binarization would have naively binarized the rule starting from ADVP and working all the way over to CC. This unfortunately does not encapsulate the overall structure of the sentence, as the *and* conjunction really splits the sentence in two. A binarizer with the knowledge of the English language would be able to recognize this and create a binary rule to reflect this. Unfortunately, this is not the case and as such a number of intermediate symbols are created starting at the left of the sentence. Due to the quinary nature of this particular sentence structure and the subsequent expansion, the intermediate symbols will be very specific, with the last expansion resembling @S $\rightarrow$ _ADVP_S_CC. The longer and more specific these expansion rules become, the less likely they are to be seen and the lower their probability will be, and therefore they will be less likely to appear in parses. We can see that the length of this expansion rule is effectively caused by the conjunction *and*, and as such it is not all that surprising that our parser performed poorly on it.

Another interesting aspect of this sentence in particular is the tagging of the terminal *nose-dived*. It is likely that this particular verb was not seen at all in training, potentially causing low probabilities for its higher-up constituencies, especially since the preceding word is *have* which often is followed by nouns. This somewhat helps to explain why the phrase *have nose-dived and...* was tagged as VP $\rightarrow$ VBP SBAR. In the end, *nose-dived* was tagged as a proper noun, reflecting both the *Lexicon*'s lack of knowledge about the proper tag and the confusion caused in subsequent constituencies.

This sentence brings light to some of the weaknesses of the CKY algorithm. Binarization, a necessary preprocessing for CKY, can cause complex intermediate rules that get assigned low probabilities. Like many artificial intelligence agents, CKY is also limited by the data it was trained on and as such can perform sub-optimally when exposed to words it hasn't seen previously.

VII. Future Work

Although our results are fairly satisfactory, there are many potential areas for expansion and improvement in the context of PCFG Parsing. Thanks to the fairly robust framework the CKY algorithm allows, a lot of work is possible in the area of non-terminal annotation.

Non-terminals that give more context have been proven to increase performance. We demonstrated this via our usage of 2^{nd} order vertical Markovization and VP-Split, yet many other techniques exist that can give similar if not better results. One such example is higher order vertical Markovization, giving context not only of the parent to a non-terminal in a tree, but of the grandparent as well. Alternatively, horizontal Markovization gives context to the types of non-terminals that appear to the left of a given non-terminal. Although horizontal Markovization hasn't been shown to be very effective as a standalone annotation, it can result in substantive results when combined with others. Another form of annotation very similar to VP-Split is NP-Split, where the pre-terminal symbol for a given noun in a noun phrase node is appended to the NP node it sits below. One last possible extension of the CKY algorithm that relates to modifying the set of non-terminals is simply defining new non-terminals that are essentially symbols for functional words within English. These functional words include words such as *in*, *to*, and *for*. With these forms of non-terminals the parser remains unlexicalized yet has explicit lexical terms as symbols. All the aforementioned strategies are techniques that modify or annotate the set of non-terminals, creating larger grammars.

Another potential avenue for future study that was alluded to earlier in this paper is the possibility of an intelligent binarization algorithm. As mentioned earlier, binarization is typically naive and splits symbols arbitrarily. It can be imagined that a binarization algorithm could be developed that would selectively split on certain symbols in order to maintain the overall structure of the training parse. Informed binarization could reduce the number of complex symbols with very low probabilities. In the eyes of these authors, two approaches seem feasible. One approach could be based off non-terminals alone, preferring to split on certain symbols rather than others. Alternatively, the binarization could occur based on the depth of the tree beneath the symbol, preferring splits on nonterminals that are closer to leaves of the trees. We believe this could both lower the size of the grammar by eliminating some complex and specific rules while also mitigating the negative effect binarization has on rules with large expansions.

References

[1] Accurate Unlexicalized Parsing. Klein, Dan. Manning, Christopher.