CS 224N Assignment 2 Writeup

Angela Gong
agong@stanford.edu
Dept. of Computer Science

Allen Nie
anie@stanford.edu
Symbolic Systems Program

1 Introduction

1.1 PCFG

A probabilistic context-free grammar (PCFG) maps natural language onto the domain of context-free grammars by using a constituency grammar. The difference between the deterministic nature of a CFG and the non-determinism of language is resolved by assigning a probability to each rule applied in the parsing of a sentence.

In general, PCFG parsing involves a grammar $G = (T, N, S, R, P)$. T is a set of terminals and N is a set of non-terminals. S is the start symbol. R is a set of rules of form $X \rightarrow \gamma$, where $\gamma \in \{T, N\}$. P is a probability function that satisfies the following constraint:

$$\forall X \in N, \sum_{X \rightarrow \gamma \in R} P(X \rightarrow \gamma) = 1$$

1.2 CKY Parser

Our analysis implemented and used the CKY parser by Cocke, Kasami, and Young. Parsing algorithms generally come in two forms: top-down (goal-driven) and bottom-up (data-driven). CKY is a bottom-up dynamic programming algorithm, and is one of the most efficient with a worst-case runtime of $\Theta(n^3 \cdot |G|)$, where $|G|$ is the size of the grammar (in CNF) and n is the number of words in the sentence.

The CKY parser uses a parsing table of size n^2. The algorithm only requires a triangular half of this table. It considers each consecutive subsequence of words in the sentence and sees if there is a production rule that can generate that subsequence. Each cell within the parsing table stores a list of rules and probabilities representing how well that rule can produce the corresponding subsequence.

1.3 Implementation

1.3.1 Training

We first trained the parser and constructed a `Lexicon` of terminals and a `Grammar` containing unary and binary production rules.

1.3.2 Getting the Best Parse

To get the best parse, we ran the CKY algorithm with a few modifications (explained below), and optimizations (explained in Section 1.4).

The first step initializes the diagonal in the parsing table corresponding to words in the sentence; the related terminal rule and its probability (found in the `Lexicon`) is added to the cell. Next, we permute spans of the sentence and fill in the table a layer at a time. We replace the current rule upon finding a production rule with a higher probability.

Our parser is a slightly modified version of the CKY algorithm and also includes empty ($X \rightarrow \omega$) and unary ($X \rightarrow Y$) rules. Empty rules are incorporated using the fencepost method and expanding to an $(n+1)^2$-sized table. Unary rules are incorporated by adding an extra step at the end of each iteration which goes through each unary rule and checks if adding the rule increases the probability of its parent in that production rule. Although both add to the code complexity, the algorithmic complexity remains at $\Theta(n^3 \cdot |G|)$.

Each rule in a cell has an "origin" node indicating the rule(s) it was derived from. By starting at the front of the sentence at `ROOT`, we can follow the origin nodes to create the best parse tree.

1.4 Optimizations

Several optimizations made our algorithm more efficient, reducing our baseline runtime from 15 minutes to 7 minutes on a `corn0x` machine.

1.4.1 Avoiding Recursion

We avoid recursive calls when constructing the best parse tree and doing vertical markovization. We use a faster `Stack` and `Deque` based solution which avoids the overhead of function calls.

1.4.2 1D Arrays

The CKY algorithm allocates an $(n+1)^2$-sized table but only half the cells are used. This table extends into the third dimension by adding one layer for each nonterminal. To avoid allocating extra memory, we instead used a 1D array and a function to map 3D indices to 1D indices. Each cell

stores a RuleScores object which encapsulates a HashMap of rules and probabilities.

1.4.3 Iterating through Production Rules

Instead of iterating through all pairs of nonterminals when handling unary rules or triplets of nonterminals for binaries, we only iterate through rules that exist, avoiding unnecessary iterations.

1.4.4 Annotation Traversal

When annotating Trees, we only do a single traversal by inlining tag splitting and external/internal annotations with vertical markovization. We also modified the Tree class to provide fast sibling detection, which utilizes object equality to avoid stringification of trees.

2 Markovization

Markovization allows us to have more context about each node, achieving more accurate parses. As a result, new dependencies are created that give information on each node's breadth and depth within the tree. Results can be found in Section 3.

2.1 Horizontal Markovization (Extra Credit)

Horizontal markovization adds more information about sibling nodes, giving a better sense of the local tree structure. We implemented forgetful binarization which omits some history. For example, VP->_VBD_RB_NP_PP becomes VP->..NP_PP.

The default parser does infinite horizontal markovization ($h = \infty$). Prior to rewriting a rule, we examine the underscores '_' of the rule (generated by the infinite markovization). We replace all but the last few with '..', depending on the level of horizontal markovization desired.

2.2 Vertical Markovization

Vertical markovization adds information about parent nodes of each tree. Not only does this clarify the location of each rule, but it also refines their probabilities (since we are being more specific).

We implemented this using a Deque to avoid recursive calls and do a breadth-first iteration through the tree and append a parent label to each node. The root of the tree has no parent and nothing is appended. Also, terminal nodes do not have a parent label appended.

2.3 Unary Splits (Extra Credit)

The implementation of Unary splits are consistent with ones mentioned in Klein and Manning's paper. We implement internal unary annotation: UNARY-INTERN. Any parent with a single child is labeled as a unary tag. For example, NP->NNS becomes NP-U->NNS and labeled UNARY-INTERN.

The Penn Treebank does not differentiate between demonstratives and determiners, which differ by the number of siblings their tag DT has. So we also mark DT-DET for determiners and DT-DEM for demonstratives, and came up with a feature called UNARY-DT. We do the same external unary annotation on adverbs to separate single and compound adverbs like "as well", which is labeled as UNARY-RB in Table 3.

Tags satisfying UNARY-DT and UNARY-RB are labeled first, and the rest of tags that only have single parent will be split by UNARY-INTERN.

2.4 Tag Splits (Extra Credit)

Tag splitting provides a more nuanced way to look at localized context. In order to avoid repetition with TAG-PA, we implement IN-SPLIT to capture only its first sibling (ignoring the differences of left or right siblings). We then implemented %-SPLIT because both $ and % are recognized as NN or NNS, but have a different distribution than normal nouns. The Penn Treebank training/testing section is WSJ, which uses those two signs quite frequently. CC-SPLIT is done for special conjunctions such as & and (Bb)ut.

At last, through observing the Penn Treebank, and error analysis, we noticed that some tags get expanded to more than two rules, especially S or PP. In order to statistically differentiate those tags from their more traditional usages, we implemented a new split called NUM-SPLIT. This feature was not in Klein and Manning's paper.

2.5 Yield Splits (Extra Credit)

We implemented yield splits in terms of VP-SPLIT and NP-POSS. The Penn Treebank does not differentiate infinite and finite verbs. Infinite verbs are included in the compounds of a verb phrase, normally associated with another verb, such as "he likes to hunt". We mark the verb phrase "likes to hunt" with VBINF. Finite verbs take NP and are marked with VBF.

Another yield split is the possessive noun. The possessive NP has a very different structure compared to other NPs, which are more flat. By observing errors in the sample, we marked all NPs that contain POS with NP-POS.

	Horizontal Order			
Vertical	$h=0$	$h=1$	$h=2$	$h=\infty$
$v=1$	77.41	79.67	79.16	78.46
$v=2$	82.90	83.60	82.81	82.50
$v=3$	82.63	84.24	84.71	83.96

Table 1: F1 score after applying different combinations of horizontal and vertical markovizations.

	Horizontal Order			
Vertical	$h=0$	$h=1$	$h=2$	$h=\infty$
$v=1$	735 (1m30)	3557 (3m16)	7915 (3m42)	14984 (4m36)
$v=2$	4457 (2m45)	11250 (6m11)	18968 (8m47)	28088 (9m58)
$v=3$	9894 (5m48)	20247 (11m40)	29586 (14m48)	38166 (16m30)

Table 2: Grammar size and (run time) after applying different combinations of horizontal and vertical markovizations.

3 Experiment

We used the provided training trees from the `treebank` set. These were not binarized, so we did that first, then annotated the trees with the `TreeAnnotations` helper class. We also applied markovization (discussed in Section 2).

We measured the performance of our parser and different markovizations by running it on the `treebank` data. The resulting F1 score is recorded in Table 1, and the grammar size and runtime are recorded in Table 2.

We then measured the performance of our parser on various annotation techniques using a new baseline model with h=1 and v=2, because it gives us the best time/F1 trade-off.

We submitted our code with h=1 and v=2. The other markovizations can be found commented out in our `TreeAnnotations` class.

4 Evaluation

4.1 Markovization

We noticed that our original baseline model (h=∞, v=1), without markovization, exceeds the F1 score from Manning and Klein's paper. This is most likely due to our larger data size: Klein and Manning used 20 sections (2-21) of `treebank` with 393 sentences, while our model was trained on 2000 sections.

Furthermore, we noticed that increasing the horizontal order from h=1 to h=2 sometimes decreased the F1 score. This is because this change results in a twofold increase in the grammar size, which greatly dilutes the probabilities for each rule. However, increasing the vertical order does increase the F1 score, at the expense of a twofold increase in runtime.

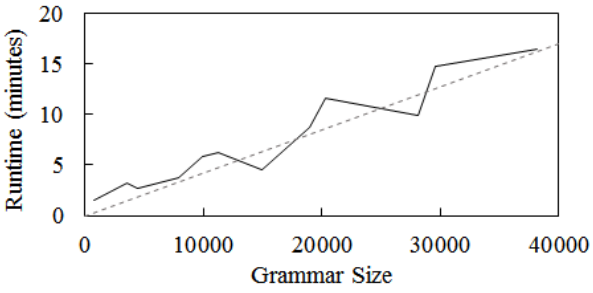

Figure 1: The grammar size changes as we change the order of vertical and horizontal markovization.

In Figure 1, we notice the almost-linear relationship between grammar size and runtime. This is as expected since the complexity of this algorithm is $\Theta(n^3 \cdot |G|)$: $|G|$ increases and n remains constant, resulting in a linear relationship.

4.2 Splitting

From Klein and Manning's paper, we already know `TAG-PA` will improve the baseline, as can be seen in Table 1; second-order markovization and `TAG-PA` increased F1 by 5%.

However, we also observed patterns that are very different from the paper. In the Klein and Manning paper, splitting techniques appear to be additive: every technique increases the baseline model's performance. The cumulative performance increases as well when combined. This was not the case for us. This could be the result due to a small and irregular testing sample, or it could have more implications, as we attempt to explain in following paragraphs.

4.2.1 Unary Tagging

`UNARY-INTERN` brought a 0.47 increase from the baseline. When combined with `UNARY-RB`, the F1 score increased by 0.54 over the baseline. However, `UNARY-DT` did not help at all. And independently `UNARY-DT` has the lowest contribution to the F1 score amongst other techniques.

Annotation	F1	Δ F1	Ind. Δ F1
Baseline ($v = 2, h = 1$) with TAG-PA	83.60	-	-
UNARY-INTERN	84.07	0.47	0.47
UNARY-DT	84.07	0.00	0.05
UNARY-RB	84.61	1.01	0.42
IN-SPLIT	84.39	-0.21	0.31
%-SPLIT	85.06	0.45	0.37
CC-SPLIT	85.03	-0.03	0.32
VP-SPLIT	84.91	-0.15	0.43
NP-POSS	**85.25**	0.23	0.00
NUM-SPLIT	85.00	-0.09	**0.55**

Table 3: Ind.Δ F1 is the F1 score increase over the baseline. Δ F1 calculates the difference between positively increased examples. Negative splits are removed from further cumulative calculations.

4.2.2 Tag Splitting

IN-SPLIT and CC-SPLIT improve F1 if used alone, but seem counter-productive when used in conjunction with unary splits. %-SPLIT improves F1 significantly by itself, and when used in conjunction with other splits, increases F1 more than just by itself. We argue that it is because the training corpus is from the Wall Street Journal, and finance/business-oriented news sources like WSJ contain more percentage symbols than other corpuses. Thus, differentiating percentage signs from normal nouns gives us a huge increase. Since %-SPLIT is a symbol tagging technique, such synergy is very difficult to explain.

4.2.3 Yield Tagging

NP-POSS splitting does not bring an F1 increase but it has synergy with the rest of the contributing techniques. VP-SPLIT works very well individually but becomes counter-productive when combined with other techniques.

In the end, our custom technique, NUM-SPLIT, gives the best results. It brings the largest F1 increase when applied independently on baseline model, but decreases F1 when used with other rules. We argue it's because NUM-SPLIT creates too many split tags when combined with other rules and cannot obtain enough statistics from the size of our training corpus.

Our best result is an F1 score of **85.25**. It's still not the 87.04 F1 score obtained from Manning and Klein's paper. This is because our implementations of splitting are different, and certain splits like SPLIT-AUX and TMP-NP are impossible to implement by us, since we do not have temporal or auxiliary tags from the training corpus.

5 Error Analysis

We do error analysis on our best run using v=2, h=1 with annotations UNARY-INTERN, UNARY-DT, UNARY-RB, %-SPLIT, NP-POSS. We examined sentences from test examples that yield the lowest F1 scores.

5.1 Resolved Errors

We first analyze errors produced by our baseline model (without splitting and markovization) and subsequently solved by our best parser.

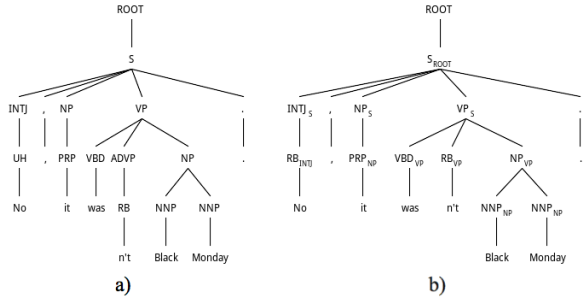

Figure 2: An error resolved by vertical markovization and UNARY-RB. (a) Incorrect parse by baseline model without markovization (b) correct vertical markovization parse. UNARY-RB also resolves this error.

As is shown by Figure 2, many parsing errors from the baseline model are of the context-ignorant nature. A plain RB tag has a higher chance of being expanded from ADVP. However, after vertical markovization, n't has a higher chance of being associated with a verb, thus the tag RB^VP is a more likely candidate for n't, leading to this particular RB tag preferring a VP as its immediate parent.

Distribution of certain tags are changed. ADVP used to prefer being attached to VP; now it is correctly attached to S: VP->ADVP VBN is gone and S->NP ADVP VP can be correctly expanded.

Horizontal markovization also helped solve certain problems. The rule NP->NP , PP gets chosen instead of VP->PP , PP. A horizontally markovized grammar would have the rightmost PP tagged as PP_,_NP, ensuring that NP is generated as the leftmost child.

5.2 Unresolved Errors

There are still unresolved parsing errors, even with annotations and higher-order markovizations.

We noticed words in the front of the sentence tend to be marked as NP and nouns. There are some sentences where this is not the case, but our parser incorrectly tags them. For example, in Figure 3, the NP produces an NNP "Heavy" instead of a JJ, which is much more common and makes more sense. This is because the CKY algorithm works by multiplying probabilities of rules – the fewer the rules, higher the probability, since each time a probability is multiplied, it gets smaller.

Figure 4 shows a sentence ending with the adverbial phrase "right now". Such sentences are rare; sentences ending with SBAR are more common. Statistically, SBAR is attached to VP more often than to NP, so our parser chooses to expand VP->VBZ SBAR rather than VP->VBZ NP ADVP and NP->NP SBAR. Since our parser uses probabilistic CFG, it always chooses the rule with the higher probability, but in this case, results in an incorrect parse.

Since CKY algorithm is bottom-up parser, it prefers building smaller phrases to reduce time complexity. This leads to local (eager) rather than global (delayed) expansions. For example the phrase "Paris and $3,200" is grouped with "and" as their conjunction, instead of the more semantically correct grouping "$2,400 to Paris and $3,200 to London".

Finally, part of our errors stems from our lack of training data. There are sentence structures that are so uncommon that our parser would never guess them unless we had a larger training set.

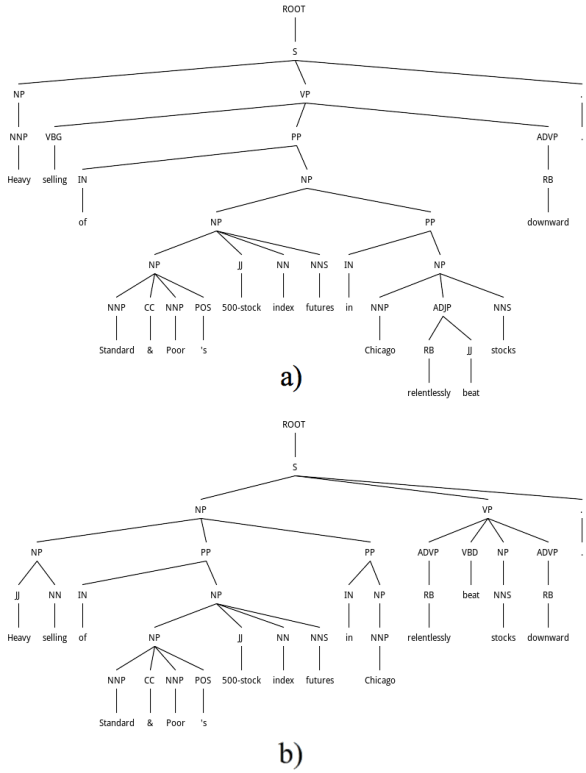

Figure 3: An error produced by our parser. (a) Incorrect parse and (b) correct (gold) parse.

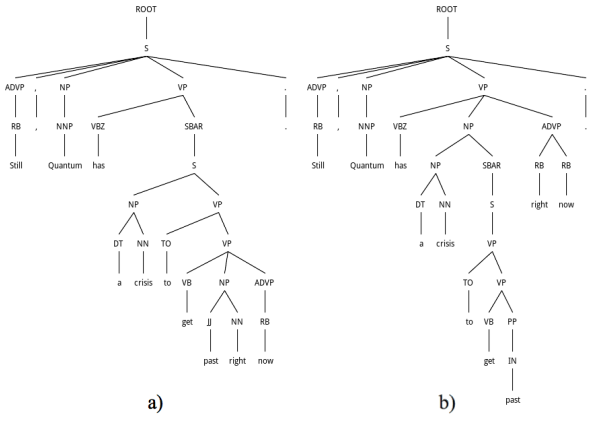

Figure 4: An error produced by our parser. (a) Incorrect parse and (b) correct (gold) parse.

6 Future Work

There are a few things we can do to further improve our model and F1 score. We did not implement them due to time constraints and difficulty.

6.1 Latent Annotations Learning

As discussed in the video lectures and in a paper by Petrov and Klein, instead of doing manual feature engineering on tag splitting, a machine-learning-driven approach using forward-backward-driven HMM to properly group (split) tags should further improve parser accuracy.

6.2 Variable History Models

As discussed in "Accurate Unlexicalized Parsing" by Klein and Manning (2004), using variable history models (e.g. $h \leq 2$ instead of $h = 2$) can improve the F1 score. This works by not removing horizontal history and adding parent history if a rule is too rare. This would fix some of the inaccuracies caused by the creation of new but rare rules when using higher-order horizontal and vertical markovizatons.