Parser

Mahsa Shafaei
User on codaLab: random_1
mahsa.shafaei@gmail.com

1 Introduction

Parsers are programs to break a sentence into its constituents and extract grammatical structure of the sentence, for example, which word is a subject or which word is an object in the sentence. Some of the most popular usages of parsers are grammar checking programs, semantic analysis and question answering applications (Jurafsky and Martin, 2014). We can categorize parsing methods into two groups. Top-down and bottom-up methods. Top-down parser begins with the start symbol "S", then it picks a production and tries to match the input. This approach only searches for trees that can be an answer for the sentence, but it also produces trees that do not match with input words. In contrary, bottom-up parsers start with the leaves and end with start symbol which only produces parse trees that match with the words, but it also contains trees that are not correct.

One of the main problem in parsers is structural ambiguity. It happens when the grammar can produce more than one parse tree for a sentence. For example, consider the sentence "I shot an elephant in my pajamas". Although semantically it is not correct, a parser can produce two syntactically correct parse trees for this sentence. Dynamic programming is a solution to address this problem. It builds a table containing the solution of all subproblem to solve the whole problem. In parsing methods, subproblems are parse trees for all constituents in the input. One of the most widely used dynamic-programming algorithms for parsing is CKY algorithm. In this study, we implement CKY algorithm with a bottom-up approach. However, CKY algorithm cannot detect the right answer among all produced trees. To solve the ambiguity problem, we add probabilistic approach to CKY to build a most probable parse tree for an input sentence. This method is named PCKY algorithm.

In this project, we use the Penn Treebank (PTB) dataset which is composed of Wall Street Journal (WSJ) collection. We use this dataset to train our rules. Using extracted rules and CKY algorithm, we produce parse trees for test set sentences. The best accuracy we accomplished is 83.26% for tagging accuracy and F-measure of 59.17% for bracketing. Figure 1 shows the flowchart of the project.

2 Methodology

We start with extracting rules from our train dataset to feed the CKY module. Then we Implement CKY and PCKY algorithms, and finally, we implement a recursive function to print parse trees for all input sentences. We also need a post-processing step to revert trees to be consistent with CFG grammars.

2.1 Extracting Rules

In the training dataset, we have parse trees sentences. So, we read each tree and extract rules for that tree with the help of open and close parenthesis. For example, (S(NP I)(VP(VP (V shot) (NP (Det an) (N elephant)))(PP (P in) (NP (Det my) (N pajamas))))) is what we have as an input, and we extract rules such as S→NP VP from this input because parenthesis show that NP and VP are at the same level and are children of "S". These rules are Context-free grammar rules, but for CKY we need the rules in Chomsky Normal Form. To change a CFG rule to a CNF one, we need to apply three steps. First, delete unit productions. For example, if A→B and B→CD, we should delete A→ B and produce A→CD. Second, we need to change rules that have terminal and nonterminal on their right side to rules with only non-terminals on the right side. For example, A→B'c' is replaced by A→BC and C→'c'. Third, we change all rules with more than

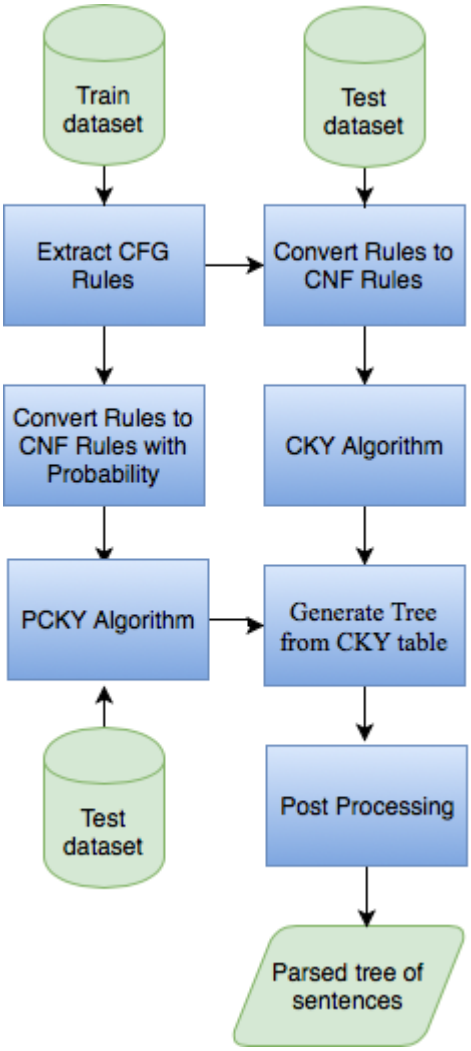

Figure 1: System Flow Chart

three non-terminals words on their right side to rules with two nonterminal words e.g. A→BCD changes to A→B_CD and B_C→BC.

2.2 CKY Parser

As we mentioned, CKY algorithm is a bottom-up parsing technique. This method works by filling a 2D matrix. At the first step, it starts with the words in the sentence and finds non-terminal variables that produce these words. Tags of each word can be retrieved from each cell in the diagonal of the matrix. In the second step, we need to fill other cells of the matrix from left to right and bottom to top. Each cell [i,j] in the matrix represents all non-terminals that produce all constituents from i to j. This algorithm is a recognizer which recognizes possible non-terminal for the parse tree. However, for the parsing, we need to keep track of non-terminal. For example, if we want to add VP in a cell [1,3], we should save VP is a parent of V in the cell [1,2] and NP in the cell [2,3]. Also, Each cell may contain more than one version of each symbol e.g. VP→VP PP and VP→V NP. As a result, a CKY parser produces all possible parse trees for each sentence, and it causes computational complexity in large datasets. The solution to this problem is that in each cell we should only keep one version of each non-terminal. We use two strategies to select. First one is simply keeping the first occurrence of each variable and second using probability model which is a PCKY algorithm.

2.3 PCKY Parser

PCKY is a statistical parsing algorithm. Everything in PCKY is same as CKY except that PCKY uses a probability model to select a non-terminal variable for each cell. Every time we want to fill a cell [i,j] with a variable we should calculate the multiplication of three probabilities. First, the probability of non-terminal variable in the cell [i,j] produces two variables in cells [i,k] and [k,j]. Second, the probability of having the first variable in the cell[i,k]. Third, the probability of having the second variable in the cell [k,j]. For example, we want to fill cell [1,4] and we have V in [1,3], PP in [4,4]. Assume, one option is VP for the cell [1,4] (according to the rule set). we calculate the probability of VP→V PP * table[1,3,V] * table[3,4,PP]. If this value is greater than last value for VP in the cell [1,4] we should replace it with the new one. otherwise, we keep it as it was. As a result, each cell may contain different non-terminal variables, but one version of each variable that has the most probability to occur. For PCKY, we need to have CNF rules with their probabilities to calculate the aforementioned formula. in the following subsection, we explain how we calculate these probabilities.

2.3.1 Probabilistic CNF

The first thing that we need in order to calculate the probability of rules is the number of repetition of each rule. So, Firs, we generate repetitive CFG rules and count the number of each rule. Then, we change CFG to CNF and assign a corresponding count number to it. In the last step, we divide the count of rules to count of the left side of the rules and consider it as the probability. For example, for rule A→B, probability is count(A→B)/count(A).

2.4 Extract Tree from Parse Table

We filled the parse table with information we need to keep track of all trees. So, for printing a parse tree we start from the most top right cell which has the start tag. Then, we find two children of this tag and continue this tracking in a recursive function until we reach to a terminal cell. By terminal cell, we mean a cell that contains part of speech tag of the input words.

2.5 Post Processing

As we mentioned before, CKY algorithm works with context-free grammar rules in Chomsky Normal Form. So, we converted our rules to CNF and then built a parse tree for sentences. As a result, we have parse trees in CNF format. To revert these trees to CFG ones we need to do two steps. First, we need to eliminate dummy variable we built, for instance, to change rule A→BCD, we defined dummy variable B_C. Now, we need to delete this variable and shift its children to its place. This task is a trivial task as we only need to ignore this variable and continue our tracking. But the second step which is reverting "unit productions" needs more calculation. For example, in our CFG rules, we had A→B and B→'c'. In CNF format we deleted A→B and instead, we inserted A→'c'. So, we need to keep all unit productions in CFG rules to retrieve CFG format of rules. In some cases, we may have more than one option to revert the unit production, in these cases, we use probability and choose the most probable rule.

3 Experimental Results

3.1 Training

We trained our model on the whole WSJ dataset and extracted 9427 CFG rules. After converting them to CNF, 3916032 rules were generated.

3.2 Testing

For the first part of the project, we implemented CKY algorithm. So, we needed a heuristic way to select one tree among all trees. We simply chose the first created tree. Using no statistic, we cannot get good accuracy. Details of the result are shown in the table 1.

For the second part of the project, we implemented the PCKY algorithm. For the baseline model, we ignored the post-processing step. This had a serious effect on both tagging accuracy and bracketing F-measure. Tagging is influenced a lot because without post processing most of the words have non terminal tags which are tags above the part of speech tags. For example we reduced S→N, N→'table' to S→'table', so the correct POS tag is "N" not "S". Also the trees are shorter than the actual trees because they do not include unit productions. The accuracy of the result is reported in the table 2.

Bracket F1	Tagging Accuracy	Average
34.88 %	58.5 %	46.69 %

Table 1: Parsing Accuracy for CKY with Heuristic Method

Bracket F1	Tagging Accuracy	Average
41.95 %	59.5%	50.7 %

Table 2: Parsing Accuracy for PCKY without Post-processing

In the next step, we added post-processing step. This function helps us to revert trees in CFG format. It means retrieving correct part of speech tags for words and correct connections. The accuracy is shown in the table 3.

Bracket F1	Tagging Accuracy	Average
59.17 %	83.26%	71.22 %

Table 3: Parsing Accuracy for PCKY with Post-processing

4 Conclusion

Parsers are tools to extract syntactical structure of the sentences. In this work, we used Penn Tree-Bank dataset to learn CFG rules, and then we applied these rules in the PCKY parser to parse the input sentences. The result is strongly depended on the probability of the rules. So, with careful rule extraction, we can reach a good accuracy.

References

Dan Jurafsky and James H Martin. 2014. *Speech and language processing*, volume 3. Pearson London:.