Probabilistic Cocke–Younger–Kasami (CKY) Parser

An approach to parsing context-free grammars

Ekal Golas

CS 6320 Natural Language Processing (Spring 2016)

The University of Texas at Dallas

Richardson, TX

exg140230@utdallas.edu

*Abstract*—This document describes an approach to probabilistic parsing of context-free grammars by using CKY algorithm and discusses some of the experiments and observations performed. Also some shortcomings of this approach and potential improvement are discussed.

*Index Terms*—CKY, Context-free, probabilistic-parsing, CYK, cocke-younger-kasami *(key words)*

# Introduction

Grammar theory to model symbol strings originated from work in computational linguistics aiming to understand the structure of natural languages. Probabilistic context free grammars (PCFGs) have been applied in probabilistic modeling of RNA structures almost 40 years after they were introduced in computational linguistics. PCFGs extend context-free grammars similar to how hidden Markov models extend regular grammars. Each production is assigned a probability. The probability of a derivation (parse) is the product of the probabilities of the productions used in that derivation. These probabilities can be viewed as parameters of the model, and for large problems it is convenient to learn these parameters via machine learning. A probabilistic grammar's validity is constrained by context of its training dataset. PCFGs have application in areas as diverse as natural language processing to the study the structure of RNA molecules and design of programming languages. Designing efficient PCFGs has to weigh factors of scalability and generality. Issues such as grammar ambiguity must be resolved. The grammar design affects results accuracy. Grammar parsing algorithms have various time and memory requirements. [1]

In computer science, the Cocke–Younger–Kasami algorithm (alternatively called CYK, or CKY) is a parsing algorithm for context-free grammars, named after its inventors, John Cocke, Daniel Younger and Tadao Kasami. It employs bottom-up parsing and dynamic programming. The standard version of CYK operates only on context-free grammars given in Chomsky normal form (CNF). However, any context-free grammar may be transformed to a CNF grammar expressing the same language. The importance of the CYK algorithm stems from its high efficiency in certain situations. Using Landau symbols, the worst case running time of CYK is O(n^3.|G|), where n is the length of the parsed string and |G| is the size of the CNF grammar G. This makes it one of the most efficient parsing algorithms in terms of worst-case asymptotic complexity, although other algorithms exist with better average running time in many practical scenarios. [2]

# Algorithm

For this paper, the standard version of the parser was implemented, which means it was fed a context-free grammar in CNF form. The pseudo-code for the CKY parser is given as:-



Fig. 1: Algorithm for standard version of CKY parsing [3]

# data source

The following sources were considered for obtaining the data required to implement this project: -

* Penn Treebank – For the corpus. To obtain this, an openly available NLTK library in python was used. [4] Its treebank package was parsed by a python script to generate the grammar from the corpus in CNF form.
* A PCFG file – This is the file obtained from the running the python script on the NLTK library. It contains a grammar that comprises of 25000 rules which map most of the common words in the English language. This is taken as the input by the program that implements CKY parser.
* A test PCFG file – To test the working of the parser, a small grammar file in CNF form was used. This was obtained from the Midterm examination paper of Natural Language Processing class at University of Texas at Dallas in Spring 2016. The grammar converted in CNF form is all follows: -

S -> VP [1.0]

VP -> Verb [0.1]

VP -> Copula Adjective [0.2]

VP -> Verb NP [0.5]

NP -> DT Noun [1.0]

DT -> the [1.0]

VP -> VP Adverb [0.2]

Verb -> 'is' [0.5]

Verb -> 'shoots' [0.5]

Copula -> 'is' [0.8]

Copula -> 'seems' [0.2]

Adjective -> 'well' [0.5]

Adjective -> 'unwell' [0.5]

Adverb -> 'well' [0.5]

Adverb -> 'badly' [0.5]

Noun -> 'duck' [0.6]

Noun -> 'well' [0.4]

This test grammar was then used to generate parse trees for sentences like “is well well” and “shoots the duck well well well”. The correct parse trees for both these sentences are given as: -

Parse tree for the sentence: is well well

(S (VP (VP (COPULA is) (ADJECTIVE well)) (ADVERB well)))

Sentence Probability: 0.008000000000000002

Parse tree for the sentence: shoots the duck well well well

(S (VP (VP (VP (VP (VERB shoots) (NP (DT the) (NOUN duck))) (ADVERB well)) (ADVERB well)) (ADVERB well)))

Sentence Probability: 1.5000000000000001E-4

The above solutions were used as a validation for the correctness of the parser implemented.

# implementation details

This section describe the input expected to the project, the design choices, the expected results and the choice of various tools and languages.

## Input Expected

As there are two phases to the implementation of this project: one the python script to generate the grammar, and second to parse the generated grammar to generate parse trees for a given sentence. So, the first phase expects python and NLTK library to be installed into the system where the python script has to be run. This program detects the installation of NLTK in the system as the input and generates the treebank grammar in CNF form.

In the next phase, the generated grammar is fed to the program that implements the CKY parser, along with a sentence. The rules in the grammar are parsed and the most probable parse tree for the given sentence is displayed to the user.

## Design choices

A third party library, Apache Command Line Interface (CLI) [5] was used to validate the arguments provided to the CKY parser.

A separate class was defined to parse the grammar as a list of rules.

A class was defined to create a concept of a probability matrix, essential for computation and comparison

A class was defined to create a concept of the backpointer matrix, along with a helper class to define the pointers that a particular element points to previous elements in the matrix. This is essential to generating the parse tree itself.

The two phases of the project were kept independent of each other, so that any context-free grammar in CNF form can be parsed and used to generate parsed trees to the sentence.

If a sentence is provided to the parser which is either invalid or unrecognized by the grammar, the parser returns a user-friendly message specifying the same. Hence, the program also performs the operation of a recognizer for a sentence with a given grammar.

## Expected Results

The following results are obtained out of the project: -

* Treebank grammar in CNF form from the python script as a PCFG file that can be parsed by the CKY parser.
* Parse tree for the sentence given by the user as input, if the sentence is recognized by the grammar provided to the CKY parser
* User-friendly error message, if the sentence given by the user as input is not recognized by the grammar provided.

## Other Implementation details

Languages: Python 2.7 (For generation of treebank grammar in CNF form), Java 8 (For CKY parser)

Environment configuration: 6GB RAM, Intel Core i3 processor with operating system as Ubuntu 15.10

Size of grammar file – 915 KB

IDE used – Gedit (For Python), Eclipse Mars .2 (For Java)

Time taken for implementation – 20 hours

# Experiments and observations

This section discusses about the various parameters of the project and experiments conducted on them and observations made along the process.

## Parameters

The following parameters were observed on the experiments conducted.

### Running time of the parser – In milliseconds.

### Size of the grammar – In KB.

* + 1. Length of the sentence provided by the user – In number of words (All strings separated by space).
    2. Running time – Compared to valid and invalid cases

## Experiments

Firstly, the test PCFG file was used to validate the working of the CKY parser. Once the parser was thoroughly tested, the bigger PCFG file was used to conduct experiments by varying each of the parameters discussed above and recording the observations made for each experiment.

Experiments were made in the following categories: -

* Running time was noted as size of the grammar was varied
* Running time was noted as the length of the sentence was varied
* Accuracy and recognition percent was noted with the size of the grammar and the length of the sentences
* Sentence probability was noted as length of sentence was varied

## Observations

With the experiments performed as discussed, the following observations were made: -

Performance compared to size of grammar

| Grammar | Runtime (In Milliseconds) | | |
| --- | --- | --- | --- |
|  | Size (In KB) | No. of runs | Run Time |
| Test PCFG | 1 | 5 | 47 |
| Treebank | 915 | 5 | 3583 |

For the above experiments in table 1, the same sentence of length 3 was chosen and the run times were compared. It is clear that grammars much bigger in size take more time to parse the sentence of the same length.

Performance compared to length of sentence

| Length | Runtime (In Milliseconds) | | |
| --- | --- | --- | --- |
|  | Grammar | No. of runs | Run Time |
| 3 | Test PCFG | 5 | 47 |
| 3 | Treebank | 5 | 3583 |
| 5 | Test PCFG | 5 | 58 |
| 5 | Treebank | 5 | 168838 |

From the above experiments in table 2, it can be observed that even a small increase in the length of sentence can have a huge impact on run time if the grammar is large. For grammar small in size (Few KBs) the impact on performance is often unnoticeable to the user.

Accuracy and recognition

| Grammar | Accuracy (In %) | | |
| --- | --- | --- | --- |
|  | Size (In KB) | No. of runs | Accuracy |
| Test PCFG | 1 | 50 | 14 |
| Treebank | 915 | 50 | 96 |

For the above experiments in table 3, 5 sentences specific to the test grammar were chosen and 45 random sentences in English language were chosen. It can be observed that test grammar did accurately recognize all the sentences specific to it but did poorly on recognizing the rest of random sentences. On the other hand, the treebank grammar which is larger in size did well on recognizing the random sentences but did worse than the test grammar in recognizing sentences specific to test grammar rules.

sentence probability compared to the length

| Grammar | Probability (In Range) | | |
| --- | --- | --- | --- |
|  | Size (In KB) | No. of runs | Prob. |
| Test PCFG | 1 | 5 | 10^(-3) |
| Treebank | 915 | 5 | 10^(-14) |

For the above experiments in table 4, a sentence with length as 4 and the possibility of it being unambiguous was taken (Example: This is a sentence) and the sentence probability with larger grammars was found to be much lower. So, the small the grammar, the more the certainty gap between two parse trees for the same sentence.

## Conclusions

After recording all the observations and analyzing the results, the following conclusions were made: -

* There is a tradeoff between running time and relevance of the results generated by the parser. A short grammar might not capture enough rules to generate relevant parse trees, or it may not recognize many sentences, but it will return the results quickly. On the other side, larger grammars will generate much more relevant parse trees and will recognize a much larger domain of text, but will take much more time in returning the results.
* Resolving ambiguity can be much harder with the larger grammars as the range of sentence probability is often low and many rules define the text. However, with smaller grammars, the gap between probabilities can be much larger which can resolve ambiguity much better. However, smaller grammar may still have the problem of relevance of the most probable result.
* Hence, as per different scenarios, it is wise to carefully decide the domain of the text to be parsed and make it as concise as possible and then define a grammar that taken as less rules as possible to parse the text and generate the relevant results so that the run time for parsing the text with the given grammar can be minimized without a huge compromise on relevancy.

# shortcomings and improvements

Following are some of the shortcomings and potential improvements proposed: -

* Once parsed, the grammar cannot be changed dynamically to adopt a new set of rules
* The algorithm could be modified to reject unknown words in the sentence much early in the computation phase.
* If the grammar is large, parsing takes a lot of time. Even for grammars under 1 MB, the running time gets to 20 minutes for sentences of length 6-7. A multithreaded parallel computation version of this algorithm can probably save time
* Valiant`s algorithm has asymptotically lesser running time for larger matrices. This could show some improvement in running time.
* The standard version of the algorithm can be extended to parse non-CNF context-free grammars

# Acknowledgment

I wish to acknowledge Professor for the Natural Language Processing class, Dr. Dan Moldovan for approving the project proposal and providing the guidelines for the project. Also, I acknowledge the teaching assistant for the class, Meghana Satpute, for resolution of queries and validating the demo run of the project.

# References

<https://en.wikipedia.org/wiki/Stochastic_context-free_grammar>

<https://en.wikipedia.org/wiki/CYK_algorithm>

<http://www.hlt.utdallas.edu/~moldovan/CS6320.16S/Lecture9%20LexProbParsing.pdf>

<http://www.nltk.org/>

<https://commons.apache.org/proper/commons-cli/>