CS 421: Final Project
Learning Context Free Grammars

John Doherty

May 7, 2018

1 Overview

I have chosen to do this project on Context Free Grammars because of their importance in many fields. In this class, we have learned how they are used in interpreting programming languages. These Grammars are used for constructing parsers that can implement efficient REPLs. They are also important in natural language processing where they can used to represent the grammatical structure of languages [1].

The topic of this paper is learning a Context Free Grammar from a set of positive examples, where a positive example is a string that is a member of the given language. There are a number of ways that this can be done, including using ADIOS, the Lattice based method, and the Omphalos algorithm [**?**, **?**, **?**]. In this paper, I examine the last method, which is Alexander Clark's method which he used to win the Omphalos context free grammar learning competition [1].

The organization of this paper is as follows. First I will summarize Clark's implementation of the algorithm. After summarizing the paper, I will explain my implementation of the greedy version of the algorithm. This includes my changes and the challenges I faced. I will summarize the tasks I completed as stated in the project proposal and the tasks I was unable to complete. After describing the implementation I will discuss the tests that were performed and summarize their results before concluding the paper. The code that was created for this project can be found on github at `https://www.github.com/jdoherty7/learn-cfg`.

2 Algorithm Description

Context Free Grammars are in general not learnable, however there exists a subset of them which are learnable [1]. This subset is the NTS grammars, non-terminally separated grammars, and are the class of languages which are learned in this paper [1]. They are defined in terms of constituents. A constituent is a substring which is derived from a single nonterminal [1]. The algorithm handles this case because random grammars, which were used in the competition,

will often satisfy this property. Furthermore the algorithm can be extended to included some deterministic grammars which are not NTS by relaxing some conditions and using a more conservative approach [1]. However, this increases the computational complexity of the already inefficient algorithm and thus the greedy approach is described and implemented in this paper.

The algorithm consists of two parts. The first part is extending the grammar, which involves creating new production rules and adding new nonterminal symbols to the hypothesis grammar. In order to do this, the algorithm first identifies the likely constituents in the training set. Because we assume the grammar is NTS, constituents will be the yields of nonterminals. To identify these constituents three steps are taken. First the frequency of all substrings is calculated. Substrings that occur infrequently are removed from this set of substrings to consider. Next, the mutual information of each of these substrings is calculated using a formula provided [1]. The mutual information is used because it has been shown to be a good metric for measuring if substrings are constituents [1]. All substrings with a mutual information below a tolerance are discarded. These remaining substrings are likely constituents. However, we must figure out if they are constituents of the same nonterminal or different nonterminals. This is done by constructing a substitution graph. A substitution graph is a graph where substrings are the nodes and there is an edge between nodes if both lur and lvr exist in the training set, where u and v are the substring nodes and l, v are in Σ^+. (The algorithm specifies Σ^* but this would be incorrect as then all substrings would have an edge and you would always have a complete graph.)

After constructing the graph, any node with a degree of 0 is removed, and each of the connected components are added as a production rule of a new nonterminal. If there is a single nonterminal character within the component though, then each substring in that component is added as a production rule to the present nonterminal. Finally, for every example string in the training set. If there is a string which occurs frequently and is less than a specified length then that string is added as a production rule to a new starting symbol. This concludes the extending grammar step.

The second part of the algorithm involves rewriting the strings in the training set using the newly created hypothesis grammar. This is done by going through each string in the training set and replacing it by the minimum length sequence of terminal and nonterminal symbols which is equivalent to the original string in the newly made hypothesis grammar. Thus, with each iteration, the hypothesis grammar grows and the training set shrinks until the training set can be rewritten as all the starting symbols in the hypothesis grammar [1]. The total cost of the algorithm is polynomial, however it is quite inefficient and the author mentions the runtime being anywhere from a few minutes to hours just to learn one grammar. The cost is dominated by the construction of the substitution graph which is $O(|u|^2|v|^2|T|^2)$, where T is the set of all strings in the training set.

3 Implementation

3.1 Components

There are three main components of the code. The first component is the learning algorithm. This is the algorithm described above and is used to learn a context free grammar from a set of positive examples. My implementation of the algorithm was mostly consistent with the algorithm described in the paper. One difference was that I restricted the length of possible constituents that would be considered for constructing the substitution graph. This was done in an effort to decrease the cost. However, this resulted in some substrings in larger training sets having a negative mutual information, which is impossible. This would go away by increasing the length of substrings considered. However, since smaller training sets ended up being used this was ignored in order to save in costs.

The second component is the performance testing framework. This is the code that is used to assess the performance of the learning algorithm. The performance is assessed by measuring the output grammar's ability to correctly identify strings that are in a target language that it has not seen before. The functions that do this include a function to generate random strings in a grammar, given a grammar as well as a function which checks if a string is in a given grammar. These were implemented using the Natural Language Toolkit, which provides a context free grammar class. This class includes a generate method for generating strings and a Shift Reduce parser for parsing given strings [2]. In addition to these, other auxiliary functions were made to calculate statistics from the training and convert grammars from my implementation to grammar objects in the Natural Language Toolkit.

The third component is the unit testing framework. The main purpose of this part of the code is for debugging. Unit tests were made with good coverage of most auxiliary functions. This ensured that when functions were changed their behavior was not. This allowed for faster debugging and ensures that if future changes are made that the code remains usable. Comprehensive documentation was also added to ensure this usability.

3.2 Status

All functions that are necessary for functionality are written and they pass the necessary unit tests. The main algorithm also mostly succeeds at identifying the likely constituents from a target grammar and will output a hypothesis grammar. However, it does not generalize as well as stated in the paper. This is a result of three problems the maximum substring length, the small size of training sets, and the hyperparameters.

The maximum substring length was a problem because of its effect on mutual information as stated previously as well as the limits on the size of constituents it could find. Practically, this was not much of a problem in the smaller training sets and helped manage high runtimes, but is not ideal theoretically because of the limits it imposes on constituent identification and its affect on larger training

sets. Furthermore, while smaller training sets had to be used for a similar reason, to limit the runtime. They also limited the ability of the algorithm to generalize for larger grammars.

The hyperparameters used included the frequency and mutual information cutoffs that are used to construct the substitution graph as well as the frequency and length cutoff used to determine a starting symbol. The optimal values are very dependent on the size of the example set. However, since the values used in the paper were not published, I was unable to accurately compare to the paper's implementation and had to rely on heuristics for setting my own.

In the original project proposal, the stated goals were to implement the greedy algorithm described in Clark's paper and quantitatively test its ability to learn random and handmade grammars. A qualitative analysis would then be done on the grammars learned from examples of Python, Haskell and English. Because of the high runtimes and sensitivity to hyperparameters instead only a qualitative analysis will be performed of small examples from some random and handmade grammars. This is done so larger training sets can be used, which will take longer to train on, but which result in a higher quality learned grammar.

4 Tests

4.1 Design of the Test

The tests performed were all qualitative because of the high runtimes of the algorithm. Two handmade grammars were used to generate positive examples and grammars were learned from these examples. The rules were then compared to the target grammar. Additionally, strings from Python, Haskell, and the English Language were used as training sets and the learned grammars were examined.

4.2 Expectations

In the paper, the algorithm was capable of identifying many of the random grammars given in the competition and did end up winning the competition [1]. It was even able to exactly identify some of these grammars [1]. The ones that it could not identify, it typically could not because there were not enough examples given to differentiate between two hypothesized grammars [1]. Slightly smaller training sets are used compared to the ones in tha paper and it is unknown how the hyperparameters compare. Thus, one would expect this implementation to generalize more poorly than the paper's implementation. However, one would still expect this implementation to identify likely constituents and even generalize well if the grammar is small enough and such a grammar will be examined.

When learning from the coding and NLP sets, if the algorithm is able to generalize well then one would expect it to learn some understanding of the structure of the examples it is looking at. Thus for these sets I expect to see these patterns appear in the productions of the learned grammar. For example

since if statements are included in the sets, for the Haskell part I may see a "if e1 then e2 else e3" production, while in Python I may see a "if b:\n\t e1 \n else: \n\t e" production. For the English section, I expect to see productions that resemble the grammatical structure of the English language, like "Subject Verb Predicate", or productions that match with words.

4.3 Results

4.3.1 Handmade Grammar, Small Grammar

When the grammar is small and the training set is comparatively large note that the algorithm is capable of inferring the correct grammar. In this specific case it infers the target grammar exactly.

Target Grammar:

$$NT0 \rightarrow aNT0 \mid NT1$$
$$NT1 \rightarrow b$$

Small Training Set :

$$T = [aab, aaaaab, aaaab, aaab, b, ab]$$

Learned Grammar - Exactly the Target Grammar:

$$NT1 \rightarrow aa \mid aaa$$
$$NT2 \rightarrow b$$
$$NT3 \rightarrow NT1b$$

4.3.2 Handmade Grammar, Larger Grammar

When the grammar becomes slightly larger you can see how with a smaller training set the correct grammar may not be induced. By using more data this can be fixed. Note that since this is the greedy version of the algorithm there are many redundant rules in both grammars.

Target Grammar:

$$NT0 \rightarrow aNT0 \mid cNT1$$
$$NT1 \rightarrow bc$$

Small Training Set:

$$T = [aaaacbc, acbc, cbc, aaacbc, aaaaacbc]$$

Learned Grammar - Small: Note that this does not allow many a's to be created, allowing only a finite number

$$NT1 \rightarrow acb \mid baaa \mid aacb \mid aaaa \mid cb \mid a \mid aa$$
$$NT2 \rightarrow aac \mid ac \mid c \mid aaac$$
$$NT3 \rightarrow cbc$$
$$NT4 \rightarrow NT1c$$

Large Training Set:

$$T = [aaacbc, aaaaaaaaaacbc, aaaaaaacbc, aaaacbc,$$
$$aaaaaaaaaaaacbc, aaaaacbc, aaaaaacbc, aaaaaaaacbc,$$
$$aaaaaaaaacbc, aaaaaaaaaaacbc, cbc, acbc, aacbc]$$

Learned Grammar - Large: Bolded are the productions which equal the target grammar's.

$$NT1 \rightarrow aac \mid aaaac \mid \mathbf{c} \mid ac \mid aaNT1 \mid aaaNT1 \mid \mathbf{aNT1}$$
$$NT2 \rightarrow cbc$$
$$NT3 \rightarrow aNT1b \mid aaNT1b$$
$$NT4 \rightarrow \mathbf{NT1bc}$$

4.3.3 Python, Haskell, English

Python Training Set:

$$T = [\text{"a=4"},$$
$$\text{"b=7"},$$
$$\text{"c=a*b"},$$
$$\text{"print(c)"},$$
$$\text{"def f():\textbackslash n\textbackslash tprint('hi')"},$$
$$\text{"def g():\textbackslash n\textbackslash treturn 5"},$$
$$\text{"if c==4:\textbackslash n\textbackslash tc=c+3\textbackslash nelse:\textbackslash n\textbackslash tprint(b)"},$$
$$\text{"if a>4:\textbackslash n\textbackslash ta=2:\textbackslash n\textbackslash tc=3"}]$$

Haskell Training Set:

$$
\begin{aligned}
T = [&\text{"let a=3 in a*5"},\\
&\text{"if a==5 then b else (a+7)"},\\
&\text{"let d=7 in d+5"},\\
&\text{"if d>7 then 4 else (a-7)"},\\
&\text{"let m=7 in v*8+1"},\\
&\text{"fac 0 = 1\nfac n = n*(fac (n-1))"},\\
&\text{"inc x = x+1"},\\
&\text{"f 0 = 0\nf a = f (a-1)"}]
\end{aligned}
$$

English Training Set:

$$T = [\text{" This "}, \text{" That "}, \text{"I am a man"}, \text{"I am a woman"}]$$

The generated productions for the Python examples were successful at identifying the "print" statement, the "\n\t" symbols, and part of the start of a function. However, it failed to identify arithmetic operations well or the expression in functions or statements.

The Haskell example set took 4 hours to run one iteration of making a hypothesis grammar before it took up too much memory and the process had to be terminated. In that time, it did not identify much of meaning. Though one production did include only the "+" and "-" symbols and other contained the symbol "in ". The reason why it took so long was because the size of the training set was too large.

The grammar leaned for the English training set was more finicky. Using similar hyperparameters to the other sets, the productions did not mean much. However, by twiddling with the hyperparameters, productions could be generated which contained words like "This" and "That". This further illustrates that the importance of the parameters for generalization and their dependence on the training set.

5 Conclusion

In conclusion, I have demonstrated an implementation of Alexander Clark's method for context free grammar induction. While the algorithm leaves much to be desired in terms of computational complexity, it does show itself to be capable of learning grammars given enough data. However, this stipulation limits it in many settings, such as the examined cases of programming and natural languages. However, speed was not the only thing limiting its performance in these cases, as it was shown that generalization can be dependent on the chosen hyperparameters. Thus, while the algorithm is a good place to start in learning how to learn Context Free Grammars, it may be worth considering newer algorithms for this task.

6 Listing

The code used in this project can be found at `https://www.github.com/jdoherty7/learn-cfg`.

References

[1] Clark, A. Learning deterministic context free grammars: The omphalos competition. *Machine Learning 66*, 1 (2007), 93–110.

[2] Loper, E., and Bird, S. Nltk: The natural language toolkit. In *Proceedings of the ACL-02 Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics - Volume 1* (Stroudsburg, PA, USA, 2002), ETMTNLP '02, Association for Computational Linguistics, pp. 63–70.