The Pumping Inference Algorithm: Exploiting the Pumping Lemma for CFGs to Learn PCFGs*

Henrique Barros Lopes HENRIQUE.LOPES@IFMG.EDU.BR
Computing Department
Federal Institute of Education, Science and Technology
Ibirité, MG 32407-190, Brazil

Abstract

The probabilistic grammar inference is a problem that, given an input string dataset, aims to find a rule set and its probabilities that better produce it. It is applied in Bioinformatics, Computer Music, Computational Linguistics, Architecture, and Software Engineering, to name a few. Developing grammar learning methods from a dataset is crucial and can benefit all these research areas. For probabilistic context-free grammar (PCFG) inference, we noticed the lack of algorithms capable of finding context-free grammar structures with only positive data, which is probably an undecidable problem. Hence, researchers choose to use probability estimation algorithms to learn it. This paper proposes a new algorithm based on the pumping lemma for Context-Free Languages. This algorithm learns grammar's structure and achieves higher prediction accuracy averages than the Gibbs Sampling algorithm in learning the Dyck-n languages and in the languages with $\|a\| = \|b\|$ and $\|2a\| = \|b\|$.

Keywords: Grammatical Inference, Probabilistic Context-Free Grammar, Pumping Lemma, Machine Learning

1 Introduction

The probabilistic grammar inference is applied in Bioinformatics, Computer Music, Architecture, and Software Engineering, to name a few. Works inferring grammars are used, for example, to validate protocols (Nouri et al., 2014; Mao et al., 2012) and to infer flow diagrams (Breuker et al., 2016). Another field that uses probabilistic grammatical inference is computational linguistics (Heinz and Rogers, 2010; Jarosz, 2013, 2015b,a), where the inferred grammars learn relationships between letters in a word or words in a phrase.

Developing grammar learning methods from a dataset is crucial and can benefit all the aforementioned fields. Currently, there are several algorithms, including the well-known ALERGIA (Carrasco and Oncina, 1994; de la Higuera, 2010), that yield promising results in inferring probabilistic finite automata (PFAs). In Section 2, we review how these algorithms can attempt to estimate rule probabilities or find a grammar structure representing the learned language.

However, for Probabilistic Context-Free Grammar (PCFG) inference, we noticed the lack of algorithms capable of finding Context-Free Grammar (CFG) structures with only positive data, that is, without knowing which strings do not belong to the inferred language. Most works try to infer probabilities for a set of all possible production rules and remove

1. This work had institutional support from the Federal Institute of Education, Science and Technology (IFMG).

the almost zero probability rules (Shibata, 2021; Tsushima et al., 2018). As the PCFG probability estimation is probably an NP-hard problem, this approach can only achieve local-optimum solutions (de la Higuera, 2010, Chapter 17.7). So, we decided to go on the probability estimation's opposite side and investigated how an intrinsic Context-Free Language feature, the pumping lemma, can aid in the PCFG inference.

Drawing inspiration from the pumping lemma for CFGs, we introduce an algorithm called Pumping Inference (PI) designed to learn Probabilistic Context-Free Grammars (PCFGs) for Context-Free Languages (CFLs). This method can infer languages such as Dyck-n language, palindromes language, or the language with equal numbers of 0s and 1s. We describe this algorithm, which is the main contribution of this paper, in Section 3.

We perform experiments in Section 4 using the PI and the Gibbs Sampling (GS) (Johnson et al., 2007) algorithms to infer 4 different Dyck-n languages and the languages with $\|a\| = \|b\|$ and $\|2a\| = \|b\|$. Results show our approach outperforms the GS in all languages inference, except the Dyck-n languages with $n = 3$, and $n = 4$. We conclude our work in Section 5, highlighting this paper's main contribution, the PI algorithm, which has a novel approach using the pumping lemma to find grammar structures. Later, we present potential future research. The source code of this work is available at author's github page[1].

2 Preliminaries

We can see the problem of inferring grammar in 3 different ways. We attempt to learn a language, a state machine that accepts these language's strings or the grammar itself. All of them lead to equivalent models. These languages belong to different language classes according to Chomsky's hierarchy (Chomsky, 1956). All of them can have their probabilistic version.

2.1 Probabilistic Context-Free Grammar

We define a Probabilistic Context-Free Grammar (PCFG) as a 5-tuple (N, T, R, S, θ) where:

1. N is the set of non-terminals;
2. T is the set of terminals;
3. R is the set of production rules;
4. $S \in N$ is the initial symbol;
5. θ is a probability vector such that $\|\theta\|$ accounts for the number of rules.

Each rule $r \in R$ takes the form $A \to \alpha$, where $A \in N$, and $\alpha = [\alpha_1, \dots] : \alpha_i \in N \cup T$.

The context-free class of languages represents string sets with more complex constraints than regular languages. Because of these constraints, we cannot build a Deterministic Finite Automaton (DFA) that accepts a CFL. We need a more complex finite state machine, a pushdown automaton. On the other hand, we can define this latter state machine to accept any regular language. We can define a Context-Free Grammar (CFG) yielding all languages

1. https://github.com/henriqueblopes/grammatical-inferences

that a DFA accepts. We need to make the set N equal to the state set Q in a DFA. Furthermore, we need to make the set R contain rules of the form $A \Rightarrow sB$ where $s \in T$, and A, $B \in N$, and for each $e \in F$, R contains rules of the form $e \rightarrow \lambda$, where λ is the empty string.

The inference of a CFG from positive data is harder to perform than the inference of a DFA or a PFA. The existing algorithms only learn a subclass of CFG (Scicluna, 2012), or they need some additional data like the derivation tree skeletons (Oates et al., 2002).

In most cases, researchers infer PCFGs by estimating probabilities. The idea is to fix the grammar's number of non-terminals, build all possible rules in the Chomsky Normal Form (Tsushima et al., 2018; Johnson et al., 2007), and calculate the probabilities with some hill-climb algorithm. An algorithm that performs this is the inside-outside (Lari and Young, 1990). The main problem with these approaches is the computational cost involving the inside table calculation. There were also attempts to infer the context-sensitivity probabilities of a CFG with a non-parametric Bayesian approach that avoids calculating the inside table (Shibata, 2021). Although the work mentions context-sensitivity probabilities, the author states that the expressiveness of this grammar class is equivalent to the CFG class.

There is a lack of algorithms in the literature that attempt to find the grammar structure instead of the probabilities of all possible rules. The issue arises from the necessity of knowing the grammar structure in certain applications (de la Higuera, 2010, Chapter 17.7). To the best of our knowledge, researchers have never looked to the pumping lemma for CFLs to infer PCFGs. The pumping lemma shows how to create greater derivations from smaller ones by "pumping" the first non-terminal that appears on the considered derivation. So, it is possible to think about a solution that uses the pumping lemma for CFLs to build an algorithm to infer PCFGs.

2.2 The Pumping Lemma

The pumping lemma is a result that proves some languages are not regular or context-free (lemma 1). The pumping lemma for CFLs states that we can pump two substrings simultaneously, and there could be a small substring separating these two (Hopcroft et al., 2001, Chapter 7.2).

Lemma 1 (The pumping lemma for CFLs) *Let L be a CFL. So, exists a constant n such that if z is any string in L such that $\|z\| \geq n$, then we can write $z = uvwxy$, subject to the following conditions:*

1. $\|vwx\| \leqslant n$
2. $vx \neq \lambda$
3. $\forall k \geq 0$, *the string* uv^kwx^ky *is also in* L

The theorem's proof uses the pigeonhole principle and assumes a CFG in the Chomsky Normal Form. In PCFG's case, a non-terminal that derives v and x pumps any number of times these substrings. So, this non-terminal appears twice in the uv^2wx^2y derivation tree, three times in uv^3wx^3y, or it does not appear, generating the string uwy. However,

the main point is the absence of algorithms that exploit the pumping lemma and attempt to find substrings v and x to infer a PCFG. In Section 3, we will present an algorithm to perform this inference task. The starting point to work with this algorithm is to generate a Frequency Prefix Tree Acceptor (FPTA) with a set of strings from a CFL.

2.3 Frequency Prefix Tree Acceptor (FPTA)

Many algorithms start from an FPTA to infer grammar structures because it contains information about strings' prefixes (de la Higuera, 2010, Chapter 16.3). An FPTA is a tree-like DFA keeping all the prefixes in the dataset and the number of uses each prefix has.

An FPTA is also a DFA. So, we can define it in a CFG-like form as mentioned in Section 2.1. The PI algorithm uses this CFG data structure to build an FPTA and performs the grammatical inference task.

3 The Pumping Inference Algorithm

This approach hypothesizes that an FPTA has all information we need to learn a PCFG from the pumping lemma. So, if the FPTA accepts the $uvwxy$, it must also have strings with v and x pumped as final states. We need to find the following data about the FPTA's non-terminals to understand the Pumping Inference (PI) algorithm:

1. a concatenation list for each non-terminal;
2. if a non-terminal nt pumps substrings v and x;
3. (v, x) pumped by nt for each pair (w, z),
4. the final states in the FPTA obtained by pumping (v,x) any number of times;
5. the number of times (v, x) is pumped;
6. identify Pumping Rules;
7. identify Concatenation Rules.

We list the definitions to understand the PI algorithm in Section 3.1. We use the concatenation compatible Test with a breadth search (Section 3.2) to address item 1. Section 3.3 shows how to obtain items 2, 3, 4, and 5. Finally, Section 3.4 performs items 6 and 7 and joins all items in our PI algorithm.

3.1 Definitions

- the *prefix* function applied to a non-terminal nt returns the prefix an FPTA reads until reaches nt. So, *prefix*(nt_{04}) in Figure 1 is the string ();
- a non-terminal is final when it is a final state in the FPTA. *prefix*(nt) when nt is final, returns a string in the dataset;
- the *frequency* function applied to a list of final non-terminals returns the sum of the number of times *prefix*(nt) appears in the dataset, for all nt in the list;

- the $\|\|$ returns the length of a string or list. $\|(())\|$ is 4;
- the *height* function returns a non-terminal's height in the FPTA;
- the symbols [ ] is used to denote lists. And we can access the list's ith element with list[i], where $0 \leq i \leq \|\text{list}-1\|$;
- the symbol : denotes an operator that returns a sub-list. list[1:$\|$list-2$\|$] returns a sub-list without the first and the last element of list;
- a rule is a pair (nt,RHS[ ]), where nt is a non-terminal, and RHS[ ] is a list of right-hand sides;
- a right-hand side (RHS) is a pair (α, θ), where $\alpha = [\alpha_1, \dots] : \alpha_i \in N \cup T$ and θ is its frequency in the FPTA (Section 2.1);
- a tuple is an aggregation of two or more data structures. A pair is a tuple;
- the operator "." (dot) access a tuple component;
- the *children* function applied to a non-terminal returns a list of non-terminals extracted from its RHS[ ]. Given a rule $nt_1 \rightarrow$ r $nt_2|$ s $nt_3|$ t nt_4, *children*(nt_1) returns the list $\{nt_2, nt_3, nt_4\}$;
- the *front* function returns the first element of a queue;
- finally, the *dequeue* function removes the element returned by the front function.

3.2 Concatenation Lists and Concatenation Rules

Concatenation, a common operation in both CFLs and Regular languages, involves rules in the format $nt \rightarrow \zeta\ nt$, where ζ represents any terminal sequence and nt is a non-terminal. It means that, after ζ, nt yields nt again and can yield any other sequence on a different nt's RHS, including $\zeta\ \zeta\ nt$. That is, we concatenate the prefix ζ with nt. We call it a Concatenation Rule (CR).

To determine concatenation compatibility between non-terminals, we propose a test that checks if a non-terminal's subtree contains another non-terminal's subtree. Algorithm 1 outlines the steps to perform this test. The idea is to perform a synchronized breadth search in the FPTA, verifying if the first non-terminal contains all the second non-terminal's RHS[ ]. If it does, we check the same for its children following the breadth search order.

Algorithm 1 returns **true** if nt_1's subtree contains nt_2's subtree, conditioned on nt_2's subtree having a minimum height. It means nt_1's subtree is starting again, and there could be a CR with format $nt_1 \rightarrow \zeta\ nt_1$, where ζ is the terminal sequence yielded from nt_1 to nt_2.

To build the concatenation list for each non-terminal, we perform a breadth search again and assume that a non-terminal can only be concatenation compatible with an ancestor non-terminal in β.

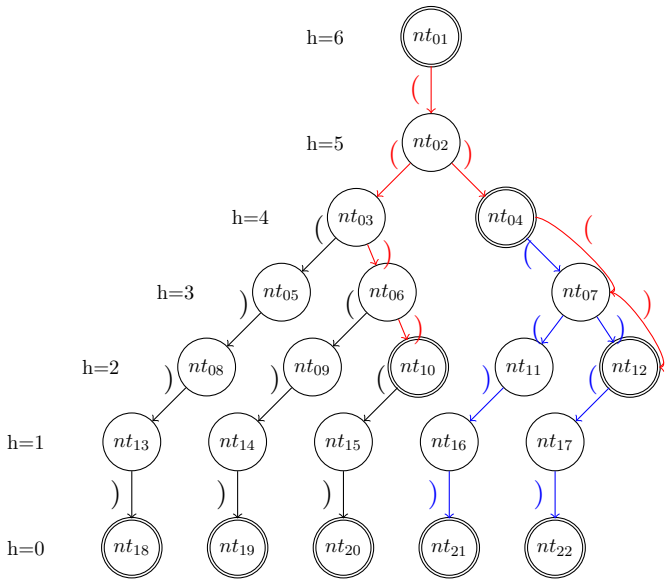


Figure 1: FPTA for Dyck-1 language with each non-terminal's height, two compatible non-terminals and their subtrees.

Algorithm 1 concatenation_compatible

Require: FPTA β, nt_1, nt_2, min_h
Ensure: boolean

1: **if** $height(nt2) <$ min_h OR (nt_1 is final AND nt_2 is not final) OR (nt_2 is final AND nt_1 is not final) **then**
2: return **false**
3: **end if**
4: q_$nt_1 \Leftarrow \{nt_1\}$
5: q_$nt_2 \Leftarrow \{nt_2\}$
6: **while** q_$nt_2 \neq \emptyset$ **do**
7: **if** $\exists s|$ $front$(q_nt_2) $\rightarrow s\gamma$, $front$(q_nt_1)$\nrightarrow s\delta$ **then**
8: return **false**
9: **end if**
10: q_$nt_2 \Leftarrow$ q_$nt_2 \cup$ $children(front$(q_nt_2))
11: q_$nt_1 \Leftarrow$ q_$nt_1 \cup$ $children(front$(q_nt_1))
12: $dequeue$(q_nt_2)
13: $dequeue$(q_nt_1)
14: **end while**
15: return **true**

3.3 Pumping Rules

According to the pumping lemma, if $uvwxy$ belongs to a language, then for all k, uv^kwx^ky also belongs to the language (Section 2.2). Figure 1 shows an FPTA for the Dyck-1 language, where we have final non-terminals in the prefixes () and (())). Note that a Dyck language's rule has the format $nt \rightarrow (\ nt\ )$, which simultaneously pumps the substrings (and). Our idea is, for a given non-terminal in the FPTA, to search for possible rules with this format that generate strings uv^kwx^ky with different values of k. If the FPTA contains enough of these strings, we assume the language has a rule of the form $nt \rightarrow v\ nt\ x$. We define this as a Pumping Rule (PR).

Algorithm 2 find_$vwxy$

```
Require: nt, β, nt_f, r
Ensure: set_vwxy
 1: y ⇐ λ
 2: path ⇐ prefix(nt_f)
 3: while ‖path‖ > 1 do
 4:     count_s_p ⇐ count_s ⇐ p_uses ⇐ 0
 5:     finals ⇐ ∅
 6:     for ‖v‖ ⇐ 1 to ‖path‖ do
 7:         for ‖x‖ ⇐ 0 to ‖path‖ − 1 do
 8:             (v,w,x) ⇐ (path[0 : ‖v‖], path[‖v‖ + 1 : ‖path‖ − ‖x‖], path[‖path‖ − ‖x‖ + 1 : ‖path‖ − 1)
 9:             for k ⇐ 2, ‖wy‖ + k ∗ ‖vx‖ <height(nt), k ⇐ k + 1 do
10:                 count_s ⇐ count_s +1
11:                 if nt →* v^k w x^k y then
12:                     p_uses ⇐ p_uses + k∗frequency(f)
13:                     count_s_p ⇐ count_s_p +1
14:                     finals ⇐ finals ∪ f
15:                 end if
16:             end for
17:             if count_s_p > 2 AND count_s_p / count_s > r then
18:                 set_vwxy ⇐ set_vwxy ∪ (v, w, x, y, finals, p_uses)
19:             end if
20:         end for
21:     end for
22:     y ⇐ path[‖path‖ − 1] y
23:     path ⇐ path[0 : ‖path‖ − 2]
24: end while
```

The Algorithm 2 shows how we can extract PRs from the FPTA β for a given non-terminal nt. The input variable nt_f is a final non-terminal. The variable r is a ratio we calculate to consider if we accept or do not accept a PR.

We start by initializing the substring y with the empty string λ and path with *prefix*(nt_f). Then, we begin a loop while *path*'s size is not 1. The goal is to identify all vwx combinations that yield enough uv^kwx^ky. For this, we iterate at v's size, varying it from 1 to *path*'s size, and at x varying from 0 to $\|path\| - 1$. The size of vwx is equals to $\|\text{path}\|$, so w's size is $\|\text{path}\| - \|vx\|$. We define substrings v, w, and x by spliting *path* according to $\|v\|$ and $\|x\|$ values in each iteration (line 8).

For each vwx combination, we start a loop iterating k from 2 until uv^kwx^ky's size becomes higher than *height*(nt). If this condition does not hold, then uv^kwx^ky's size is higher than any string in β, so we must stop the search. For each k, we increment 1 unit in count_s, a variable storing the number of possible strings obtained by pumping v and x. If nt yields the suffix v^kwx^ky in β (line 11), then we increment p_uses with the frequency of the non-terminal f multiplied by k, increment count_s_p, which stores the obtained strings present in β, and insert f in the set finals. f is the final non-terminal in β after yielding v^kwx^ky from nt.

After these findings, we test if we found more than 2 strings in β and if the portion count_s_p in count_s is higher than r. If it holds, we create the tuple (v,w,x,y, finals, p_uses) and insert it in the ordered set set_$vwxy$, the algorithm's output. The p_uses component's value gives its order. Finally, at lines 22 and 23, y receives path's last symbol concatenated with itself, and we remove the last symbol from path. We repeat this process until path's size reaches 1. In the end, we return a set of tuples containing each substring $vwxy$ from the pumping lemma associated with each string it generates and the number of times it pumps v and x. This set has all the information we need to build the PRs.

3.4 Finishing the PI Algorithm

The Algorithm 3 outlines the steps to run the PI. It receives as input a string dataset D, a minimum height min_h for the concatenation compatible test, and the ratio r between the number of strings in the FPTA and the number that the FPTA should have. The first thing we do is build the FPTA β (Mao et al., 2016) and the concat_lists with the Algorithm 1. Then, for each non-terminal $nt \in \beta$ and each final non-terminal nt_f, where nt is an ancestor of nt_f, we use Algorithm 2 to find the set set_$vwxy$ of PRs candidates for a given nt. For each tuple in the set_$vwxy$, with different wy, we create a new non-terminal ntp and a new rule r with an empty RHS and ntp at its left-hand side.

Algorithm 3 PI

Require: D, min_h, r
Ensure: PCFG β
1: $\beta \Leftarrow$ FPTA(D)
2: concat_lists $\Leftarrow$ build_concatenation_lists(β)
3: **for** $nt \in \beta$ **do**
4: **for** $nt_f \in \beta$, where nt is ancestor of nt_f **do**
5: set_$vwxy$ $\Leftarrow$ set_$vwxy$ $\cup$ find_$vwxy$(nt, β, nt_f, r)
6: **end for**
7: **for all** $wy \in$ set_$vwxy$ **do**
8: r $\Leftarrow$ ($ntp, \emptyset$)
9: **for all** $vx \in$ set_$vwxy$ with same (w, y) **do**
10: r.RHS $\Leftarrow$ r.RHS $\cup$ (v ntp x, vx.p_uses)
11: r.RHS $\Leftarrow$ r.RHS $\cup$ ($vx.w$, frequency(vx.finals))
12: **for all** f $\in$ vx.finals **do**
13: **for all** RHS used in $nt \xrightarrow{*} \zeta$ f **do**
14: RHS.θ $\Leftarrow$ RHS.θ - frequency(f)
15: **end for**
16: **end for**
17: β.R $\Leftarrow$ β.rules $\cup$ r
18: β.R[nt].RHS $\Leftarrow$ β.R[nt].RHS $\cup$ (ntp y, sum(r.RHS[ ].θ))
19: **end for**
20: **end for**
21: **end for**
22: insert_concat_rules(β, concat_lists)
23: return β

Then, for each element in set_$vwxy$ with a different combination of v and x, and the same w and y, we add a new RHS in rule r, where v ntp x forms the yielded string, and p_uses is the RHS's frequency. After that, we add another RHS in r yielding w, with frequency equal to the sum of all non-terminals' frequencies in the component finals.

Since we will include these rules in β's rule set (β.R), the strings represented by the component finals will count twice. One time in the PR and another in the prefixes' frequencies already present in β. So, the next step is to reduce the old frequencies in β for all prefixes from nt to all nt_f in set_$vwxy$.finals. We take the frequency of a non-terminal

representing a final string and subtract this value in all prefixes non-terminals in β (loops in lines 12 to 16).

We insert r in β's rule set and add a new RHS in the rule with left-hand side nt. This RHS yields $ntp\ y$, with frequency equal to the sum of all r's RHSs. These steps conclude the procedure for finding and creating the PRs. The last step is include the CRs using the method insert_concat_rules in line 22.

Algorithm 4 defines how to insert the CRs in β. It receives as input the compatible lists, one for each non-terminal, and β, and returns the same β with updated rules. For each non-terminal nt_2 compatible with the current nt, nt_2's RHSs become nt's RHSs. After that, for each PR p present in nt's RHS, if p yields the prefix in β that reaches nt_2 we reduce p's frequency by half and add a new rule in nt's RHS yielding $p\ nt$ with the same frequency of p. We assume we will use a CR the same number of times we use the correspondent PR. However, we could weight it by the number of times each yielded string in the dataset according to each rule's yielding capability.

Algorithm 4 insert_concat_rules

Require: compatible_lists, β
Ensure: β (updated)
1: **for all** (nt,cl) $\in$ compatible_lists **do**
2: **for all** $nt_2 \in$ cl **do**
3: nt_2.rhs $\Leftarrow$ nt.rhs
4: **end for**
5: **for all** $p \in nt$.rhs, where p a is PR **do**
6: **if** $p \rightarrow$ prefix(nt_2) **then**
7: p.frequency $\Leftarrow$ p.frequency/2
8: nt.rhs $\Leftarrow$ nt.rhs $\cup$ ($p\ nt, p$.frequency/2)
9: **end if**
10: **end for**
11: **end for**

After inserting the CRs, we finish the PI execution by returning the PCFG β with all found CRs and PRs. In Section 4, we conduct experiments with the PI algorithm to infer PCGs for CFL languages, and we start a discussion about the results and how the PI algorithm fits in the literature.

4 Results

We validated our algorithm using six different datasets, including the Dyck-n languages with lengths smaller than 10, and languages with $\|a\| = \|b\|$ and $\|2a\| = \|b\|$.

The Dyck-n languages contain strings generated by grammars of the following form: $S \rightarrow s_1Ss_2|s_3Ss_4|...|s_{2n-1}Ss_{2n}|\lambda|SS|$. They are known CFLs, subject of recent studies (Ebrahimi et al., 2020; Conrado and Pavlogiannis, 2024), and as we increase n, they become harder to infer. For each language, we created training datasets (TDs) containing all strings with lengths smaller than 9 for Dyck-n languages, and smaller than 13 for the other languages. The Dyck-n TDs have 1290, 3940, 9430, and 19302 strings, for $n = 3, 4, 5$ and 6 respectively. The other two languages TDs have 1276 ($\|a\| = \|b\|$), and 597 ($\|2a\| = \|b\|$). We created validation datasets (VDs) containing 2000 randomized strings with lengths 10 for Dyck-n languages, 14 for $\|a\| = \|b\|$ language, and 15 for $\|2a\| = \|b\|$ language.

We compared our algorithm with Johnson's Blocked Gibbs' Sampling algorithm and a random generation of strings measuring the inferred grammar's prediction accuracy (PA) (Shibata, 2021) $Score(G)$ defined in Equation 1. In Equation 1, x represents a string in

a validation set VD, and P is the probability of the grammar G to derive x. If G cannot yield x, P assumes the value of its random generation probability (RGP). We calculate the RGP by powering the value $\frac{1}{t}$ with an exponent equal to each string's size in VD, where t is the number of terminals in the dataset.

$$Score(G) = \frac{\sum_{x \in VD} log_{10} P(x)}{\|VD\|} \tag{1}$$

The algorithms' parameters were $r = 0.1$, and min_h = 3 for the PI algorithm, and we inferred grammars with 2, 3, 5, 7 and 10 non-terminals ($\|NT\|$) for the Gibbs' Sampling algorithm (GS). The prior probabilities were the uniform distribution among the rules, and we run the GS for a maximum of 1000 iterations or 24 hours, without burn-in. We used an AMD Ryzen 7 5800X 8-Core Processor 2.2 GHz with 16GB of RAM to perform this experiment. We ran 30 samples for each algorithm and parameter combination, except for the samples where $\|NT\| = 10$, where it was possible to run only 6 samples. Each execution used a single thread.

In Table 1, we show PA averages for all experiment factors and Dyck-n datasets with the highest values in bold. Figure 2 shows the average PA evolution through time for each $\|NT\|$ and a star point representing the PI Score for the Dyck-n datasets. Since the PI algorithm is deterministic, we only need 1 run for it. In Equation 2, we can see the PRs and CRs that the PI algorithm inferred in the Dyck-3 dataset. The PI found similar rules for the other Dyck-n languages. We omitted them to avoid repeating rules because the only difference is the number of terminals and RHSs. Figure 3 shows the average PA evolution through time for each $\|NT\|$ and a star point representing the PI Score for the other two languages.

Table 1: Prediction accuracy averages for Dyck-n languages inference with GS, with different value of $\|NT\|$, and PI algorithms. Highest values in bold.

	Dyck-n			
	3	4	5	6
	Pumping Inference			
	−6.480	−7.427	**-8.190**	**-8.755**
$\|NT\|$	Gibbs' Sampling			
2	−8.917	−10.245	−11.239	−12.111
3	−8.356	−9.599	−10.569	−11.362
5	−7.444	−8.874	−9.764	−10.564
7	−6.418	−8.169	−9.656	−10.068
10	**-6.053**	**-6.954**	−8.366	−9.160
	Random			
	−8.451	−9.542	−10.414	−11.139

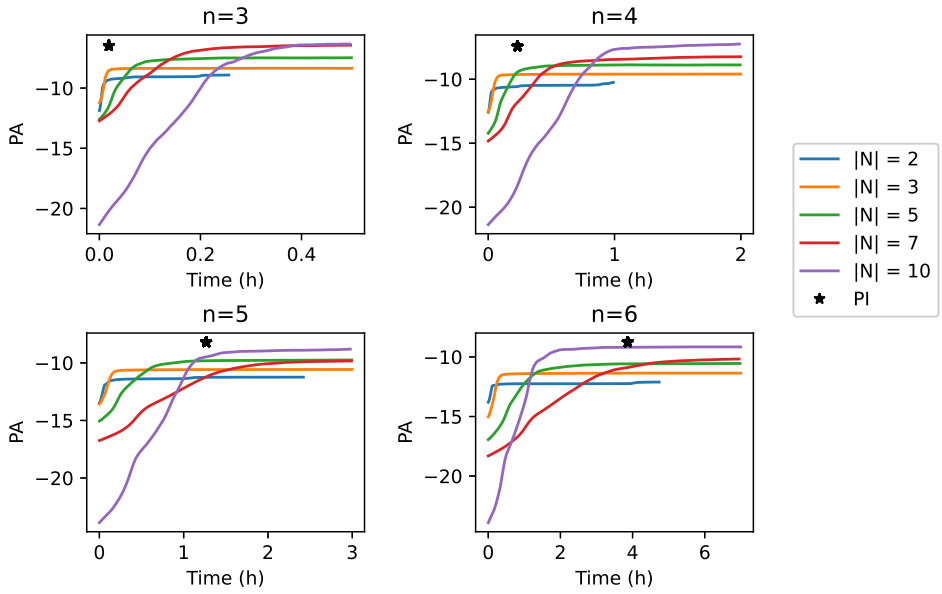


Figure 2: Results for Dyck-n languages. Prediction Accuracy averages on vertical axis and time in hours on horizontal axis for GS algorithm with $\|NT\| = 2$ (blue line), $\|NT\| = 3$ (yellow line), $\|NT\| = 5$ (green line), $\|NT\| = 7$ (red line), $\|NT\| = 10$ (purple line), and for PI algorithm (black starpoint).

$$
\begin{array}{rcl}
 & t_4\ ntp\ t_5 \mid t_4\ t_5\ ntp \mid t_2\ ntp\ t_3 \mid & \} \text{ (PRs)} \\
 & t_2\ t_3\ ntp \mid t_0\ ntp\ t_1 \mid t_0 t_1\ ntp & \\
ntp \rightarrow & t_4\ ntp\ t_5\ ntp \mid t_4\ t_5\ ntp\ ntp \mid t_2\ ntp\ t_3\ ntp \mid t_2\ t_3\ ntp\ ntp \mid & \} \text{ (CRs)} \\
 & t_0\ ntp\ t_1\ ntp \mid t_0\ t_1\ ntp\ ntp \mid & \\
 & \lambda &
\end{array}
\tag{2}
$$

Comparing the GS with the random values, the GS with $\|NT\| = 2$ and 3 performs worse than a RGP of strings. The results mean that this $\|NT\|$ can not learn the Dyck-n language rules to generalize the grammar for longer strings. The GS overcomes the RGP when $\|NT\| = 5$, 7 and 10. Because of this, we decided to test the GS only with $\|NT\| = 5$, 7 and 10 for the other two languages.

Considering the scenarios better than a RGP, the PI algorithm outperforms the GS in PA for the Dyck-n languages with $n = 5$ and $n = 6$, the most complex languages. With $n = 3$ and $n = 4$, it is outperformed by GS with $\|NT\| = 10$. Considering the other two languages, the PI outperforms the GS in all scenarios. The PI algorithm finishes its execution before GS with $\|NT\| = 5$ converges for Dyck-3 and Dyck-4, and before GS with $\|NT\| = 7$ and $\|NT\| = 10$ for all Dyck-n, except when $n = 6$ and $\|NT\| = 10$. Because of PI's structure-based approach, it finds CRs and PRs equivalent to the original Dyck-n grammar rules. This structural search makes the PI algorithm outperform the GS.

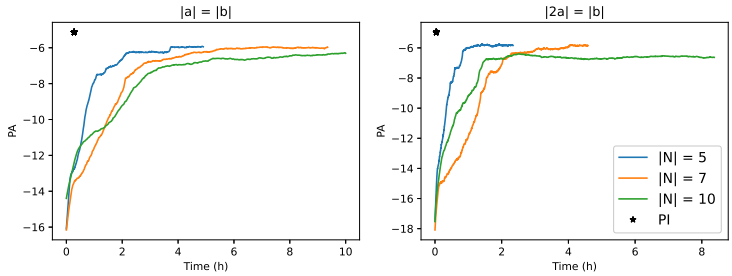


Figure 3: Results for other two language datasets. Prediction Accuracy (PA) averages on the vertical axis and time in hours on the horizontal axis for the GS algorithm with $\|NT\| = 5$ (blue line), $\|NT\| = 5$ (yellow line), $\|NT\| = 10$ (green line), and for PI algorithm (black Star-point). PI algorithm reached PA average of -5.134 and -4.960 respectively, against 5.945 and -5.839 ($\|NT\| = 5$), -5.972 and -5.851 ($\|NT\| = 7$), and, -5.813 and -6.627 ($\|NT\| = 10$)

Our findings corroborate de la Higuera (2010, Chapter 17.7) statement that the grammar structure matters for learning a PCFG.

While the GS algorithm gives non-zero probabilities for strings absent in TD, the PI algorithm gives exactly 0 if we can not yield them by the PRs and CRs. It is a weakness that is difficult to solve. The PI algorithm will need the PRs and CRs information to deliver a useful PCFG for its users. However, this issue is also a strength. Learning CFL properties is probably an undecidable problem, and PI offers a way to exploit the pumping lemma to deliver a good model. This approach works well for the tested languages, and we believe it will work well for other CFL languages.

Another problem with the PI algorithm lies in its generalization method. If we find two PRs in the same non-terminal, we assume we can combine these rules to yield other strings. However, it can lead PI to make mistakes. If we try to learn the union of two Dyck-1 languages with different alphabets, then PI will assume it is a Dyck-2 language. For instance, if we have the grammar $S \to S1|S2; S1 \to t_1 S1 t_2|\lambda; S2 \to t_3 S2 t_4$, PI will infer a Dyck-2 grammar.

5 Conclusions

We proposed a new algorithm to infer PCFGs based on the context-free pumping lemma in this paper. This new algorithm, called Pumping Inference can learn context-free pumpings of some CFLs, like the dyck-n languages. When comparing the Pumping Inference with GS algorithm, we conclude our algorithm achieves higher prediction accuracy and lower time values for almost all tested languages than GS algorithms. Our findings highlight the importance of a grammar's structure in grammatical inference. We believe PI can achieve similar results for a wide range of CFLs. Although the Pumping Inference algorithm achieved good results, we believe we could improve it to perform the learning task better. We could develop and test different kind of rule inference other PRs and CRs. Besides, there could be better ways to use the pumping lemma than Pumping Inference.

References

Dominic Breuker, Martin Matzner, Patrick Delfmann, and Jorg Becker. Comprehensible predictive models for business processes. *MIS Q.*, 40(4):1009–1034, dec 2016. ISSN 0276-7783. doi: 10.25300/MISQ/2016/40.4.10. URL `https://doi.org/10.25300/MISQ/2016/40.4.10`.

Rafael C. Carrasco and Jose Oncina. Learning stochastic regular grammars by means of a state merging method. In Rafael C. Carrasco and Jose Oncina, editors, *Grammatical Inference and Applications*, pages 139–152, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. ISBN 978-3-540-48985-6.

N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, Sep. 1956. ISSN 2168-2712. doi: 10.1109/TIT.1956.1056813.

Giovanna Kobus Conrado and Andreas Pavlogiannis. A better approximation for interleaved dyck reachability. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, pages 18–25, 2024.

Colin de la Higuera. Grammatical inference. learning automata and grammars. *Grammatical Inference: Learning Automata and Grammars*, 04 2010. doi: 10.1017/CBO9781139194655.

Javid Ebrahimi, Dhruv Gelda, and Wei Zhang. How can self-attention networks recognize Dyck-n languages? In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 4301–4306, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.384. URL `https://aclanthology.org/2020.findings-emnlp.384`.

Jeffrey Heinz and James Rogers. Estimating strictly piecewise distributions. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 886–896, Stroudsburg, PA, USA, 2010. Association for Computational Linguistics. URL `http://dl.acm.org/citation.cfm?id=1858681.1858772`.

John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. Introduction to automata theory, languages, and computation. *Acm Sigact News*, 32(1):127–131, 2001.

Gaja Jarosz. Learning with hidden structure in optimality theory and harmonic grammar: beyond robust interpretive parsing. *Phonology*, 30(1):27–71, 2013. doi: 10.1017/S0952675713000031.

Gaja Jarosz. Learning opaque and transparent interactions in harmonic serialism, 2015a.

Gaja Jarosz. Expectation driven learning of phonology*, 2015b.

Mark Johnson, Thomas Griffiths, and Sharon Goldwater. Bayesian inference for PCFGs via Markov chain Monte Carlo. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 139–146, Rochester, New York, April 2007.

Association for Computational Linguistics. URL https://www.aclweb.org/anthology/N07-1018.

K. Lari and S.J. Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech & Language*, 4(1):35–56, 1990. ISSN 0885-2308. doi: https://doi.org/10.1016/0885-2308(90)90022-X. URL https://www.sciencedirect.com/science/article/pii/088523089090022X.

Hua Mao, Yingke Chen, Manfred Jaeger, Thomas Nielsen, Kim Larsen, and Brian Nielsen. Learning markov decision processes for model checking. *Electronic Proceedings in Theoretical Computer Science*, 103, 12 2012. doi: 10.4204/EPTCS.103.6.

Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim Guldstrand Larsen, and Brian Nielsen. Learning deterministic probabilistic automata from a model checking perspective. *Machine Learning*, 105:255–299, 2016.

Ayoub Nouri, Balaji Raman, Marius Bozga, Axel Legay, and Saddek Bensalem. Faster statistical model checking by means of abstraction and learning. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, pages 340–355, Cham, 2014. Springer International Publishing. ISBN 978-3-319-11164-3.

Tim Oates, Devina Desai, and Vinay Bhat. Learning k-reversible context-free grammars from positive structural examples, 11 2002.

James Scicluna. Context-free grammar inference (master's dissertation), 2012.

Chihiro Shibata. Learning (k,l)-context-sensitive probabilistic grammars with nonparametric bayesian approach. *Machine Learning*, 2021.

Hiroaki Tsushima, Eita Nakamura, Katsutoshi Itoyama, and Kazuyoshi Yoshii. Generative statistical models with self-emergent grammar of chord sequences. *Journal of New Music Research*, 47(3):226–248, 2018. doi: 10.1080/09298215.2018.1447584. URL https://doi.org/10.1080/09298215.2018.1447584.