Universiteit Leiden
The Netherlands

Non-linear optimization methods for learning regular distributions

Chu, W.; Chen, S.; Bonsangue, M.M.; Riesco, A.; Zhang, M.

Citation

Chu, W., Chen, S., & Bonsangue, M. M. (2022). Non-linear optimization methods for learning regular distributions. *Formal Methods And Software Engineering*, 54-70. doi:10.1007/978-3-031-17244-1_4

Version:	Publisher's Version
License:	Licensed under Article 25fa Copyright Act/Law (Amendment Taverne)
Downloaded from:	https://hdl.handle.net/1887/3590001

Note: To cite this publication please use the final published version (if applicable).

Non-linear Optimization Methods for Learning Regular Distributions

Wenjing Chu[1(✉)], Shuo Chen[2], and Marcello Bonsangue[1]

[1] Leiden University, Leiden, The Netherlands
w.chu@liacs.leidenuniv.nl
[2] University of Amsterdam, Amsterdam, The Netherlands

Abstract. Probabilistic finite automata (PFA) are recognizers of regular distributions over finite strings, a model that is widely applied in speech recognition and biological systems, for example. While the underlying structure of a PFA is just that of a normal automaton, it is well known that PFA with a non-deterministic underlying structure is more powerful than deterministic one. In this paper, we concentrate on passive learning non-deterministic PFA from examples and counterexamples using a two steps procedure: first we learn the underlying structure using an algorithm for learning the underlying residual finite state automaton, then we learn the probabilities of states and transitions using three different optimization methods. We experimentally show with a set of random probabilistic finite automata that the ones learned using RFSA combined with genetic algorithm for optimizing the weight outperforms other existing methods greatly improving the distance to the automaton to be learned. We also apply our algorithm to model the behavior of an agent in a maze. Also here RFSA algorithms have better performance than existing automata learning methods and can model both positive and negative samples well.

Keywords: Probabilistic finite automata · Residual finite state automata · Learning automata · Passive learning · L_2 distance between discrete distributions

1 Introduction

Probabilistic Finite Automata (PFAs) [22] are non-deterministic automata where every state is allocated an initial and a final probability, and every transition is allocated a transition probability in addition to the alphabet symbol. PFAs are similar to Hidden Markov Models (HMM): HMMs and PFA with no final probabilities generate distributions over complete finite prefix-free sets. On the other hand, HMMs with final probabilities and probabilistic automata generate distributions over strings of finite length. In fact, a PFA can be converted into an HMM and vice-versa [12,28]. PFA and HMM are all in the same class of probabilistic models that are widely used for machine learning, such as in speech recognition [1,17,18] and biological modeling [2,13].

© Springer Nature Switzerland AG 2022
A. Riesco and M. Zhang (Eds.): ICFEM 2022, LNCS 13478, pp. 54–70, 2022.
https://doi.org/10.1007/978-3-031-17244-1_4

While nondeterministic finite automata (NFA) are equivalent to deterministic finite automata (DFA) [15], we have that PFAs are strictly more powerful than deterministic probabilistic finite automata (DPFAs) [12,16,28]. Consequently, a lot of effort has been paid to learning DPFAs from examples. The most famous algorithm is ALERGIA [6], based on merging and folding states guided only by a finite set of positive samples. In ALERGIA, the automaton's structure and probabilities are learned simultaneously. Later on, Carrasco *et al.* [6] provided a simpler version of ALERGIA, named RLIPS algorithm [7]. Ron *et al.* [24] developed an algorithm for learning only acyclic automata. As for learning full PFA, Baum *et al.* pioneered the Baum-Welch (BW) algorithm [3], which first constructs a fully connected graph and then assigns zero weight to unnecessary transitions. The approach is not practical as it has a vast number of parameters [21]. The Expectation-Maximization (EM) algorithm could learn the distribution of probabilistic automata [26]. However, how the resulting distribution could be adapted to fit into the structure of the automaton we learn is not fully clear. Another limitation of the EM algorithm is that each iteration can be slow when there are many parameters, meaning that the method can be computationally expensive [14,27].

In [9] we proposed a strategy to build a PFA using residual languages then assigning probabilities to the automaton by fairly distributing the values among the transitions. The advantage of this method is that we can learn the structure of a target automaton as an NFA. Unfortunately, the strategy used to assign probabilities is often not very effective, as in general probabilities are not fairly distributed in a PFA.

In this paper, we improve our previous work by using different strategies in learning the weight of a PFA. We focus on passive learning PFAs from examples and counterexamples and following two steps: first we learn the underlying structure of PFAs using residual languages, and then we use state of the art optimization methods to learn the probabilities labelling the states and transitions of the automaton. This boils down to defining a parametric PFA with unknown variables for probabilities that are then assigned to value by solving an appropriate optimization problem dictated by the sample and the structure of the automaton. We design two sets of experiments to compare our algorithm with flip-coin, ALERGIA, and k-testable [8]. First, we use a set of randomly generated PFAs. The results show that the numerical solution under constrained nonlinear optimization problems together with learning by residual learns automata generating a distribution very close to the target one, even in the case of non-deterministic distribution. In fact, our method based on genetic algorithm achieves improvements on existing learning algorithm up to 96%. Then we use all these algorithms to model an agent's behavior in a maze. Only the RFSA algorithm learns both positive and negative samples well. Since the target automata of traces are deterministic, RFSA with flip-coin, genetic algorithm, and sequential quadratic programming all have good performance.

2 Preliminaries

This section recalls some basic notions and facts of (probabilistic) automata and fixes the notation we use.

An alphabet Σ is a finite set of symbols and a string x over Σ is a finite sequence of alphabet symbols. We denote the empty string by λ and the set of all strings over Σ by Σ^*. A language L over Σ is a subset of Σ^*. For any language L and any string $u \in \Sigma^*$, the residual $u^{-1}L$ of a language L with respect to u is the language $u^{-1}L = \{v \in \Sigma^* | uv \in L\}$ and we call u the characterizing word of $u^{-1}L$.

A non-deterministic finite automaton (NFA) over an alphabet Σ is a tuple $A = \langle \Sigma, Q, I, F, \delta \rangle$, where Q is a finite set of states, $I : Q \to \{0, 1\}$ maps to 1 all states that are initial, $F : Q \to \{0, 1\}$ maps to 1 all states that are final, and $\delta : Q \times \Sigma \to \{0, 1\}^Q$ is a transition function. The extension δ^* of δ to strings instead of alphabet symbols is defined as usual by $\delta^*(q, \lambda)(q) = 1$ and $\delta^*(q, ax)(q'') = 1$ iff there exists q' such that $\delta(q, a)(q') = 1$ and $\delta^*(q', x)(q'') = 1$. Given an NFA A and a state $q \in Q$ the language $L(A, q)$ consist of all strings such that there exists q' such that $\delta(q, x)(q') = 1$, and $F(q') = 1$. The language $L(A)$ accepted by an NFA A is the union of all $L(A, q)$ for states q such that $I(q) = 1$. A language L is regular if it can be accepted by an NFA. An NFA A is said to be deterministic (DFA) if $I(q) = 1$ for at most one state, and for every q and a, $\delta_p(q, a)(q') = 1$ for at most one state.

Residual finite state automaton (RFSA) A is a non-deterministic automaton whose states correspond exactly to the residual languages of the language recognized by A, that is for each state $q \in Q$, there exists a string $u \in \Sigma^*$ such that $L(A, q) = u^{-1}L(A)$ [10]. In this case, every string from an initial state to q is a characteristic word for $L(A, q)$. For example, every minimal (no two states recognize the same language) and trimmed (every state is reachable from an initial one) automata is a residual automata with a finite set of characteristic strings.

A probabilistic language over Σ^* is a function $D : \Sigma^* \to [0, 1]$ that is also a discrete distribution, that is: $\sum_{x \in \Sigma^*} D(x) = 1$. An interesting class of probabilistic languages can be described by a generalization of non-deterministic automata with probabilities as weight on states and transitions. A probabilistic finite automaton (PFA) over a finite alphabet Σ is a tuple $A = \langle \Sigma, Q, I_p, F_p, \delta_p \rangle$, where: Q is a finite set of states, $I_p : Q \to (\mathbb{Q} \cap [0, 1])$ is the initial probability such that $\sum_{q \in Q} I_p(q) = 1$, $F_p : Q \to (\mathbb{Q} \cap [0, 1])$ determines the weight of the final states, and $\delta_p : Q \times \Sigma \to (\mathbb{Q} \cap [0, 1])^Q$ is the transition function such that $\forall q \in Q$,

$$F_p(q) + \sum_{a \in \Sigma, q' \in Q} \delta_p(q, a)(q') = 1.$$

We define the support of a PFA A as the NFA where the initial map, the final map and the transition function coincides with those of A when the value is 0 and otherwise maps everything else to 1. A PFA is said to be deterministic if its support is a DFA.

Given a string $x = a_1 \cdots a_n \in \Sigma^*$ of length n, an accepting (or valid) path π for x is a sequence of states $q_0 \cdots q_n$ such that: $I_p(q_0) > 0$, $\delta_p(q_i, a_{i+1})(q_{i+1}) > 0$ for all $0 \leq i < n$, and also $F_p(q_n) > 0$. We set $i_p(\pi) = I_p(q_0)$, $e_p(\pi) = F_p(q_n)$, and $\delta_p(\pi) = \Pi_{i=0}^{n-1} \delta_p(q_i, a_{i+1})(q_{i+1})$. Further, denote by $Paths_p(x)$ the set (necessarily finite) of all accepting paths for a string x. A PFA is said to be consistent if all its states appear in at least one accepting path. The probability of a path $\pi \in Paths_p(x)$ is given by $i_p(\pi) \cdot \delta_p(\pi) \cdot e_p(\pi)$, while the distribution of a string $x \in \Sigma^*$ is defined by:

$$D(A)(x) = \sum_{\pi \in Paths_p(x)} i_p(\pi) \cdot \delta_p(\pi) \cdot e_p(\pi). \tag{1}$$

If a PFA A is consistent then it is easy to show [12] that $D(A)$ is indeed a distribution on Σ^*, that is $\sum_{x \in \Sigma^*} D(A)(x) = 1$. We therefore call a distribution D on Σ^* regular if it is generated by a PFA A, that is $D = D(A)$.

The language $L(A)$ accepted by a probabilistic automaton A is the support of its distribution and is given by mapping an x to 1 if and only if $D(A)(x)$ a strictly positive. In other words, $L(A)$ is the language of the support of A. A language is regular if and only if it is accepted by a (deterministic) probabilistic finite automaton. However, differently, than for ordinary automata, the class of distributions characterized by deterministic PFAs is a proper subclass of the regular ones [12].

3 Learning a Regular Distribution from a Sample

Our strategy in learning a regular distribution on Σ^* from a sample will be to first learn the underlying non-deterministic RFSA automaton and then enrich its states and transitions with weights such that the resulting probabilistic automaton is consistent with the sample.

A *sample* (S, f) consists of a finite set of strings $S \subseteq \Sigma^*$ together with a frequency function $f : S \to \mathbb{N}$ assigning the number of occurrences of each string in the sample. The frequency function f partitions the strings in S into positive samples and negative ones. We denote by $S_+ = \{x \mid f(x) > 0\}$ the set of positive samples and by $S_- = \{x \mid f(x) = 0\}$ the set of negative samples. Obviously, one can convert a sample to a discrete distribution $D : S \to \mathbb{Q} \cap [0, 1]$ by mapping each x in the sample to its frequency divided by the total number of observations in the sample.

An NFA A is *consistent* with respect to a sample (S, f), if every positive sample is accepted by A and every negative sample is not, i.e. $S_+ \subseteq L(A)$ and $S_- \cap L(A) = \emptyset$. Learning a regular distribution D from a sample (S, f) of finite strings independently drawn with a frequency f according to the distribution D means building a probabilistic finite automaton A with a support consistent with respect to (S, f), and generating a distribution $D(A)$ that gets arbitrarily closer to D when the size of the sample (S, f) increases.

3.1 Learning the Structure

Assume given a regular distribution D and a sample (S, f) generated from D by counting the occurrence of independent draws. We build a RFSA from a sample (S, f) using the algorithm presented in [9]. The algorithm is similar to that presented in [11] but approximates the inclusion relation between residual languages by calculating on the fly the transitivity and right-invariant (with respect to concatenation) closure $\prec^{tr}$ of the $\prec$ relation, defined for $u, v \in Pref(S_+)$ by:

- $u \prec v$ if for all string x, $ux \notin S_+$ or $vx \notin S_-$,

Two strings u and v are indistinguishable with respect to a sample (S, f), denoted by $u \simeq v$, if both $u \prec v$ and $v \prec u$. This means that we can distinguish two strings u and v if we can extend them with a string x such that one of the resulting string belong to the positive sample and another to the negative sample.

We will use prefixes of strings in the positive samples as states of the learned RFSA, but we want to equate indistinguishable states with respect to the sample. Here $u \prec^{tr} v$ is an estimate for the inclusion between the residuals $L_u \subseteq L_v$. In fact, under some conditions on the sample with respect to language underlying the distribution D to be learned, this approximation will be exact [11].

Algorithm 1: Building a RFSA from a simple sample

Input: A simple sample (S, f)
Output: A RFSA $\langle \Sigma, Q, I, F, \delta \rangle$
1: Pref := $Pref(S_+)$ ordered by length-lexicographic order
2: $Q := I := F := \delta := \emptyset$
3: $u := \varepsilon$
4: **loop**
5:   **if** $\exists u' \in Q$ such that $u \simeq^{tr} u'$ **then**
6:     Pref := Pref $\backslash u\Sigma^*$
7:   **else**
8:     $Q := Q \cup \{u\}$
9:     **if** $u \prec^{tr} \varepsilon$ **then**
10:       $I := I \cup \{u\}$
11:     **if** $u \in S_+$ **then**
12:       $F := F \cup \{u\}$
13:     **for** $u' \in Q$ and $a \in \Sigma$ **do**
14:       **if** $u'a \in$ Pref and $u \prec^{tr} u'a$ **then**
15:         $\delta := \delta \cup \{\delta(u', a) = u\}$
16:       **if** $ua \in$ Pref and $u' \prec^{tr} ua$ **then**
17:         $\delta := \delta \cup \{\delta(u, a) = u')\}$
18:   **if** u is the last string of Pref or $\langle \Sigma, Q, I, F, \delta \rangle$ is consistent with S **then**
19:     exit loop
20:   **else**
21:     $u :=$ next string in Pref
22: **return** $\langle \Sigma, Q, I, F, \delta \rangle$

The algorithm is shown in 1. Basically, given a sample (S, f) the algorithm starts with an empty set of states Q for the learned NFA. All prefixes of S_+ are explored, and only those which are distinguishable are added to the Q. States below λ with respect to $\prec$ are set to be initial states, while states that belong

to S_+ are final ones. Finally, a transition $\delta(u,a)(v) = 1$ is added when $v \prec ua$, for u and v distinguishable prefixes of S_+ and $a \in \Sigma$. The algorithm ends either when all prefixes of S_+ are explored or earlier if the learned automaton is consistent with the sample. By construction the resulting automaton will be a (non-deterministic) RFSA consistent with respect to the sample. In general, the above schema will learn regular languages as NFA in the limit in polynomial time and space.

Example 1. Given a sample (S, f) with $f(\lambda) = 0.3$, $f(aa) = 0.03$, $f(ba) = 0.039$, $f(bb) = 0.036$, $f(abb) = 0.0045$, $f(a) = f(b) = f(ab) = 0$, so that $S_+ = \{\lambda, aa, ba, bb, abb\}$ and $S_- = \{a, b, ab\}$. Prefixes of strings in S_+ missing from S_+ are a, b, ab. Because $aa \in S_+$ and $\lambda a \in S_-$ it follows that $a \not\prec^{tr} \lambda$. Hence a is distinguishable from λ, it is not an initial state and is also not final. State λ instead is both initial and final. Finally, $\delta(\lambda, a)(a) = 1$ and $\delta(a, a)(\lambda) = 1$ since $a \prec^{tr} a$ and $\lambda \prec^{tr} aa$. Since the automata is not yet consistent with the sample, we add a new prefix of S_+ as state. String b is distinguishable from λ for similar reason as a, and again it is neither final nor initial. It is also distinguishable from a because $b \not\prec^{tr} a$ as they are both in S_-. Six transitions are added: $\delta(\lambda, b)(a) = 1$, $\delta(\lambda, b)(b) = 1$, $\delta(b, a)(\lambda) = 1$, $\delta(b, b)(\lambda) = 1$, $\delta(a, b)(b) = 1$ and $\delta(b, a)(b) = 1$. Now the automaton is consistent with the sample, and thus the algorithm terminates. See Fig. 1a.

3.2 Learning the Probabilities

Once we have learned the structure of the RFSA needed to represent the language underling the sample (S, f), we need to label it with weights representing the probabilities of a PFA. We will treat the probabilities for the initial states, the final states and the transitions as parameters, that will be used as variables in the solution of a non-linear optimization problem.

Given an NFA A, we first construct a system of equations depending on the structure of the automaton and the probabilities induced by the sample for each string in it. For each state $q \in Q$, we have variables i_q and e_q to denote the unknown values of $I(q)$ and $F(q)$, respectively. We also use variables $x^a_{q,q'}$ for denoting the unknown probability of the transition $\delta(q, a)(q')$. We add a few structural equations which are dictated by the structure PFA definition:

$$\sum_q i_q = 1, \quad \text{and} \quad \text{for all } q \in Q,\ f_q + \sum_{a,q'} x^a_{q,q'} = 1$$

Besides the above structural equations, we have equations depending on the sample and the automaton. For each string $u = a_0 \cdots a_n \in S$ we define $E(u)$ to be the polynomial equation:

$$\sum_{q_0 \cdots q_{n+1} \in Paths_p(u)} i_{q_0} \cdot x^{a_0}_{q_0,q_1} \cdots x^{a_n}_{q_n,q_{n+1}} \cdot e_{q_{n+1}}(\pi) = p(u).$$

where $p(u)$ is the probability of u induced by the frequency f in the sample. In other words, equations like $E(u)$ above represent the symbolic calculation of

the probability of u in the automaton A with weights as parameters. In order to guarantee linear independence between the equations, we consider prime strings in S. A string u is said to be prime if there exists at least one path in $Path_p(u)$ without repeated loops, that is, without occurrence of the same part (at least two states) twice. If we have more prime strings in S than variable, we consider only prime strings u to build our equations $E(u)$. Otherwise, we consider strings from S_+, prioritizing them in lexicographic order. If we have a small sample with more variables than strings in S the result may be very poor, as expected.

We rewrite the system of equations as a function with some constraints. The function is derived from the structural equations while the constraints are stemming from the sample depending equations. We use three different methods to result optimization problem with constraints. The first one is via the solvers module in SymPy [19]. SymPy is a Python library for solving equations symbolically, trying to find algebraic solutions. In our experiments below, in most cases, SymPy is not able to find the exact algebraic solution, due to the fact that the structure of learned automaton is not always equal to the target one. The second method uses a genetic algorithm (GA). GAs are computational models simulating ideal for searching optimal solutions by imitating the natural evolutionary processes. GAs take individuals in a population and use randomization techniques to guide an efficient search of an encoded parameter space [20]. The third method is based on Sequential Quadratic Programming (SQP), one of the most widely-used methods for solving nonlinear constrained optimization problems [4]. It is an iteration method with a sound mathematical foundation that can be applied to large scale optimization problems.

The solutions from the GA and SQP methods are an approximation of the results, and in general will need a light adaptation via normalization to satisfy the structural rules of a PFA.

Example 2. Given the RFSA in Fig. 1a constructed from the sample (S, f) with $f(\lambda) = 0.3$, $f(aa) = 0.03$, $f(ba) = 0.039$, $f(bb) = 0.036$, $f(abb) = 0.0045$, $f(a) = f(b) = f(ab) = 0$, we obtain the PFA with variables as in Fig. 1b. From that we derive the system of equations

$$\begin{cases} i_\lambda = 1 \\ f_\lambda + x^a_{\lambda,a} + x^b_{\lambda,a} + x^b_{\lambda,b} = 1 \\ x^a_{a,\lambda} + x^b_{a,b} = 1 \\ x^a_{b,\lambda} + x^b_{b,\lambda} + x^a_{b,b} = 1 \end{cases} \qquad \begin{cases} i_\lambda f_\lambda = 0.3 \\ i_\lambda x^a_{\lambda,a} x^a_{a,\lambda} f_\lambda = 0.03 \\ i_\lambda x^b_{\lambda,a} x^a_{a,\lambda} f_\lambda + i_\lambda x^b_{\lambda,b} x^a_{b,\lambda} f_\lambda = 0.039 \\ i_\lambda x^b_{\lambda,b} x^b_{b,\lambda} f_\lambda = 0.036 \\ i_\lambda x^a_{\lambda,a} x^b_{a,b} x^b_{b,\lambda} f_\lambda = 0.0045 \end{cases}$$

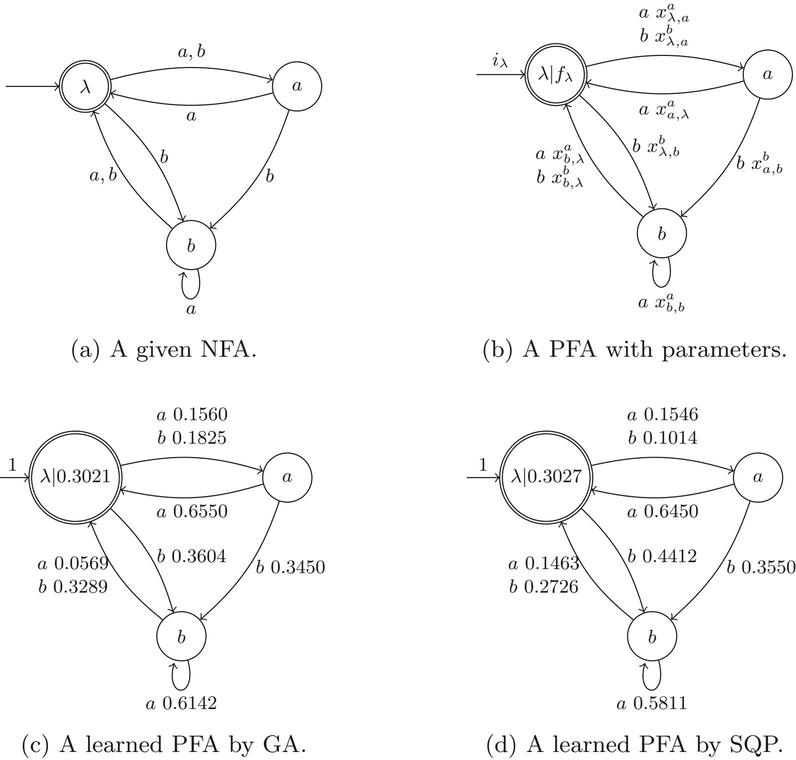
The corresponding function to be optimized is

$$(i_\lambda - 1)^2 + (f_\lambda + x^a_{\lambda,a} + x^b_{\lambda,a} + x^b_{\lambda,b} - 1)^2 + (x^a_{a,\lambda} + x^b_{a,b} - 1)^2 + (x^a_{b,\lambda} + x^b_{b,\lambda} + x^a_{b,b} - 1)^2 = 0$$

with as constraints all variables ranging between 0 and 1 and:

$$(i_\lambda f_\lambda - 0.3)^2 = 0 \qquad (i_\lambda x^b_{\lambda,b} x^b_{b,\lambda} f_\lambda - 0.036)^2 = 0$$
$$(i_\lambda x^a_{\lambda,a} x^a_{a,\lambda} f_\lambda - 0.03)^2 = 0 \qquad (i_\lambda x^a_{\lambda,a} x^b_{a,b} x^b_{b,\lambda} f_\lambda - 0.0045)^2 = 0$$
$$(i_\lambda x^b_{\lambda,a} x^a_{a,\lambda} f_\lambda + i_\lambda x^b_{\lambda,b} x^a_{b,\lambda} f_\lambda - 0.039)^2 = 0.$$

Then we use the GA and SQP to approximate the solution, the results are shown in Fig. 1c and Fig. 1d. The learned PFAs approximate to the given sample closely.



(a) A given NFA.

(b) A PFA with parameters.

(c) A learned PFA by GA.

(d) A learned PFA by SQP.

Fig. 1. The RFSA and PFA automata from Example 1.

4 Experimental Results

In this section, we summarize some experiments to compare the performance of our new learning method with other existing algorithms using some distributions generated from a random set of PFAs. In particular, we consider ALERGIA, the most well-known probabilistic languages learning algorithm, k-testable algorithm and RFSA algorithm with flip-coin distribution. ALERGIA and k-testable can identify only deterministic distributions. We use 999 different parameters setup for ALERGIA and 14 different values for k for the k-testable method. We avoid too large values for k to not make the learning model overfitting. In both cases we only consider the parameter achieving the best performance. For RFSA-GA and RFSA-SQP, we choose 1000 different start points at random. Also here we choose the start point with best result for each algorithm.

4.1 Learning Randomly Generated Probabilistic Automata

Target automata are generated by a PFA generator according to the number of states, symbols and transitions for each state. We generate three sets of 20 automata each with 3, 5 and 10 states. All automata are over a 2 symbols alphabet and with at most 3 transitions for each state. The probabilities of initial states, final states and of the transitions are chosen randomly. There are both DPFAs and PFAs.

From each of these 60 automata, we generate a sample of 248 strings over a two symbols alphabet uniformly and use the automaton to compute a probability for each string, including strings with probability 0. We generate samples with frequencies by scaling up the probabilities. For each sample, we learn an automaton with six different algorithms. We compute the L_2 distance between each learned automaton and the respective target PFA [9], considering the smallest L_2 distance for each algorithm. We repeat this experimental setup 20 times for different target automata, give the average variance of results, and then calculate the improvement between the best of our new methods and the best of the others. The results are reported in the table below. There are no results of RFSA with solver algorithm in this table since we cannot find the exact algebraic solutions in 75% of the cases for the set of 3-states automata. More experimental results are shown in Appendix.

For $3-$states automata, our method combining RFSA learning with genetic algorithms (RFSA-GA) has on average the smallest distance from the target distribution and the smallest variance too, with an improvement on the learning via k-testable algorithm of 90%. The combination of RFSA with SQP scores as the second best on average. The average size of the automata learned by RFSA-GA is 3.05 states on average, a significant improvement compared to 12.95 for ALERGIA and 66.65 for k-testable. This means that RFSA learned automata structure is much simpler and closer to the target model.

As for 5-states automata, the situation is similar, with RFSA-GA scoring as the best, followed by RFSA-SQP and the k-testable algorithm. Our RFSA-GA algorithm learns 4.6 states on average, compared with 13.15 states by ALERGIA and 54.55 states by k-testable.

When the target automata have 10 states, the RFSA-GA still has the smallest average and variance with an improvement of 86% when compared to ALERGIA. The RFSA learns 8.1 states on average, while ALERGIA and k-testable get 20.1 and 59.4 states, respectively.

Only when the algebraic solver can find the solution, we have that the learned automaton is closer to the target one than RFSA-GA. In some cases the distance is even 0, meaning that the distribution learned is precisely the target one. In a few other cases, the distance is almost 0 due to the approximate structure given by the RFSA. In the table, we see the results of RFSA combined with a flip-coin method, assigning probabilities by equally distributing them among the transition. Clearly, this naive strategy has the largest distance on average from the target automata, but is not extremely far from ALERGIA and k-testable,

underlying the importance of a simple and as close as the possible structure of the learned automata with respect to the target ones (Table 1).

Table 1. Averages, variances and improvements of L_2 distance between target 3-state, 5-state, 10-state automata and learned automata respectively.

Algorithms	3-states		5-states		10-states	
	Average	Variance	Average	Variance	Average	Variance
ALERGIA	0.1874	0.0208	0.1462	0.0238	0.2121	0.0310
k-testable	0.1729	0.0202	0.1065	0.0095	0.2128	0.0317
Flip-coin	0.2229	0.02147	0.1593	0.0112	0.2807	0.0324
RFSA-SQP	0.0213	0.0013	0.0348	0.0034	0.0301	0.0031
RFSA-GA	**0.0171**	**0.0006**	**0.0289**	**0.0017**	**0.0264**	**0.0007**
Improvement	90%↓	97%↓	67%↓	82%↓	86%↓	98%↓

4.2 Learning a Model of an Agent's Traces in a Maze

Next we compare our optimization-based approaches using a model for which we do not know a-priori the target regular distribution, but we only have a sample with frequency, as often happens in a real-world situation.

The idea is to build a model for an intelligent agent in two dimensional space with the goal of arriving to target end points. For simplicity, the space is represented as a matrix of possible positions, and the agent in any position can take four actions representing a move up, down, left or right to the current position. We model the agent as a PFA $A = \langle \Sigma, Q, I_p, F_p, \delta_p \rangle$. Here $\Sigma = \{U, D, L, R\}$ is the set of the four actions that the agent can perform, and strings over Σ represent possible consecutive actions taken by the agent. The set of states Q contains all possible positions of the agent in the space. I_p is the set of probabilities of being at a certain starting state, F_p is assigning 1 only to those states that are the target end points, and δ_p is the set of probabilities of executing one of the four actions in a state. We assume given a number of sequence of possible consecutive that are obtained, for example, in a training phase, when the agent uniformly select an action to try to find the target end point. Differently than ordinary reinforcement learning methods, we assume not known a-priory the size and shape of space, that, moreover, may have insurmountable obstacles.

Training an automaton from a sample is therefore to find the set of states Q, and right structure where the agent determine the probabilities of each transition $\delta(q, a)(q')$, the initial probabilities I_p and the final one F_p in accordance to the space structure.

We generated 20 different 10×10 rectangle maps of the space, all of them surrounded by obstacles (walls) that an agent cannot trespass. We differentiate those spaces randomly generating obstacles inside. For simplicity, for each map

we choose only one start state (say with coordinates $(0, 0)$) and only one target end state that is randomly chosen among the allowed positions. Here we show a simple example about the positive and negative samples under certain agent's moving memoryless strategy.

Example 3. Figure 2 is the illustration of a 3×3 maze, where 0 is available, 1 is an obstacle. $(0, 0)$ is the start point, $(2, 2)$ is the end point. The agent's moving strategy is $\{U : 0.1, D : 0.4, L : 0.1, R : 0.4\}$. The traces from start point to end point are positive samples. Otherwise, the traces hit the wall or the obstacle are negative samples. The following are few instances of traces in a sample (S, f): $f(DDLL) = 0.0256$, $f(DUDDLL) = 0.001024$, $f(DDLRLL) = 0.001024$, $f(DUDUDDLL) = 0.00004108$, $f(U) = 0$, $f(UUDD) = 0$, $f(DLRLU) = 0$ and $f(DRLDD) = 0$.

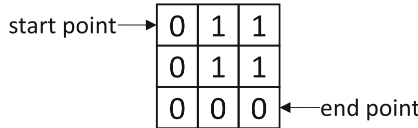


Fig. 2. A 3×3 maze, where 0 means available, 1 is an obstacle. $(0, 0)$ is the start point, $(2, 2)$ is the end point.

We simulate a training phase for the agent by using a uniform strategy, that is, we generate a trace by uniformly selecting the next action among the set of allowed one (thus avoiding obstacles). Note that this is a deterministic strategy and can therefore be approximated by all other methods for learning PFA we have considered in the previous section. The traces successfully reaching the target end point are our positive samples, with associated frequency (or probability) as calculated on the basis of the probability of each action taken.

In order to balance the data, we consider as negative samples all prefixes of the positive one (we assume that the agent once arriving to a target end state it stops) and concatenation of prefixes with suffixes that do not occurs as positive samples. We use 90% of the resulting traces for training, and 10% for evaluating the learned automaton and compare the performance with other PFA learning methods. As we do not have the full distribution to be learned in advance, to compare the different methods we use the F_1 score and optimized precision OP. Both methods are based on a probabilistic version of precision, sensitivity, specificity and accuracy, where the number of true positive TP and false negative FN is weighted by a the $L1$ distance between the finite sample (S, f_p) and the regular distribution $D(A)$ of the automaton A learned using one of the method we consider. In particular we consider

$$Precision = \frac{cTP}{|TP|+cFP} \qquad Sensitivity = \frac{cTP}{|TP|+cFN}$$
$$Accuracy = \frac{|TP|+|TN|}{|TP|+|TN|+|FP|+|FN|} \qquad Specificity = \frac{|TN|}{|TN|+cFP}$$

where $|TP|, |TN|, |FP|$, and $|FN|$ are the number of true positive, true negative, false positive and false negative of the automaton A with respect to the sample, and $cTP = \sum_{x \in TP} 1 - |f(x) - D(A)(x)|$, $cFN = \sum_{x \in FN} f(x)$, and $cFP = \sum_{x \in FN} D(A)(x)$ (see [9] for a more extensive discussion). The F_1 score [25] is used to measure the method accuracy. It is computed in terms of both the precision and sensitivity, and basically is the harmonic average of them.

$$F_1 = 2 \cdot \frac{Precision \cdot Sensitivity}{Precision + Sensitivity} \tag{2}$$

The optimized precision OP [5,23] is a hybrid threshold metric combining accuracy, sensitivity and specificity, where the last two are used for stabilizing and optimizing the accuracy when dealing with possibly imbalanced data.

$$OP = Accuracy - \frac{|Specificity - Sensitivity|}{|Specificity + Sensitivity|} \tag{3}$$

When the distribution of the learned automaton coincides with that of the sample, precision, sensitivity and accuracy will all be 1, and thus both F_1 and OP will be equal to 1 too. However, the more the distribution of the learned automaton is distant from that of the sample, the more precision, sensitivity and accuracy will be closer to 0, setting both the scores F_1 and OP closer to 0 too.

Table 2 shows the summary of the results taking the average and the variance when learning with different methods the 20 mazes from the randomly generated samples. As in the case of learning the randomy generated automata, the RFSA method enhanced with an algebraic solver does not work in general because of the too many variables involved. RFSA-SQP is the most stable method as it has the lowest variance across the different mazes when compared using the F_1 score. In general, all algorithms perform well with respect to the F_1 score. However, when considering the OP score we see that RFSA-GA has the highest OP score on average and also has the lowest variance. This means that RFSA-GA has low probability of false positives and false negatives. When compared with the second best given by learning using the k-testable algorithm, we see that RFSA-GA has an improvement of 21% on the average OP score.

Table 2. Average, variance and improvement of F_1 and OP

Algorithms	F_1		OP	
	Average	Variance	Average	Variance
ALERGIA	0.9933	0.0003	0.3431	0.0107
k-testable	0.9997	1.68e−08	0.7990	0.0050
Flip-coin	0.9998	1.28e−08	0.9679	0.0005
RFSA-SQP	**0.9998**	**9.47e−09**	0.9586	0.0012
RFSA-GA	0.9998	9.94e−09	**0.9683**	**0.0004**
Improvement	0.003%↑	43%↑	21%↑	91%↑

5 Conclusion

In this paper, we learn regular distributions by combining the learning of the structure via RFSA with the learning of the probabilities using three different optimization methods: an algebraic solver, a genetic algorithm and sequential quadratic programming. We use some randomly generated PFAs and modeling an agent's traces in a maze for comparing these methods with existing ones. While theoretically the algebraic solver method is the best, in practice it often fails to provide a solution even for three states automata. The other two optimization methods are iterative and always find an approximate solution. In practice, we have seen that the solution is very close to the target distribution, order of magnitudes more than existing algorithms. Because the structure learned via RFSA is a non-deterministic automaton, our method behaves well even for regular distributions that are not deterministic, showing that one of the disadvantage of learning regular languages by RFSA has been actually turned into an advantage in the context of learning regular distribution. Besides, compared with k-testable and ALERGIA algorithm, which could only learn positive samples well, our method can model both positive and negative samples well. Important in learning the structure is the presence of both examples and counterexamples, i.e. strings with 0 frequency/probability, and to have a fair balance between them. The scalability of our algorithm depends very much on the scalability of the non-linearity optimization method used to solve the constrained equations. Algebraic solver becomes impractical already with 5 states automata. In contrast, GA and the SQP method seem to be more appropriate for larger one. Many works investigate concurrency to improve the scalability of the GA and SQP algorithms. Specifically, the evolutionary algorithm we used in our experiments is capable of learning in reasonable time automata up to 62 states and hundreds of transitions, resulting in a system with more than 110 variables. Our algorithm could be used for speech recognition, and biological modeling depending on how large the samples and target automata are. Next, we plan to investigate how our algorithm performs in these practical situations.

A Appendix

(See Table 3).

Table 3. L_2 distance between target 3-states automata and learned automata.

Nb	RFSA-solver	RFSA-SQP	RFSA-GA	ALERGIA	k-testable
1	-	**0.0023**	0.0260	0.1962	0.1822
2	**3.73e–09**	0.0006	0.0033	0.1714	0.1314
3	-	2.33e–05	**1.23e–06**	0.2621	0.1847
4	**1.29e–08**	0.0054	2.91e–06	0.0593	0.0561
5	-	**0.0002**	0.0105	0.5255	0.5251
6	**0**	0.0045	3.39e–06	0.2798	0.1326
7	-	**0.0001**	0.0164	0.0365	0.0366
8	-	0.1021	0.1027	0.0064	**0.0046**
9	-	0.0006	0.0023	0.0491	0.0492
10	-	**7.94e–05**	0.0041	0.0152	0.0118
11	3.72e–05	0.0033	**3.05e–05**	0.0002	0.0002
12	-	**0.0123**	0.0131	0.1994	0.1989
13	-	**0.0265**	0.0279	0.1638	0.1634
14	-	0.0913	**0.0141**	0.5176	0.5172
15	-	**0.0290**	0.0296	0.2138	0.2101
16	-	0.0053	**1.49e–08**	0.3081	0.3080
17	-	0.1162	**0.0570**	0.2418	0.2417
18	-	0.0257	**0.0248**	0.1243	0.1928
19	-	**8.01e–05**	0.0101	0.1733	0.1732
20	**0**	1.22e–05	6.43e–05	0.2050	0.1388

From this table, we see that only 5 times the RFSA with algebraic solver found a solution, either approximate or precise. For all other automata no solution could be found, even if there were only at most 15 variables in the system of equations. Interestingly, the approximate solution resulted for automaton 11 with the algebraic solver is not better than the one found by a genetic algorithm.

B Appendix

From Table 4, we see that RFSA-GA outperforms other algorithms in 19 out of 20 experiments, proving the stability and generality of our RFSA-GA method.

From the Table 5, we see that 15 times the RFSA with SQP algorithm outperforms. We also see all RFSA and k-testable work well since F_1 score only considers how well to learn the positive samples.

Table 4. OP of all learned automata.

Nb	RFSA-flip	RFSA-GA	RFSA-SQP	k-testable	ALERGIA
1	0.9684	**0.9684**	0.9684	0.8287	0.3732
2	0.9890	**0.9890**	0.9890	0.8287	0.3402
3	0.9802	**0.9802**	0.9802	0.9303	0.3672
4	0.9162	**0.9211**	0.9071	0.7261	0.2539
5	0.9589	**0.9589**	0.9589	0.7426	0.2119
6	0.9520	0.9521	**0.9522**	0.7471	0.1326
7	0.9909	**0.9909**	0.9909	0.7714	0.3403
8	0.9691	**0.9691**	0.9691	0.8445	0.3648
9	0.9725	**0.9725**	0.9725	0.7674	0.2765
10	0.9820	**0.9820**	0.9820	0.7604	0.3436
11	0.9381	**0.9401**	0.9381	0.7658	0.3646
12	0.9955	**0.9955**	0.9685	0.7299	0.4503
13	0.9877	**0.9877**	0.9877	0.8697	0.4878
14	0.9745	**0.9745**	0.9745	0.8124	0.5544
15	0.9342	**0.9351**	0.8741	0.8804	0.1451
16	0.9315	**0.9325**	0.9315	0.7880	0.5377
17	0.9651	**0.9651**	0.8755	0.7577	0.3898
18	0.9829	**0.9829**	0.9829	0.7810	0.2608
19	0.9850	**0.9850**	0.9850	0.6998	0.2499
20	0.9840	**0.9840**	0.9840	0.7890	0.2817

Table 5. F_1 of all learned automata.

Nb	RFSA-flip	RFSA-GA	RFSA-SQP	k-testable	ALERGIA
1	0.999432	**0.999498**	0.999497	0.999362	0.987498
2	0.999752	0.999751	**0.999775**	0.999706	0.923148
3	0.999783	0.999790	**0.999813**	0.999731	0.999541
4	0.999832	0.999846	**0.999848**	0.999826	0.999819
5	0.999801	0.999806	**0.999811**	0.999772	0.998079
6	0.999809	0.999807	**0.999816**	0.999785	0.998923
7	0.999823	0.999819	**0.999825**	0.999806	0.999560
8	**0.999824**	0.999809	0.999821	0.999807	0.999603
9	**0.999795**	0.999754	0.999769	0.999766	0.999331
10	0.999819	0.999814	**0.999831**	0.999866	0.966374
11	0.999817	0.999811	**0.999819**	0.999809	0.999786
12	0.999888	0.999896	**0.999898**	0.999857	0.999648
13	0.999882	0.999889	**0.999892**	0.999862	0.999849
14	0.999898	0.999899	**0.999902**	0.999877	0.999587
15	0.999721	**0.999733**	0.999732	0.999733	0.999033
16	0.999719	0.999723	**0.999733**	0.999738	0.999787
17	0.999700	0.999704	**0.999707**	0.999698	0.998969
18	0.999823	0.999824	**0.999849**	0.999749	0.999636
19	0.999518	0.999595	**0.999595**	0.999502	0.998340
20	0.999700	**0.999701**	0.999700	0.999499	0.998910

References

1. Bahl, L.R., Brown, P.F., de Souza, P.V., Mercer, R.L.: Estimating hidden Markov model parameters so as to maximize speech recognition accuracy. IEEE Trans. Speech Audio Process. **1**(1), 77–83 (1993)
2. Baldi, P., Brunak, S.: Bioinformatics: The Machine Learning Approach. MIT Press, Cambridge (2001)
3. Baum, L.E., Petrie, T., Soules, G., Weiss, N.: A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. Ann. Math. Stat. **41**(1), 164–171 (1970)
4. Bonnans, J.F., Gilbert, J.C., Lemaréchal, C., Sagastizábal, C.A.: Numerical Optimization: Theoretical and Practical Aspects. Springer, Heidelberg (2006). https://doi.org/10.1007/978-3-540-35447-5
5. Branco, P., Torgo, L., Ribeiro, R.P.: A survey of predictive modeling on imbalanced domains. ACM Comput. Surv. (CSUR) **49**(2), 1–50 (2016)
6. Carrasco, R.C., Oncina, J.: Learning stochastic regular grammars by means of a state merging method. In: Carrasco, R.C., Oncina, J. (eds.) ICGI 1994. LNCS, vol. 862, pp. 139–152. Springer, Heidelberg (1994). https://doi.org/10.1007/3-540-58473-0_144
7. Carrasco, R.C., Oncina, J.: Learning deterministic regular grammars from stochastic samples in polynomial time. RAIRO-Theor. Inform. Appl. **33**(1), 1–19 (1999)
8. Chu, W., Bonsangue, M.: Learning probabilistic languages by k-testable machines. In: 2020 International Symposium on Theoretical Aspects of Software Engineering (TASE), pp. 129–136. IEEE (2020)
9. Chu, W., Chen, S., Bonsangue, M.: Learning probabilistic automata using residuals. In: Cerone, A., Ölveczky, P.C. (eds.) ICTAC 2021. LNCS, vol. 12819, pp. 295–313. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85315-0_17
10. De La Higuera, C.: Characteristic sets for polynomial grammatical inference. Mach. Learn. **27**(2), 125–138 (1997)
11. Denis, F., Lemay, A., Terlutte, A.: Learning regular languages using RFSAs. Theor. Comput. Sci. **313**(2), 267–294 (2004)
12. Dupont, P., Denis, F., Esposito, Y.: Links between probabilistic automata and hidden Markov models: probability distributions, learning models and induction algorithms. Pattern Recogn. **38**(9), 1349–1371 (2005)
13. Durbin, R., Eddy, S.R., Krogh, A., Mitchison, G.: Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, Cambridge (1998)
14. Feldmann, A., Whitt, W.: Fitting mixtures of exponentials to long-tail distributions to analyze network performance models. Perform. Eval. **31**(3–4), 245–279 (1998)
15. De la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, Cambridge (2010)
16. de la Higuera, C., Thollard, F., Vidal, E., Casacuberta, F., Carrasco, R.C.: Probabilistic finite state automata-part ii. Rapport technique RR-0403, EURISE (2004)
17. Jelinek, F.: Statistical Methods for Speech Recognition. MIT Press, Cambridge (1998)
18. Lee, K.F.: On large-vocabulary speaker-independent continuous speech recognition. Speech Commun. **7**(4), 375–379 (1988)
19. Meurer, A., et al.: SymPy: symbolic computing in python. PeerJ Comput. Sci. **3**, e103 (2017)

20. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press, Cambridge (1998)
21. Murphy, K.P., et al.: Passively Learning Finite Automata. Citeseer (1995)
22. Paz, A.: Introduction to Probabilistic Automata. Academic Press, Cambridge (2014)
23. Ranawana, R., Palade, V.: Optimized precision-a new measure for classifier performance evaluation. In: 2006 IEEE International Conference on Evolutionary Computation, pp. 2254–2261. IEEE (2006)
24. Ron, D., Singer, Y., Tishby, N.: On the learnability and usage of acyclic probabilistic finite automata. J. Comput. Syst. Sci. **56**(2), 133–152 (1998)
25. Sammut, C., Webb, G.I.: Encyclopedia of Machine Learning. Springer, Heidelberg (2011). https://doi.org/10.1007/978-0-387-30164-8
26. Turin, W.: Fitting probabilistic automata via the EM algorithm. Stoch. Model. **12**(3), 405–424 (1996)
27. Turin, W., Van Nobelen, R.: Hidden Markov modeling of flat fading channels. IEEE J. Sel. Areas Commun. **16**(9), 1809–1817 (1998)
28. Vidal, E., Thollard, F., de la Higuera, C., Casacuberta, F., Carrasco, R.: Probabilistic finite state automata-part I. Pattern Anal. Mach. Intell. **27**(7), 1013–1025 (2005)