Automata learning: from probabilistic to quantum

Chu, W.

Chapter 4

Passive Learning PFAs with Counterexamples

In the previous chapter, given a finite alphabet Σ, we have seen that we can learn a subset of the regular distribution on Σ^* based on a finite set of strings, each equipped with a positive frequency, and a parameter k determining how many symbols one can remember. Next, we continue our investigation on passive learning regular distributions on Σ^* by enlarging the class of distributions that we can learn and by removing the input parameter k. To do this, we will use a learning technique based on both positive and negative samples. A negative sample is a string that occurs with probability 0, or below a certain fixed threshold. Since the string does not belong to the language support of the distribution to be learned, it can be considered as a counterexample. Samples with zero probabilities reflect our prior knowledge about the system or the problem and are not a result of statistical estimation. For example, certain events are impossible or explicitly excluded based on our domain knowledge, physical constraints, or the nature of the problem being modeled.

Learning from examples and counterexamples is a common approach in machine learning, where counterexamples are used in refining and improving the learned model by explicitly indicating what the model should not do or predict. In binary classification tasks, where the goal is to learn to differentiate between two classes, counterexamples belonging to the negative class are used in defining the rules that separate the two classes. For example, the task of a Support Vector Machine (SVM) is to find a hyperplane that best separates the positive and negative examples in a high-dimensional space [72]. Also, while the Naive Bayes algorithm does not typically use counterexamples as distinct elements, the algorithm can benefit from a training dataset that includes both positive and negative instances [15, 48].

Given a regular language L over an alphabet Σ, we have seen that a positive sample is simply a finite subset of L, representing strings that need to be recognized. A negative

sample, instead, is a finite set that has no string in common with L and thus should be not accepted by the automaton to be learned. When moving from regular languages to regular distributions over Σ, a positive sample is a finite set of strings with associated frequencies that conform to the distribution. Strings with explicit 0 frequency cannot occur in the set of positive samples, as they have a probability of zero of occurring, an impossible event that cannot happen in any circumstances. However, these impossible events might be known a priori and therefore included in the process of learning a distribution to emphasize the distinction between possible and impossible outcomes. Therefore they form a set of negative samples. In practice, one can also consider strings with low counts (relative to the other strings in the sample) of observed frequencies as part of the negative sample.

4.1 Learning from positive and negative samples

When learning a regular language, positive data alone is not sufficient in the deterministic case [46]. Labelling strings as either being in the language (positive samples) or not in the language (negative samples) may help. For example, learning an automaton may start with an initial representation that strictly recognizes only the positive samples and then tries to merge its states. The merging occurs only when there is no evidence that the associated language is different from the one to be learned, that is, it does not accept any string in the negative sample.

4.1.1 Characteristic Sample

This section introduces important concepts and terminology related to DFA and their behavior on sets of positive and negative examples.

In a DFA A, a state q' is considered *live* if it belongs to a path of an accepting string, that is there exist strings α, β and state q'' such that $\alpha\beta \in L(A)$, $\delta^*(q_0,\alpha)(q') = 1$, $\delta^*(q',\beta)(q'') = 1$, and $F(q'') > 0$. Conversely, a state that is not live is called *dead*. Additionally, a state q' is considered *reachable* if there exists a string α such that $\delta^*(q_0,\alpha) = q'$. Note that all live states are also reachable, but not all reachable states are necessarily live.

Given a sample (S,Fr), we use S_+ to denote the set of positive samples of S, that is, a set of strings such that $S_+ = \{x|Fr(x) > 0\}$. Similarly, we use S_- to denote the set of negative examples of S, meaning that $S_- = \{x|Fr(x) = 0\}$. A sample S is then defined as the union of positive strings in S_+ and negative ones in S_-, that is, $S = S_+ \cup S_-$. A finite automaton is *consistent* with a sample S if it accepts all strings in S_+ and rejects all strings in S_-.

For a DFA A, the set S_+ is said to be *structurally complete* with respect to A, if it includes strings that cover each transition of A and use every element of the set of final states of A as an accepting state [40]. In general, a structurally complete set of positive strings is not unique for a given DFA. Any set of positive samples of a DFA A that includes a structurally complete set of strings is also structurally complete.

Given a language $L \subseteq \Sigma^*$, the set $PREF(L)$ is the set of prefixes of L, and $L_\alpha = \{\beta | \alpha\beta \in L\}$ is the set of all strings beginning with α in L. The length-lexicographic order of strings over the alphabet Σ is denoted by $<$. For example, the enumeration of strings over $\Sigma = \{a,b\}$ in length-lexicographic order is $\varepsilon, a, b, aa, ab, ba, bb, aaa, \ldots$.

The set of short prefixes $S_p(L)$ of a language L is defined as $S_p(L) = \{\alpha \in PREF(L) | \forall \beta \in \Sigma^*$ if $L_\alpha = L_\beta$ then $\alpha < \beta\}$. The kernel $N(L)$ of a language L is defined as $N(L) = \{\varepsilon\} \cup \{\alpha a | \alpha \in S_p(L), a \in \Sigma, \alpha a \in PREF(L)\}$.

Consider a regular language L accepted by a DFA A. A sample $CS_L = CS_L^+ \cup CS_L^-$ is said to be characteristic with respect to the language L if it satisfies the following two conditions:

- $\forall \alpha \in N(L)$, if $\alpha \in L$ then $\alpha \in CS_L^+$ else $\exists \beta \in \Sigma^*$ such that $\alpha\beta \in CS_L^+$
- $\forall \alpha \in S_p(L), \forall \beta \in N(L)$, if $L_\alpha \neq L_\beta$ then $\exists \gamma \in \Sigma^*$ such that $(\alpha\gamma \in CS_L^+$ and $\beta\gamma \in CS_L^-)$ or $(\beta\gamma \in CS_L^+$ and $\alpha\gamma \in CS_L^-)$

Intuitively, the set of short prefixes $S_p(L)$, is a live complete set with respect to A: if two strings are distinguishable then they are distinguishable also by the sample CS_L. Since the kernel $N(L)$ includes the set of short prefixes as a subset it is a live complete set as well. Moreover, The first condition says that $N(L)$ covers every transition between each pair of live states of A. For instance, given the language L corresponding to the DFA A in Fig. 4.1. Here, the set of short prefixes is $S_p(L) = \{\varepsilon, a, b, bb\}$ and the kernel is $N(L) = \{\varepsilon, a, b, bb, bbb\}$. Additionally, the set $S = S^+ \cup S^-$ where $S^+ = \{a, bb, bbbb\}$ and $S^- = \{\varepsilon, b, bbb, abb\}$ is a characteristic sample for L.

4.1.2 Identification in the limit from polynomial time and data

The concept of identification in the limit from polynomial time and data has been introduced by Gold in her seminal work [47]. She proved that regular languages represented by DFAs can be learned within this framework.

Definition 7. *We say that a regular language L is identifiable in the limit from polynomial time and data using the representation scheme R if there exist two algorithms $\mathscr{T}$ and $\mathscr{L}$ such that, for any target language L and any representation r of L in the representation scheme R:*

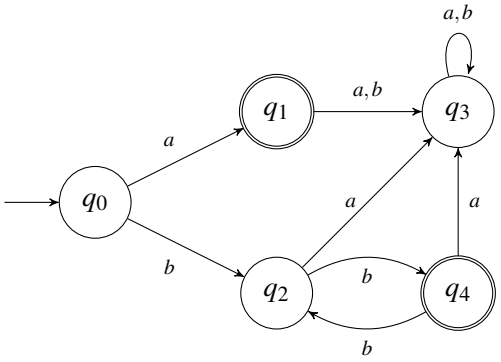


Fig. 4.1 A deterministic finite automaton

- *$\mathscr{T}$, given input r, computes a characteristic sample CS_L whose size is polynomial in the size of r,*
- *for any sample S, $\mathscr{L}$ computes a representation in R compatible with S in time polynomial in the size of S,*
- *if a sample S contains CS_L, $\mathscr{L}$ with input S computes a representation r' equivalent to r, which is $r'(L) = r(L)$.*

Gold showed that the regular language L is identifiable in the limit from polynomial time and data using the representation scheme of DFA [47]. To infer DFAs in this framework, several algorithms have been proposed, including the Greedy Russian algorithm [67] and the Regular Positive and Negative Inference (RPNI) algorithm [89].

However, all these algorithms fail to efficiently infer languages as simple as $\Sigma^*0\Sigma^n$ for fixed n and $\Sigma = \{0,1\}$ because these languages have exponentially long DFA representations. So the question of whether regular languages are identifiable in the limit from polynomial time and data by using more concise representations arises naturally.

Definition 8. *A regular language is polynomially characterizable using the representation scheme R if there exists a function $\mathscr{T}$ such that for any target language $L \in Reg(\Sigma^*)$ and any representation r of L in the representation scheme R:*

- *for any sample S, $\mathscr{L}$ computes a representation in R compatible with S in time polynomial in the size of S,*
- *$\mathscr{T}$ with input r computes a characteristic sample S_L whose size is polynomial in the size of r,*
- *if a sample S contains S_L, $\mathscr{L}$ with input S computes a representation r' equivalent to r.*

4.2 Residual languages and residual finite state automata

For any language L and string $u \in \Sigma^*$, the residual language of L associated with u is defined by the u-derivative $L_u = \{x \in \Sigma^* | ux \in L\}$, and we call u a characterizing word for L_u. A language $L' \subseteq \Sigma^*$ is a residual language of L if a string $u \in \Sigma^*$ exists, such as $L' = L_u$. The number of distinct residual languages of a language L is finite if and only if L is regular [38]. This implies that there exists a finite set of strings $\mathscr{B}(L)$ such that $x \in \mathscr{B}(L)$ if L_x is a residual language of a regular language L. The set $\mathscr{B}(L)$ can be constructed depending on the representation of the language L. For example, suppose L is the language accepted by a trimmed NFA A (i.e., minimal and with all states reachable from an initial state). In that case, $\mathscr{B}(L)$ can be constructed as a finite set of minimal length strings reaching all states of A from some initial state.

Definition 9. ***Residual finite state automaton [33]*** *A residual finite state automaton (RFSA) is an NFA $A = \langle \Sigma, Q, I, F, \delta \rangle$ such that, for each state $q \in Q$, $L(A, q)$ is a residual language of $L(A)$.*

In other words, an RFSA A is a non-deterministic automaton whose states correspond precisely to the residual languages of the language recognized by A. This concept means that the states of the RFSA are intricately linked to the residual languages generated by the automaton.

The regular languages are not identifiable in the limit in polynomial time and data using RFSAs [37]. However, this does not mean that an inference algorithm must not look for an RFSA representation of the target language. This representation may not necessarily be the smallest, but the pursuit of such an RFSA can offer valuable insights into the underlying structure of the target language.

Moreover, extending from non-deterministic automata like RFSAs, there exist generalizations to frequency and probabilistic automata. Frequency finite automata, for instance, associate positive rational numbers with transitions, initial states, and final states, signifying the 'number of occurrences' of a transition or state in the language recognized by the automaton.

4.3 Learning Probabilistic Automata using Residuals

In the previous sections, we have introduced DFFA, and now we delve into the Non-deterministic Frequency Finite Automaton (NFFA). A crucial difference between these two lies in the fact that NFFA operates with rational numbers. This is because, in the case of

a nondeterministic automaton, it implies that for an accepting string, there can be multiple accepting paths. In the forthcoming learning algorithm, we will distribute the frequencies of strings based on the number of paths they have. This approach will yield rational numbers. Furthermore, the application of rational numbers is theoretically viable. By proportionally scaling all the values, we can eventually transform the automaton into one that solely utilizes natural numbers.

Definition 10. ***Nondeterministic frequency finite automaton*** *An nondeterministic frequency finite automaton (NFFA) is a 5-tuple $A = \langle \Sigma, Q, I_f, F_f, \delta_f \rangle$, where:*

- Σ *is a finite alphabet,*
- Q *is a finite set of states,*
- $I_f : Q \to \mathbb{Q}_{\geq 0}$,
- $F_f : Q \to \mathbb{Q}^+$,
- δ_f: $Q \times \Sigma \to \mathbb{Q}^{+Q}$

such that for every state $q \in Q$ the weight of the incoming transitions is equal to the weight of the outgoing transitions:

$$I_f(q) + \sum_{q' \in Q, a \in \Sigma} \delta_f(q', a)(q) = F_f(q) + \sum_{q' \in Q, a \in \Sigma} \delta_f(q, a)(q').$$

Intuitively, the above condition says frequency is preserved by passing through states. Note that we allowed weights to be positive rational numbers instead of positive integers. This is for technical convenience but does not affect the definition. Frequency automata are strictly related to probabilistic automata.

The following lemma will be helpful to later state that if an accepting path contains a cycle, then we can pump that cycle to obtain infinitely many other accepting paths.

Lemma 1. *For a probabilistic automaton A, the probability of an accepting path π with a cycle is strictly smaller than* 1.

4.3.1 Learning structure of probabilistic languages using residuals

An NFA $A = \langle \Sigma, Q, I, F, \delta \rangle$ is consistent concerning a sample (S, Fr), if every positive sample is accepted by A, and every negative sample is not, i.e. $S_+ \subseteq L(A)$ and $S_- \cap L(A) = \emptyset$.

A sample (S, Fr) is *complete* with respect to a regular language L if there exists a finite characteristic set $\mathscr{B}(L) \in \Sigma^*$ such that

- the positive samples cover the language, that is, both x and xa are in $Pref(S_+)$ for every $x \in \mathscr{B}(L)$ and $a \in \Sigma$,
- the positive samples contain enough strings of L, that is, $Pref(S_+) \cap L \subseteq S_+$,
- distinguishable strings in the language are distinguishable in the sample too, that is, for every $u, v \in Pref(S_+)$, if $L_u \nsubseteq L_v$ then there exists $x \in \Sigma^*$ such that $ux \in S_+$ but $vx \in S_-$.

The first condition guarantees that prefixes of strings in S_+ are enough to reach all residual languages of L and to cover all possible transitions from it. The second condition is about requiring all characteristic strings of the residual languages to be in S_+. The third condition ensures that S_- is large enough to distinguish different residual languages.

Learning a regular language L from a sample (S, Fr) means building a non-deterministic finite automaton A consistent with the sample and such that if the sample is complete with respect to L, then $L(A) = L$. Of course, one should consider time and space complexity bounded by the two steps above, which are typically required to be polynomial on the number of strings in the sample and of the model representing the language L [33]

Learning a regular distribution D from a sample (S, Fr) of finite strings independently drawn with a frequency Fr according to the distribution D means building a probabilistic finite automaton A with a support learning the language of the support of D and with a distribution associated with A that gets arbitrarily closer to D when the size of the sample (S, Fr) increases. In general, we cannot realistically expect to get exact information on the learned distribution with respect to the target one.

Next, we present our algorithm to learn an unknown regular distribution D from a sample (S, Fr). The idea is first to learn the non-deterministic structure of the automaton underlying D using residual languages and then label the transitions consistently with the sample frequency using a fair distribution when needed.

In our first step, we use Algorithm 8 below to build an RFSA from a simple sample (S, Fr). The algorithm is similar to that presented in [37] 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 following relation. For $u, v \in Pref(S_+)$, we define:

- $u \prec v$ if there is no string x such that $ux \in S_+$ and $vx \in S_-$,
- $u \simeq v$ if $u \prec v$ and $v \prec u$.

The idea is to characterize all distinguishable states (seen as prefixes of the positive samples). Intuitively, $u \prec^{tr} v$ is an estimate for the inclusion between the residuals $L_u \subseteq L_v$, and if the sample is complete concerning the unknown language L, this is indeed the case.

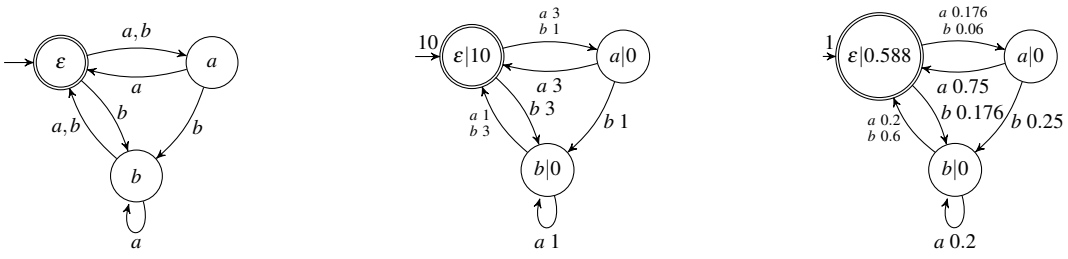
Initially, the set of states Q of the automaton is empty. All prefixes of S_+ are explored, and only those distinguishable are added to Q. States below ε concerning $\prec$ are set to be initial states, while states that belong to S_+ are final ones. Finally, a transition $\delta(u,a) = v$ is added when $v \prec ua$, where $a \in \Sigma$. The algorithm ends when u is the last string in $Pref$ or when the learned automaton is consistent with the sample.

4.3.2 Learning the probabilities with the flip-coin algorithm

Once we have learned the structure of an RFSA from a sample (S, Fr), the next step is adding frequencies to get an FFA based on the frequency information of the sample. This step will not change the structure of the automaton, so Σ and Q are the same as the ones resulting from Algorithm 8. Frequency is distributed fairly by dividing it among non-deterministic transitions.

It is not hard to prove that the resulting automaton is indeed an NFFA, satisfying the frequency preservation condition when passing through states.

Example 7. *Continuing from the previous example, let us consider the case of $ba \in S_+$. Two paths are accepting this string, namely $\varepsilon\ a\ \varepsilon$ and $\varepsilon\ b\ \varepsilon$. As they both start from and end in the same state, $I_f(\varepsilon)$ and $F_f(\varepsilon)$ are incremented by 2, respectively. However, the frequency $f(ba) = 2$ is divided equally between the two b-transitions from state ε, incrementing each of them by 1. After all strings in S_+ are treated, we get the NFFA shown in Fig.4.2b.*



(a) RFSA (b) NFFA (c) PFA

Fig. 4.2 Three automata learned from the sample (S, Fr), with $Fr(\varepsilon) = 3, Fr(aa) = Fr(ba) = 2, Fr(bb) = Fr(abb) = Fr(bab) = 1$, and $Fr(a) = Fr(b) = Fr(ab) = Fr(abb) = 0$.

The last step is the standard for building a PFA from a given NFFA. Again, the structure is not modified, but frequencies labeling the transitions and the states are used to calculate

Algorithm 8 Building an RFSA from a simple sample

Input: A simple sample (S, Fr)
Output: An RFSA $\langle \Sigma, Q, I, F, \delta \rangle$

Pref $= PREF(S_+)$ ordered by length-lexicographic order
$Q = I = F = \delta = \emptyset$
$u = \varepsilon$
loop
  if $\exists u' \in Q$ such that $u \simeq^{tr} u'$ **then**
    Pref $=$ Pref $\setminus u\Sigma^*$
  else
    $Q = Q \cup \{u\}$
    if $u \prec^{tr} \varepsilon$ **then**
      $I = I \cup \{u\}$
    end if
    if $u \in S_+$ **then**
      $F = F \cup \{u\}$
    end if
    for $u' \in Q$ and $a \in \Sigma$ **do**
      if $u'a \in$ Pref and $u \prec^{tr} u'a$ **then**
        $\delta = \delta \cup \{\delta(u', a) = u\}$
      end if
      if $ua \in$ Pref and $u' \prec^{tr} ua$ **then**
        $\delta = \delta \cup \{\delta(u, a) = u')\}$
      end if
    end for
  end if
  if u is the last string of Pref or $\langle \Sigma, Q, I, F, \delta \rangle$ is consistent with S **then**
    exit loop
  else
    $u =$ next string in Pref
  end if
end loop
return $\langle \Sigma, Q, I, F, \delta \rangle$

Algorithm 9 Building an NFFA from an RFSA

Input: A RFSA $\langle \Sigma, Q, I, F, \delta \rangle$ consistent with a sample (S, Fr)
Output: An NFFA $\langle \Sigma, Q, I_f, F_f, \delta_f \rangle$

$I_f(q) = 0$ for all $q \in Q$
$F_f(q) = 0$ for all $q \in Q$
$\delta_f(q,a) = 0$ for all $q \in Q$ and $a \in \Sigma$.
for $a_1 \cdots a_n \in S_+$ **do**
  compute $Paths(x)$
  for every $\pi = q_0 \ldots q_n \in Paths(x)$ **do**
    $I_f(q_0) = I_f(q_0) + \frac{Fr(x)}{|Paths(x)|}$
    $F_f(q_n) = F_f(q_n) + \frac{Fr(x)}{|Paths(x)|}$
    for $i = 0, i = i+1, i \leq n-1$ **do**
      $\delta_f(q_i, a_{i+1})(q_{i+1}) = \delta_f(q_i, a_{i+1})(q_{i+1}) + \frac{Fr(x)}{|Paths(x)|}$
    end for
  end for
end for
return $\langle \Sigma, Q, I_f, F_f, \delta_f \rangle$

the probabilities. In the algorithm 8, $FREQ(q)$ denotes the number of both strings either passing through a state q or ending in it, and SUM_I denotes the number of strings entering all initial states. For every state q in Q, the probability of being initial state is $\frac{I_f(q)}{SUM_I}$ and of being final state is $\frac{F_f(q)}{FREQ(q)}$, while the probability associated to each transition from q to q' with input a is $\frac{\delta_f(q,a)(q')}{FREQ(q)}$. (same as the last chapter)

The algorithm 9 returns a probabilistic automaton when the input is an NFFA.

Example 8. *The probabilistic automaton A resulting from the NFFA in Fig.4.2b is shown in Fig.4.2c. The support automaton is consistent with the sample* (S, f).

4.3.3 Experimental results

We use the metrics introduced in the Chapter 3 to study the performance of our algorithm for learning probabilistic languages. We selected different sizes of samples independently drawn according to a distribution presented by four different probabilistic automata depicted in Figure 4.3: one DPFA, one PFA, one RFSA, and one PFA that cannot be expressed as a DPFA. First, we generate a set S of size n of strings from the alphabet by length-lexicographic order and assign the probability of each string according to the target automaton. Given a fixed number of total occurrences m, we then calculate the frequency of each string in the sample based on its assigned probability. Note that samples generated in this way need not

Algorithm 10 Building a PFA from an NFFA

Input: An NFFA $\langle \Sigma, Q, I_f, F_f, \delta_f \rangle$
Output: A PFA $\langle \Sigma, Q, I_p, F_p, \delta_p \rangle$

for $q \in Q$ **do**
  $FREQ(q) = F_f(q) + \sum_{a \in \Sigma, q' \in Q} \delta_f(q,a)(q')$
  $F_p(q) = \frac{F_f(q)}{FREQ(q)}$
  for $a \in \Sigma_1, q' \in Q$ **do**
    $\delta_p(q,a)(q') = \frac{\delta_f(q,a)(q')}{FREQ(q)}$
  end for
end for
$SUM_I = \sum_{q \in Q} I_f(q)$
for $q \in Q$ **do**
  $I_p(q) = \frac{I_f(q)}{SUM_I}$
end for
return $\langle \Sigma, Q, I_p, F_p, \delta_p \rangle$

be complete. All target automata we consider have 3 to 5 states, for which we generate a sample set of size $n < 50$ and the total number of occurrences m varying between 10 to 200.

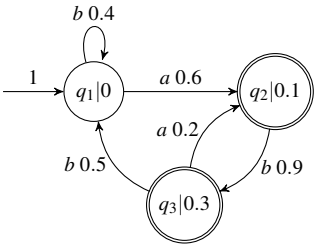
We compare our algorithm to ALERGIA [24] and k-testable algorithms [26]. Contrary to our algorithm presented here, the performance of these other algorithms may be impacted by a parameter setup. For ALERGIA we choose two different parameters $\alpha = 0.9$ and $\alpha = 0.1$. For k-testable algorithms, we set k to be $2, 3, 4$ and 5.

For the case of the DPFA A_1, the distribution found by all algorithms converges concerning the L_2 distance rather quickly towards the original one. The 5-testable algorithm has the highest precision and sensitivity and the smallest L_2 distance, but it needs 19 states to learn an automaton of 3. Our algorithm has the best accuracy and is the only one learning the same structure as the original automata.

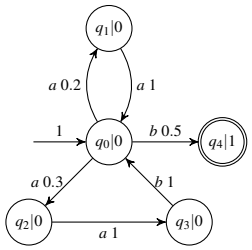
A similar situation happens when learning the RFSA A_3. In this case, our algorithm learns a distribution that cannot be described by any DPFA.

Considering the PFA A_2, our algorithm, ALERGIA, and the 5-testable algorithm outperform all the others. See Figure 4.5. Only our algorithm can learn the same number of states but with a few more transitions. Accuracy is one again. Some errors are introduced because of the fair distribution among non-deterministic transitions.

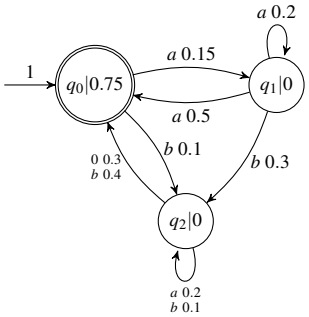
Finally, we considered the PFA A_4 that cannot be expressed by any DPFA and does not have an equivalent RFSA as support. All algorithms cannot learn the same structure as the target automaton. Nevertheless, our algorithm achieves the best performance. The L_2 distance is smallest, precision is highest, sensitivity is second highest, and accuracy is always



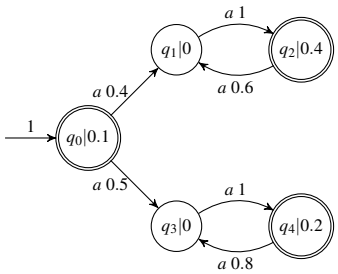
(a) A DPFA A_1



(b) A PFA A_2



(c) A RFSA A_3



(d) A PFA A_4 not equivalent to a DPFA

Fig. 4.3 The four target automata for our experiments

1 (something not true for all other algorithms). Even if we perform better because the RFSA we learn has the same structure as the support of target distribution, our algorithms will never be able to identify it.

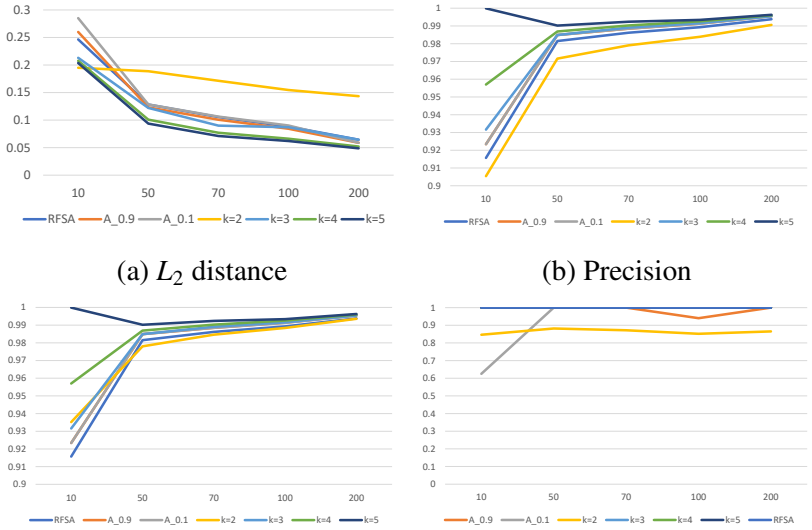
4.3.4 Learning the probabilities with non-linear optimization methods

Once we have learned the structure of the RFSA needed to represent the language underlying 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:

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

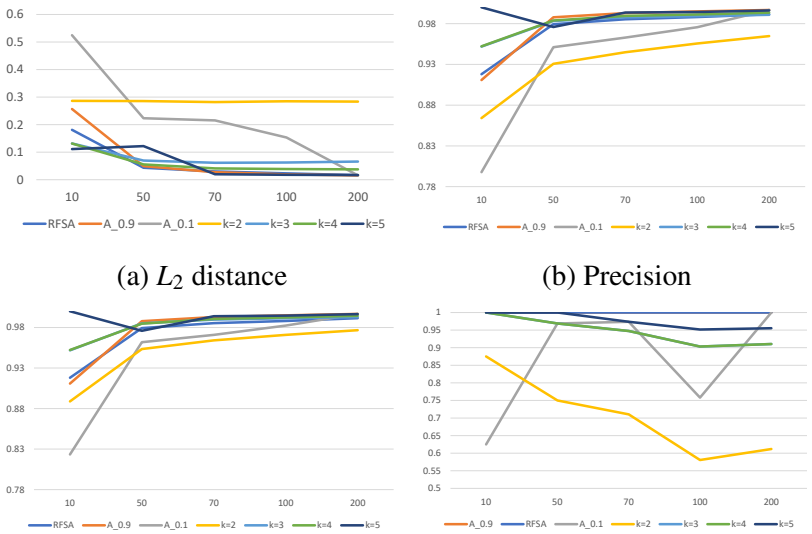
Besides the above structural equations, we have equations depending on the sample and the automaton. For each string $u = a_0 \cdots a_n \in S$ we define $E(u)$ to be the polynomial



(a) L_2 distance (b) Precision

(c) Sensitivity (d) Accuracy

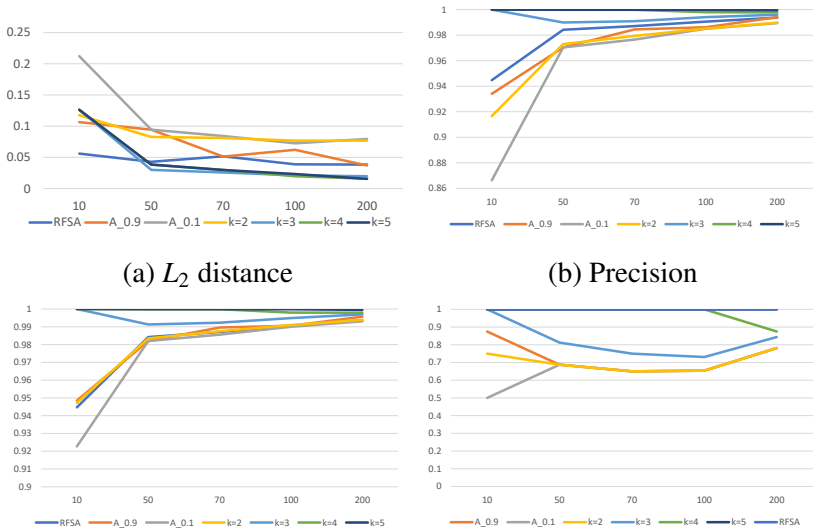
Fig. 4.4 Results of learning A_1



(a) L_2 distance (b) Precision

(c) Sensitivity (d) Accuracy

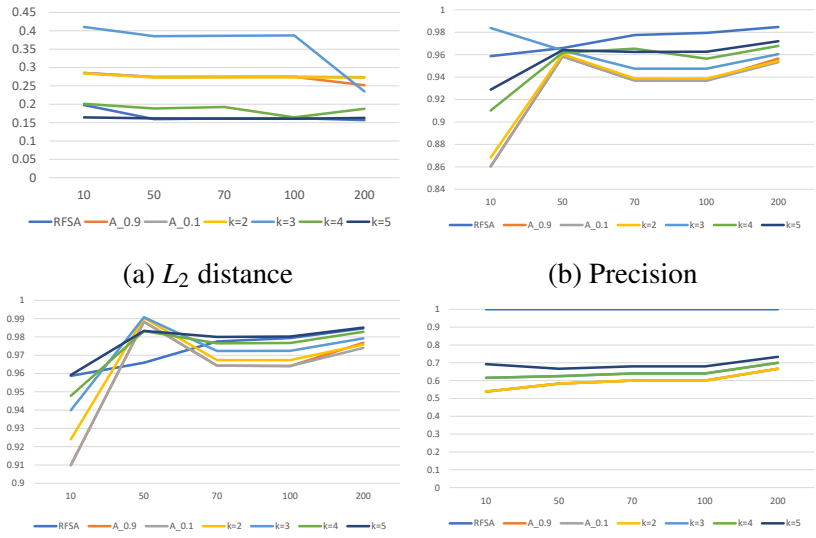
Fig. 4.5 Results of learning A_2



(a) L_2 distance (b) Precision

(c) Sensitivity (d) Accuracy

Fig. 4.6 Results of learning A_3



(a) L_2 distance (b) Precision

(c) Sensitivity (d) Accuracy

Fig. 4.7 Results of learning A_4

equation:

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

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.

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 stem from the sample. We use three different methods to solve the optimization problem with constraints. The first one is via the solver module in SymPy [76]. SymPy is a Python library for solving equations symbolically to find algebraic solutions.

In our experiments below, in most cases, SymPy is not able to find the exact algebraic solution, because the structure of the learned automaton is not always equal to the target one. The second method uses a genetic algorithm (GA). GAs are computational models ideal for searching for optimal solutions by imitating natural evolutionary processes. GAs take individuals in a population and use randomization techniques to guide an efficient search of an encoded parameter space [78]. The third method is based on Sequential Quadratic Programming (SQP), one of the most widely-used methods for solving nonlinear constrained optimization problems [19]. 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 9. *Given the RFSA in 4.8a 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 4.8b. From that we derive the system of equations

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

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 :*

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

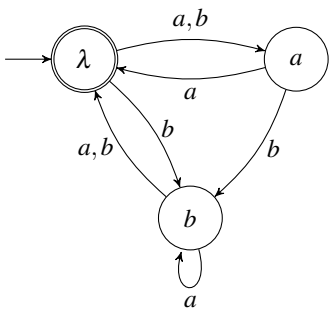
Then we use the GA and SQP to approximate the solution, the results are shown in Figure 4.8c and Figure 4.8d. The learned PFAs approximate to the given sample closely.

4.3.5 Experimental results

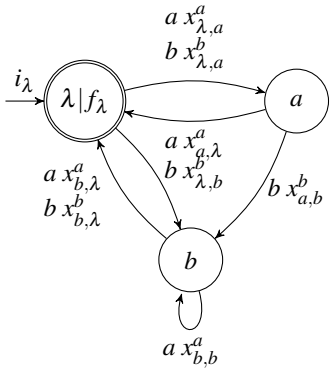
In this section, we summarize some experiments to compare the performance of our new learning method with other existing algorithms using some distributions generated from a random set of PFAs. In particular, we consider ALERGIA, the most well-known probabilistic language learning algorithm, k-testable 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 the best result for each algorithm.

4.3.6 Learning randomly generated probabilistic automata

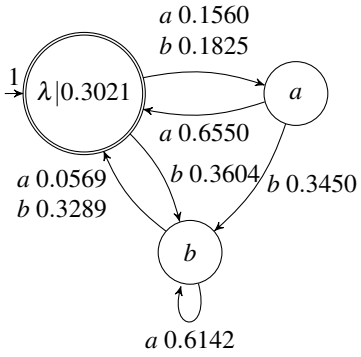
Target automata are generated by a PFA generator according to the number of states, symbols, and transitions for each state. We generate three sets of 20 automata each with 3, 5, and 10



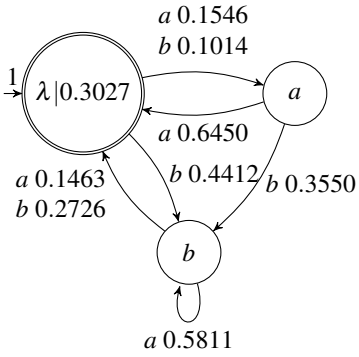
(a) A given NFA.



(b) A PFA with parameters.



(c) A learned PFA by GA.



(d) A learned PFA by SQP.

Fig. 4.8 The RFSA and PFA automata from Example 1.

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 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-symbol 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 [27], 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.

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 is 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 the 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. 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 4.1 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%↓

4.3.7 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 apriori 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 a two-dimensional space with the agent's goal of arriving at target endpoints. 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 endpoints, and δ_p is the set of probabilities of executing one of the four actions in a state. We assume given several sequences of possible consecutive that are obtained, for example, in a training phase, when the agent uniformly selects an action to try to find the target endpoint. Differently from ordinary reinforcement learning methods, we assume that the size and shape of space are not known a-priory, that, moreover, may have insurmountable obstacles.

Training an automaton from a sample is therefore to find the set of states Q, and the right structure where the agent determines 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

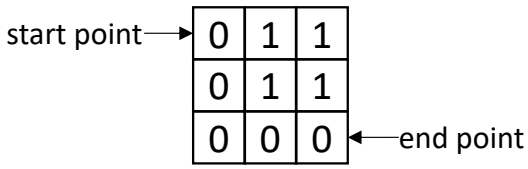


Fig. 4.9 A 3×3 maze, where 0 means available, 1 is an obstacle. $(0,0)$ is the start point, $(2,2)$ is the end point.

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 of the positive and negative samples under a certain agent's moving memoryless strategy.

Example 10. *Fig. 4.9 shows a* 3×3 *maze where* 0 *is a path and* 1 *is an obstacle. The start point is* $(0,0)$*, and the endpoint is* $(2,2)$*. The agent's movement strategy is* $\{U : 0.1, D : 0.4, L : 0.1, R : 0.4\}$*. Traces that reach the endpoint are positive samples, while those that hit walls or obstacles are negative samples. Sample* (S, f) *trace instances:* $f(DDLL) = 0.0256$*,* $f(DUDDLL) = 0.001024$*,* $f(DDLRLL) = 0.001024$*,* $f(DUDUDDLL) = 0.00004108$*,* $f(U) = 0$*,* $f(UUDD) = 0$*,* $f(DLRLU) = 0$*, and* $f(DRLDD) = 0$*.*

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 ones (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 endpoint are our positive samples, with associated frequency (or probability) as calculated based on the probability of each action taken.

To balance the data, we consider it as negative samples all prefixes of the positive one (we assume that the agent once arriving at a target end state stops) and concatenation of prefixes with suffixes that do not occur as positive samples. We use 90 percent of the resulting traces for training, and 10 percent 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 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 methods we consider. In particular, we consider:

The F_1 score [103] is used to measure the method's accuracy. It is computed in terms of both the precision and sensitivity and basically, is the harmonic average of them.

$$F_1 = 2 \cdot \frac{Precision \cdot Sensitivity}{Precision + Sensitivity} \tag{4.1}$$

The optimized precision (OP) [99, 21] 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.

$$OP = Accuracy - \frac{|Specificity - Sensitivity|}{|Specificity + Sensitivity|} \tag{4.2}$$

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 4.2 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 randomly 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 a 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.

4.4 Summary

In this chapter, 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 model an agent's traces in a maze to compare 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-state automata. The other two optimization

Table 4.2 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%↑

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 disadvantages of learning regular languages by RFSA has been turned into an advantage in the context of learning regular distribution. Besides, compared with the k-testable and ALERGIA algorithms, 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 ones. 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.