Automata learning: from probabilistic to quantum

Chu, W.

Chapter 5

Active learning

In this chapter, we explore the concept of active learning and its significance in the context of automata learning. We begin with an introduction to active learning, highlighting its benefits and applications in various domains. Then we focus on the L^* algorithm [2], a famous active learning algorithm for DFA. We present an overview of the L^* algorithm, discussing its key components such as observation tables, membership queries, and equivalence queries. Through an example, we illustrate the process of learning DFA using the L^* algorithm.

By actively selecting informative queries the L^* algorithm has shown to be very flexible, and adaptable to different types of automata maintaining efficiency and accuracy among automata learning techniques [34]. In particular, our focus is on active learning weighted finite automata (WFA), which are automata equipped with weights. We discuss the modifications and adaptations required for the L^* to handle learning of weighted automata effectively. Additionally, we present a detailed example to illustrate the process of learning WFA.

5.1 Active learning

In the previous chapters, our learning process was subject to no control by the learner. At best, we could expect the data to be correctly labeled and compliant with some completeness conditions. However, having control over the data we receive has at least two advantages. First, if we cannot learn even in the most favorable conditions, we can draw negative results for the general case. Second, in some realistic situations, we have the opportunity to choose the data we receive, such as in testing, exploration, or interaction with robots, or when there is an abundance of data and interactive data selection is required for effective learning [70, 53, 59]. As a result, active learning has become a popular approach for maximizing the information gained from the data while minimizing the time and resources required for learning. Consequently, lots of researchers have investigated ways of learning

through more active interactions between the oracle and the learner. The oracle is an abstract machine that knows the target and responds to queries. Typically, the oracle is considered to be flawless and capable of responding to any specific queries even those that a concrete machine cannot handle, thereby solving undecidable problems. The oracle's capacity is established by the learning framework [112, 111].

In certain situations, the oracle may have multiple possible answers, and in such cases, any permissible answer should be allowed. Since the objective of learning is to consider the worst-case scenarios, it is always assumed that the oracle will provide the least informative answer out of all the available answers. This guarantees that, in the learning process, the learner can still effectively learn the target, even under the most unfavorable conditions.

5.1.1 Interacting with the oracle

There are many types of queries one can make to an oracle [4, 34]. Some are natural in the sense that there is at least some application with a biological, mechanical, or cognitive instantiation of these queries; others are defined to build negative results only.

- Membership queries: MQ. A membership query is made by asking the oracle if a string belongs to the target language, to which the oracle responds with either Yes or No. Correction queries are an extension of MQs, where the oracle could return a similar string in the language in the case of a response. Extended membership queries return, for example, the probability of a given string belonging to the language, or the weight assigned by the language to a given string.

- Equivalence queries (weak): WEQ. This approach is used to determine the correctness of a hypothesis about the target language. A hypothesis is proposed to the oracle using some finite representation. The oracle answers with either Yes or No, indicating whether the hypothesis is correct or not.

- Equivalence queries (strong): EQ. This approach is also used to determine the correctness of a hypothesis about the target language. In contrast to the weak query, the oracle provides either a Yes or, in the case of a negative response, a counter-example. A counter-example is a string in the symmetric difference between the target language and the submitted hypothesis.

- Subset queries: SSQ. To evaluate its correctness a hypothesis language is submitted to the oracle. If the hypothesis language is included in the target language, the oracle responds with Yes. However, if the hypothesis language is not included in the

target language, the oracle provides a counter-example that belongs to the hypothesis language but not to the target language.

In the case of equivalence or subset queries, the hypothesis language is submitted via a finite representation, as an automaton for example.

5.1.2 The L* algorithm

The most important active learning algorithm to learn DFAs from membership and equivalence queries is L^* [2]. It triggered a lot of subsequent research on active automata learning, with notable contributions from Bergadano and Varricchio, who used a similar model to learn weighted finite automata [14], and Bollig *et al.* who provided an NFA version of the L^* algorithm [18]. In 2022, Muškardin *et al.* presented AALpy, a library with extensions to the L^* algorithm to efficiently learn deterministic, non-deterministic, and stochastic systems [85].

The general idea behind the L^* algorithm is to use MQs to construct a consistent table that represents a partial DFA. MQs are submitted one after the other to make the table closed and complete. When this is the case, then the table represents a DFA and it is submitted as an EQ. If the hypothesis is correct the learning process terminates, and otherwise the received counter-example is used to update the table. This process is iterated until the oracle confirms via an EQ that the DFA recognizes the target language.

The data structure representing an automaton in the L^* algorithm is the observation table. It consists of two parts: a top part containing rows over a finite set $S \subseteq \Sigma^*$, and a bottom part containing rows over $S \cdot \Sigma$ (where $\cdot$ represents pointwise concatenation). The columns range over a finite set $E \subseteq \Sigma^*$. For each $u \in S \cup S \cdot \Sigma$ and $v \in E$, the corresponding cell in the table contains 1 if and only if $uv \in L$ and it contains 0 if and only if $uv \notin L$, which can be determined using an MQ. Intuitively, each row u provides an approximation of the Myhill–Nerode equivalence class of u for the target language. Rows with the same content are considered members of the same equivalence class. That is, the content in each row informs us whether, when this string u is concatenated with different input strings v, it produces results that belong to the target language. In other words, each row provides information about how string u influences the target language. If two rows have the same content, it signifies that they have similar effects in the target language, and they are considered to belong to the same equivalence class. Recall that the Myhill-Nerode right congruence $\equiv_L$ of L is defined for all strings $u_1, u_2 \in \Sigma^*$ by $u_1 \equiv_L u_2 \Leftrightarrow [\forall v \in \Sigma^*, u_1 v \in L \Leftrightarrow u_2 v \in L]$. The functions *row* and *srow* are used to describe the top and bottom parts of the table, respectively:

- $row : S \to 2^E$

- $row(u)(v) = 1$ when $uv \in L$
- $srow : S \cdot \Sigma \to 2^E$
- $srow(ua)(v) = 1$ when $uav \in L$

Note that the S and $S \cdot \Sigma$ may intersect. For the sake of conciseness, when depicting tables, elements in the intersection are only shown in the top part.

The key idea of the algorithm is that we can reconstruct a hypothesis DFA from the rows of the table. The rows with the same content are considered equivalent. In other words, the state space of the hypothesis DFA corresponds to the set $H = \{row(s) | s \in S\}$. The construction is analogous to that of the minimal DFA from the Myhill-Nerode equivalence. In the L^* algorithm, each state is associated with a specific observation result vector, such as $row(s)$. This observation result vector captures the state's response to input sequences and the equivalence information related to the state. Therefore, $row(s)$ is essentially a representation of the state s, containing information about its behavior under specific input conditions. By comparing the observation result vectors of different states, the L^* algorithm can deduce equivalence relationships between states and subsequently construct a model of the automaton. The initial state is $row(\lambda)$. Further, the λ column determines whether a state is accepting: s is accepting whenever $row(s)(\lambda) = 1$. A state s advances on a transition with $a \in \Sigma$ to the state sa given by $srow(sa)$. This describes the progression of a transition within an automaton. It means that when a specific input a is encountered while in state s, it leads to a new state defined by the function $srow(sa)$. For this hypothesis automaton to be well-defined, λ must be in S and E, and the table must satisfy two properties:

- Closedness: the table is considered closed if, for every t in S and a in Σ, there exist $s \in S$ such that $row(s) = srow(ta)$. In other words, closedness ensures that every transition in the table leads to a state in the hypothesis.
- Consistency: the table is considered consistent if, for any s_1 and s_2 in S such that $row(s_1) = row(s_2)$, we have $srow(s_1a) = srow(s_2a)$ for all a in Σ. In other words, consistency ensures no ambiguities in determining the transitions.

The algorithm iteratively updates the sets S and E to ensure the properties of closedness and consistency. It constructs a hypothesis based on the updated table, submits it as an equivalence query, and refines the hypothesis based on any counterexample provided by the oracle. This iterative process continues until the hypothesis is correct.

To be more specific, we illustrate the L^* algorithm in two procedures: Algorithm 11 ensures a table is closed and consistent, and Algorithm 12 handles the learning iterations.

The latter algorithm works in the following manner: Initially, the table is constructed with $S = E = \{\lambda\}$ to ensure closure and consistency (line 1). Then, if the corresponding hypothesis $H_{(S,E)}$ is found to be incorrect via an EQ that yields a counterexample $c \in \Sigma^*$ (line 2), the prefixes of c are added to S (line 3). The table is updated to maintain closure and consistency (line 4). This iterative process continues until an EQ yields a positive answer, at which point the hypothesis is returned (line 6).

Algorithm 11 A closed and consistent table

Input: S, E
Output: S, E
1: **Function** $Fix(S,E)$
2: **while** (S,E) is not closed or not consistent **do**
3: **if** (S,E) is not closed **then**
4: find $s \in S$, $a \in \Sigma$ such that $\forall t \in S.\ srow(sa) \neq row(t)$
5: $S = S \cup \{sa\}$
6: **else if** (S,E) is not consistent **then**
7: find $s_1, s_2 \in S$, $a \in \Sigma$ and $e \in E$ such that
8: $row(s_1) = row(s_2)$ and $srow(s_1a)(e) \neq srow(s_2a)(e)$
9: $E = E \cup \{ae\}$
10: **end if**
11: **end while**
12: **return** (S,E)

Algorithm 12 L^* algorithm

Input: S, E
Output: $H_{(S,E)}$
1: $S, E = Fix(\{\lambda\}, \{\lambda\})$
2: **while** $EQ(H_{(S,E)}) = c \in \Sigma^*$ **do**
3: $S = S \cup PREF(c)$
4: $S, E = Fix(S,E)$
5: **end while**
6: **return** $H_{(S,E)}$

Let us consider an example run of the L^* algorithm to further illustrate its usage.

Example 11. *Given a target language $L = \{w \in a^* || w| \neq 1\}$, we run the algorithm. Initially, $S = E = \{\lambda\}$. We construct the observation table as shown in Table 5.1a. However, this table is not closed because the a row, which has a 0 in the only column, does not appear in the top part of the table. To fix this, we ask a membership query for string aa then add the word a to the set S. Now Table 5.1b is both closed and consistent. Therefore, we construct*

the hypothesis shown in Figure 5.1a and submit an equivalence query. The response from the oracle is negative and returns the counterexample aaa which should have been accepted as it is part of the language L. To resolve this, we add all its prefixes aa and aaa to the set S. The resulting Table 5.1c is closed but not consistent because both the rows λ and aa have value 1*, but their extensions a and aaa differ. Therefore, we prepend the continuation a to the column λ, where they differ, and add a to E. This distinguishes row(λ) from row(aa), as shown in the Table 5.1d. The table is now closed and consistent, and the resulting hypothesis automaton in Figure 5.1b is correct.*

Table 5.1 Example run of $L = \{w \in a^* \mid |w| \neq 1\}$.

(a)

	λ
λ	1
a	0

(b)

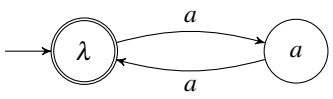
	λ
λ	1
a	0
aa	1

(c)

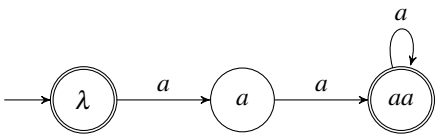
	λ
λ	1
a	0
aa	1
aaa	1
$aaaa$	1

(d)

	λ	a
λ	1	0
a	0	1
aa	1	1
aaa	1	1
$aaaa$	1	1



(a) The first hypothesis.



(b) The correct hypothesis.

Fig. 5.1 Figures depicting two hypotheses

5.2 Active learning weighted finite automata

Our goal is to learn probabilistic automata using active learning. To this end, we first recall the extension of the L^* algorithm for learning weighted finite automata (WFA) [9] so that we can combine it with optimization techniques as in the previous chapters for learning probabilistic automata.

A WFA is a finite automaton that incorporates weights within its transitions and states. These automata find their roots in the mathematical theory of rational power series, and they are widely applied. Notably, image processing and speech recognition have greatly contributed to the popularity and widespread use of weighted automata [61, 79, 81, 82, 42] The significance of these applications, along with several theoretical questions, has strongly

motivated the challenge of learning WFAs. This challenge revolves around the task of constructing a WFA that closely approximates a semiring-valued target function. This involves training the WFA using a finite sample of strings labeled with their corresponding target values. Before exploring the algorithm for learning WFAs, we briefly introduce some fundamental notions related to semirings and weighted automata needed for the discussion in the following sections.

The algebraic structure $(\mathbb{S}, \oplus, \otimes, \bar{0}, \bar{1})$ is a semiring if $(\mathbb{S}, \oplus, \bar{0})$ is a commutative monoid with identity element $\bar{0}$, $(\mathbb{S}, \otimes, \bar{1})$ is a monoid with identity element $\bar{1}$, $\otimes$ distributes over $\oplus$, and $\bar{0}$ is an annihilator for $\otimes$. At its core, a WFA is a finite automaton where the transitions and states are equipped with weights that, like probabilities, allow for a richer representation and modeling capability.

Definition 11. ***Weighted finite automaton*** *A weighted finite automaton (WFA)* $A = \langle Q, \Sigma, I, F, \delta \rangle$ *defined over a semiring* $(\mathbb{S}, \otimes, \oplus, \bar{0}, \bar{1})$ *is:*

- Q *is a finite set of states,*
- Σ *is a finite alphabet,*
- $I : Q \to \mathbb{S}$ *is initial weight,*
- $F : Q \to \mathbb{S}$ *is final weight,*
- $\delta : Q \times \Sigma \to \mathbb{S}^Q$.

We denote by $w[\pi]$ the weight of a path $\pi = q_0 \cdots q_n$ of string $x = a_1 \cdots a_n$ accepted by a WFA A, which is $w[\pi] = \delta(q_0, a_1)(q_1) \cdots \delta(q_{n-1}, a_{n-1})(q_n)$. Let Π denote the set of all accepting paths. The weighted language over $\mathbb{S}$ is a function $\Sigma^* \to \mathbb{S}$. A language accepted by a WFA A is the function $L_A : \Sigma^* \to \mathbb{S}$ given by

$$\forall x \in \Sigma^*,\ W(x) = \bigoplus_{\pi \in \Pi} (I(q_0) \otimes w[\pi] \otimes F(q_n)) \tag{5.1}$$

To simplify notation, we introduce vectors and matrices for weights. $\boldsymbol{\alpha} \in \mathbb{S}^{Q \times 1}$ denotes the vector of initial weights, $\boldsymbol{\beta} \in \mathbb{S}^{Q \times 1}$ denotes the vector of final weights. For any $a \in \Sigma$, let $\boldsymbol{W}_a \in \mathbb{S}^{Q \times Q}$ be the matrix, where $[\boldsymbol{W}_a]_{qq'} = \delta(q, a)(q')$ is the weight of transition labeled with a from q to q'. Then we also write the weight of string x with matrix:

$$\forall x = x_1 \cdots x_n \in \Sigma^*,\ L_A(x) = \boldsymbol{\alpha}^\intercal \boldsymbol{W}_{x_1} \cdots \boldsymbol{W}_{x_n} \boldsymbol{\beta} \tag{5.2}$$

This matrix representation provides a concise way to compute the weight of a string by performing matrix multiplications and taking the inner product with the initial and final weight vectors.

Let us illustrate the concept of WFA and their weight calculations with an example. Consider a WFA A as shown in Figure 5.2a, where weights are from the semiring of natural numbers with the usual multiplication and addition. The detailed computation of the weights assigned to the strings a and ab using the matrix representation is in the example below. The initial weight vector, the transition matrices, and the final weight vector are combined to determine the overall weight of a string.

Example 12. *Given a WFA A shown in Fig. 5.2a the weights assigned to the strings a and ab are, respectively:*

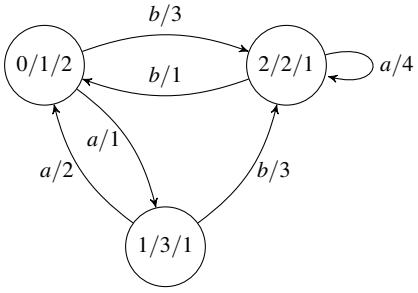
$$\begin{aligned} L_A(a) &= \boldsymbol{\alpha}^\intercal \boldsymbol{W}_a \boldsymbol{\beta} \\ &= \begin{pmatrix} 1 & 3 & 2 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 4 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix} \\ &= 1 \times 1 \times 1 + 3 \times 2 \times 2 + 2 \times 4 \times 1 \\ &= 21. \end{aligned} \tag{5.3}$$

$$\begin{aligned} L_A(ab) &= \boldsymbol{\alpha}^\intercal \boldsymbol{W}_a \boldsymbol{W}_b \boldsymbol{\beta} \\ &= \begin{pmatrix} 1 & 3 & 2 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 4 \end{pmatrix} \begin{pmatrix} 0 & 0 & 3 \\ 0 & 0 & 3 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix} \\ &= 1 \times 1 \times 3 \times 1 + 3 \times 2 \times 3 \times 1 + 2 \times 4 \times 1 \times 2 \\ &= 27. \end{aligned} \tag{5.4}$$

For a given field $\mathbb{S}$, a function $f : \Sigma^* \to \mathbb{S}$ is said to be rational if it can be represented by a weighted finite automaton A [102, 65], meaning that $L_A = f$. Rational functions are strictly related to Hankel matrices.

Definition 12. ***Hankel matrix of a function***. *The Hankel matrix $\boldsymbol{H}_f$ of a function $f : \Sigma^* \to \mathbb{S}$ is defined by $\boldsymbol{H}_f(u,v) = f(uv)$, for all $u, v \in \Sigma^*$.*

The following theorem introduces an interesting relationship between the Hankel matrix of a function and WFA [43].



(a) Example of WFA A.

$$\boldsymbol{\alpha} = \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}, \boldsymbol{W}_a = \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 0 & 0 & 4 \end{pmatrix}$$

$$\boldsymbol{\beta} = \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix}, \boldsymbol{W}_b = \begin{pmatrix} 0 & 0 & 3 \\ 0 & 0 & 3 \\ 1 & 0 & 0 \end{pmatrix}$$

(b) Corresponding initial vector $\boldsymbol{\alpha}$, final vector $\boldsymbol{\beta}$, and transition matrices $\boldsymbol{W}_a$ and $\boldsymbol{W}_b$

Fig. 5.2 An example of a WFA.

Theorem 2. *Given a field $\mathbb{S}$ and a Hankel matrix $\boldsymbol{H}_f$ of the function $f : \Sigma^* \to \mathbb{S}$. The rank of $\boldsymbol{H}_f$ is finite if and only if f is rational. There exists a WFA A generating f with rank$(\boldsymbol{H}_f)$ states and no WFA can represent f with fewer states.*

Proof. Given a WFA A representing f. For any $u, v \in \Sigma^*$ and all final states, we have:

$$f(uv) = L_A(uv) = \boldsymbol{\alpha}^\intercal \boldsymbol{W}_u \boldsymbol{W}_v \boldsymbol{\beta}. \tag{5.5}$$

Where $\boldsymbol{\alpha}^\intercal \boldsymbol{W}_u$ is a row vector in $\mathbb{S}^{1\times Q}$ and $\boldsymbol{W}_v\boldsymbol{\beta}$ is a column vector in $\mathbb{S}^{Q\times 1}$. Let $\boldsymbol{P}$ and $\boldsymbol{S}$ be matrices in $\mathbb{S}^{\Sigma^*\times Q}$ which $\boldsymbol{P}(u,\cdot) = \boldsymbol{\alpha}^\intercal \boldsymbol{W}_u$ for all $u \in \Sigma^*$ and $\boldsymbol{S}(v,\cdot) = (\boldsymbol{W}_v\boldsymbol{\beta})^\intercal$ for all $v \in \Sigma^*$. Then, we could get

$$f(uv) = \boldsymbol{\alpha}^\intercal \boldsymbol{W}_u \boldsymbol{W}_v \boldsymbol{\beta} = (\boldsymbol{P}\boldsymbol{S}^\intercal)(u,v). \tag{5.6}$$

Which means $\boldsymbol{H}_f = \boldsymbol{P}\boldsymbol{S}^\intercal$. Since $\boldsymbol{P}$ and $\boldsymbol{S}$ are in $\mathbb{S}^{\Sigma^*\times Q}$, the rank of $\boldsymbol{H}_f$ is upper bounded by the number of states in A.

Next, assume that the $rank(\boldsymbol{H}_f) = n$. We use $\boldsymbol{H}_f(\cdot, v)$ to denote the $v-$indexed column in $\boldsymbol{H}_f$. Let $(\boldsymbol{H}_f(\cdot,v_1),\cdots,\boldsymbol{H}_f(\cdot,v_n))$ be a basis in $\mathbb{S}^{\Sigma^*\times n}$. Hence, there exist $\beta_1,\cdots,\beta_n \in \mathbb{S}$ such that $\boldsymbol{H}_f(\cdot,\lambda) = \sum_{i=1}^n \beta_i \boldsymbol{H}_f(\cdot,v_i)$. So for $w \in \Sigma^*$, we have $f(w) = \boldsymbol{H}(\lambda,w) = \boldsymbol{H}(w,\lambda) = \sum_n^{i=1} \beta_i \boldsymbol{H}_f(w,v_i)$.

Now for $i \in [1,n]$ and $a \in \Sigma$, there exist $\gamma_{1i}^a \cdots \gamma_{ni}^a$, we have $\boldsymbol{H}_f(\cdot,v_i) = \sum_{j=1}^n (\gamma_{ji}^a)\boldsymbol{H}_f(\cdot,v_j)$. Let $\boldsymbol{A}_a$ be a matrix where $(\boldsymbol{A}_a)_{ji} = \gamma_{ji}^a$. Then for string $w = a_1\cdots a_k \in \Sigma^*$, $\boldsymbol{H}_f(\cdot,wv_i) = \sum_{j=1}^n (\boldsymbol{A}_w)_{ji}\boldsymbol{H}_f(\cdot,v_j)$, where $\boldsymbol{A}_w = \boldsymbol{A}_{a_1}\cdots\boldsymbol{A}_{a_k}$.

Indeed, if $w = w_1w_2 \in \Sigma^*$ and $u \in \Sigma^*$, we have $\boldsymbol{H}_f(u,wv_i) = \boldsymbol{H}_f(uw_1,w_2v_i)$ and:

$$\begin{aligned}
\boldsymbol{H}_f(uw_1) &= \sum_{j=1}^{n} (\boldsymbol{A}_{w_2})_{ji} \boldsymbol{H}_f(uw_1, v_j) \\
&= \sum_{j=1}^{n} (\boldsymbol{A}_{w_2})_{ji} \boldsymbol{H}_f(u, w_1 v_j) \\
&= \sum_{j=1}^{n} (\boldsymbol{A}_{w_2})_{ji} \sum_{k=1}^{n} (\boldsymbol{A}_{w_1})_{kj} \boldsymbol{H}_f(u, v_k) \\
&= \sum_{k=1}^{n} (\boldsymbol{A}_{w_1} \boldsymbol{A}_{w_2})_{ki} \boldsymbol{H}_f(u, v_k).
\end{aligned} \tag{5.7}$$

Furthermore, for any $w = a_1 \cdots a_k \in \Sigma^*$, we could get:

$$\begin{aligned}
f(w) &= \sum_{i=1}^{n} \beta_i \boldsymbol{H}_f(\lambda, w v_i) \\
&= \sum_{j=1} \beta_i \sum_{j=1}^{n} (\boldsymbol{A}_w)_{ji} \boldsymbol{H}_f(\lambda, v_j) \\
&= \boldsymbol{\alpha}^{\intercal} \boldsymbol{A}_{a_1} \cdots \boldsymbol{A}_{a_k} \boldsymbol{\beta},
\end{aligned} \tag{5.8}$$

where $\boldsymbol{\alpha}_j = \boldsymbol{H}_f(\lambda, v_j)$ is a column vector and $\boldsymbol{\beta}_j = \beta_j$ for $j \in [1, n]$ is also a column vector. Therefore, we have that that f can be represented by a WFA with $rank(\boldsymbol{H}_f)$ states. □

5.3 Active learning WFAs from queries

In this section, we recall an algorithm for learning WFAs over an arbitrary field $\mathbb{S}$ [119]. It is an extension of Angluin's algorithm for learning DFAs through membership and equivalence queries, and its applicability can be extended to various other learning problems [11, 12].

The learning scenario for this algorithm corresponds to the active learning scenario introduced and adopted by Angluin for learning unweighted automata [2]. To learn a target rational function $f : \Sigma^* \to \mathbb{S}$, the learner can make two types of queries, both of which are responded to by the oracle:

- Weighted membership queries: $MQ_f s$. the learner asks for the weight value $f(w)$ of the string $w \in \Sigma^*$;
- Weighted equivalence queries: $EQ_f s$. the learner gives the oracle a weighted finite automaton A and the oracle answers yes if $f = L_A$. Otherwise, the oracle gives a counter-example $f(w)$ with $f(w) \neq L_A(w)$.

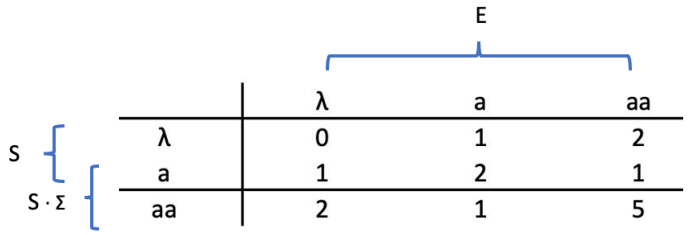


Fig. 5.3 An observation table.

As for the classical case, membership queries are used to build an observation table. An observation table with two parts: rows range over a finite set $S \subseteq \Sigma^*$ and $S \cdot \Sigma$; columns range over a finite set $E \subseteq \Sigma^*$. For each $u \in S \cup S \cdot \Sigma$ and $v \in E$, the cell uv in the table contains the weight $f(uv)$.

- row: $S \to \mathbb{R}^E$ such that $row(u)(v) = f(uv)$,
- srow: $S \cdot \Sigma \to \mathbb{R}^E$ such that $srow(ua)(v) = f(uav)$.

Example 13. *The observation table, depicted in Figure 5.3, illustrates a rational function over the fields of the real numbers that assigns* 0 *to* λ, 1 *to a,* 2 *to aa,* 1 *to aaa and* 5 *to aaaa. Notably, we observe that* $row(a)(a) = srow(a)(a) = 2$. *Since the function row and srow are determined by the weighted language L to be learned, we can refer to the observation table as a pair* (S,E), *leaving L implicitly represented.*

In the weighted setting, a table is closed if for all $x \in S \cdot \Sigma$, there exists $\alpha_s \in \mathbb{S}$ for all $s \in S$ such that:

$$srow(x) = \sum_{s \in S} \alpha_s \cdot row(s). \tag{5.9}$$

If the observation table (S,E) is closed, it represents the WFA $\langle S, \Sigma, I, F, \delta \rangle$ with

- $I : S \to \mathbb{S}$ is the initial weight map defined as $I(\lambda) = 1$ and $I(x) = 0$ for $x \neq \lambda$,
- $F : S \to \mathbb{S}$ is the final weight map defined as $F(s) = row(s)(\lambda)$,
- $\delta : S \times \Sigma \to \mathbb{S}^S$ is the weighted transition weight defined as $\delta(x,\sigma)(y) = \alpha_y$ for all $x\sigma \in S \cdot \Sigma$ and $y \in S$ such that $srow(x\sigma) = \sum_{s \in S} \alpha_s \cdot row(s)$ by Eq. 5.9.

This algorithm for learning a weighted automaton [119] is given below. It starts with both S and E only containing the empty word. The while loop from line 5 to line 7 repeatedly checks whether the current table is closed. If the table is not closed, then it adds new rows. Once the table is closed, we could build a hypothesis according to line 8 to line 14. Then, we

Algorithm 13 Learning algorithm for a WFA over $\mathbb{S}$

Input: strings with weights
Output: a WFA $\langle \Sigma, Q, I, F, \delta \rangle$

$S = \{\lambda\}$
$E = \{\lambda\}$
$I(\lambda) = 1$
while true **do**
  while table is not closed for some $t \in S \cdot \Sigma$ **do**
    $S = S \cup \{t\}$
  end while
  for $s \in S$ **do**
    $Q = Q \cup \{s\}$
    $F(s) = row(s)(\lambda)$
    for $a \in \Sigma$ **do**
      $\delta(x,a)(s) = \alpha_s$
    end for
  end for
  if the equivalence check is not right **then**
    the oracle returns a counterexample w with weight $f(w)$
    $E = E \cup SUFF(w)$
  else
    return the WFA $\langle \Sigma, Q, I, F, \delta \rangle$
  end if
end while

give this hypothesis to the oracle for an equivalence check. If our hypothesis is not correct, the oracle gives a counterexample $w \in \Sigma^*$. The suffixes of w are added to E, then we restart from line 4. Otherwise, if the hypothesis is correct, the algorithm returns the correct WFA (line 19).

Example 14. *We begin with the sets $S = E = \lambda$ and fill the entries on the left table in Figure 5.4 by conducting weighted membership queries for λ and a. However, since the table is not closed, we proceed to construct the right table below by adding the membership result for aa. Upon completing this step, we observe that the table is now closed, as $srow(aa) = 3 \cdot row(a)$. Consequently, we can construct the hypothesis A_1 shown in Figure 5.4 based on this closed table.*

	λ
λ	0
a	1

	λ
λ	0
a	1
aa	3

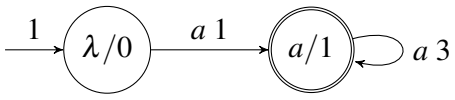


Fig. 5.4 Example of learning a WFA (Part I).

The oracle checks the weighted equivalence and, as it does not hold, provides the counterexample aaa, which the hypothesis automaton above assigns a weight of 9, while the target language assigns it a weight of 7. Consequently, we extend the set E to $E \cup \{a, aa, aaa\}$. The resulting table is displayed in Figure 5.5. Notably, it is closed since $srow(aa) = 3 \cdot row(a) - 2 \cdot row(\lambda)$. Hence, we proceed to construct a new hypothesis shown in Figure 5.5. Then the oracle replies yes, so this is the target WFA.

	λ	a	aa	aaa
λ	0	1	3	7
a	1	3	7	15
aa	3	7	15	31

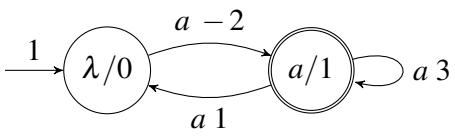


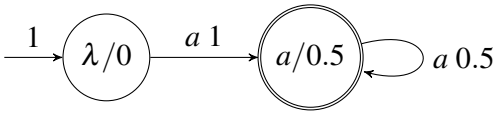
Fig. 5.5 Example of learning a WFA (Part II).

5.4 Active learning probabilistic finite automata

In the previous sections, we discussed the process of active learning a WFA. Next, we show how to extend this methodology to learn a PFA. To begin, we can treat a PFA as a WFA with weights defined in the field of rational numbers. The weighted language L_A of a probabilistic automaton A seen as a weighted automaton is exactly the probability distribution P_A generated

by A. However, when learning a probabilistic automaton A that is seen as a weighted one using the above algorithm, the resulting automaton will not be, in general, a probabilistic one, even if the distribution represented by the learned automaton is a probability distribution.

Example 15. *Starting with a target PFA, as depicted in 5.6a, our initial step involves initializing the sets $S = E = \lambda$. We populate the entries in the table below through weighted membership queries for both the empty string λ and the symbol a. However, since this initial table is not closed, we proceed to expand and complete it by incorporating the membership query result for the string aa. Upon this inclusion, we notice that the table becomes closed. As a result, we can confidently formulate the hypothesis A, as illustrated in 5.6d, based on the information in this table.*



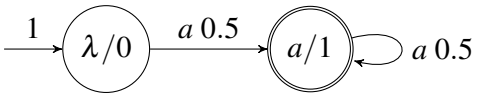
(a) The target PFA.

	λ
λ	0
a	0.5

(b) The initial observation table.

	λ
λ	0
a	0.5
aa	0.25

(c) The closed observation table.



(d) The hypothesis A.

Fig. 5.6 Example of active learning a PFA.

Note that the learned automaton is not a probabilistic one as in both states the sum of the probability of the outgoing transition with that of acceptance is not 1. *Nevertheless, the oracle finds the learned automaton equivalent to the target one, as their generated distributions are identical.*

What is needed is a way to transform a WFA A into a PFA, knowing that the weighted language L_A it recognizes is a regular distribution. To achieve this, we can treat probabilities in transitions and states as variables and employ two types of constraints to enforce the automata to be probabilistic. The first set of constraints is based on the PFA's structure, while the second set encompasses the weight/probabilities of the strings we already know.

Example 16. *Continuing from the previous example, we designate all probabilities of the learned automaton as variables. Subsequently, we establish two sets of equations:*

$$\begin{cases} 0 + x^a_{\lambda,a} &= 1 \\ f_a + x^a_{a,a} &= 1 \end{cases} \quad \begin{cases} x^a_{\lambda,a} f_a &= 0.5 \\ x^a_{\lambda,a} x^a_{a,a} f_a &= 0.25 \end{cases}$$

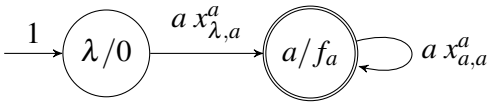


Fig. 5.7 The learned automaton with all probabilities as variables.

Here x's are probabilities on transition and f's probabilities of final states. Subsequently, we solve this system of equations, resulting in a PFA that perfectly matches the target PFA.

5.5 Summary

In this chapter, we introduced active learning and showed how it is applied to learning DFA and weighted automata. In particular, we emphasized the key role played by the Hankel matrix in this learning algorithm. Finally, we briefly showed how to extend this method to learn PFA. We will use these two techniques to learn quantum automata in the next chapter.

A different approach but also grounded in the L^* algorithm is presented in [114] for deterministic Markov Decision Processes (MDPs). The authors give two learning algorithms, an exact learning approach assuming perfect knowledge of system trace distribution, and a sampling-based approach that relaxes this assumption by estimating trace distributions through sampling. It would be interesting to experimentally compare our approach with the precise one proposed in [114].