16 Learning Automata and Languages

This chapter presents an introduction to the problem of learning languages. This is a classical problem explored since the early days of formal language theory and computer science, and there is a very large body of literature dealing with related mathematical questions. In this chapter, we present a brief introduction to this problem and concentrate specifically on the question of learning finite automata, which, by itself, has been a topic investigated in multiple forms by thousands of technical papers. We will examine two broad frameworks for learning automata, and for each, we will present an algorithm. In particular, we describe an algorithm for learning automata in which the learner has access to several types of query, and we discuss an algorithm for identifying a sub-class of the family of automata in the limit.

16.1 Introduction

Learning languages is one of the earliest problems discussed in linguistics and computer science. It has been prompted by the remarkable faculty of humans to learn natural languages. Humans are capable of uttering well-formed new sentences at an early age, after having been exposed only to finitely many sentences. Moreover, even at an early age, they can make accurate judgments of grammaticality for new sentences.

In computer science, the problem of learning languages is directly related to that of learning the representation of the computational device generating a language. Thus, for example, learning regular languages is equivalent to learning finite automata, or learning context-free languages or context-free grammars is equivalent to learning pushdown automata.

There are several reasons for examining specifically the problem of learning finite automata. Automata provide natural modeling representations in a variety of different domains including systems, networking, image processing, text and speech

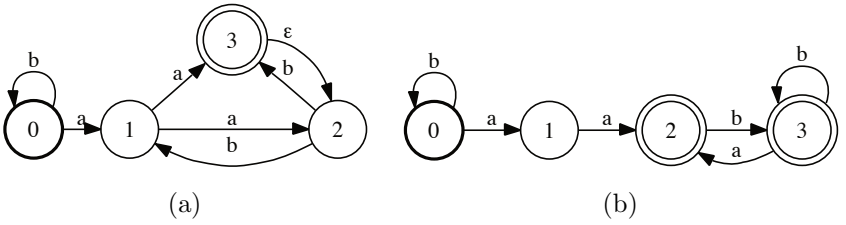

Figure 16.1
(a) A graphical representation of a finite automaton. (b) Equivalent (minimal) deterministic automaton.

processing, logic and many others. Automata can also serve as simple or efficient approximations for more complex devices. For example, in natural language processing, they can be used to approximate context-free languages. When it is possible, learning automata is often efficient, though, as we shall see, the problem is hard in a number of natural scenarios. Thus, learning more complex devices or languages is even harder.

We consider two general learning frameworks: the model of *efficient exact learning* and the model of *identification in the limit*. For each of these models, we briefly discuss the problem of learning automata and describe an algorithm.

We first give a brief review of some basic automata definitions and algorithms, then discuss the problem of efficient exact learning of automata and that of the identification in the limit.

16.2 Finite automata

We will denote by Σ a finite alphabet. The length of a string $x \in \Sigma^*$ over that alphabet is denoted by $|x|$. The *empty string* is denoted by ϵ, thus $|\epsilon| = 0$. For any string $x = x_1 \cdots x_k \in \Sigma^*$ of length $k \geq 0$, we denote by $x[j] = x_1 \cdots x_j$ its prefix of length $j \leq k$ and define $x[0]$ as ϵ.

Finite automata are labeled directed graphs equipped with initial and final states. The following gives a formal definition of these devices.

Definition 16.1 (Finite automata) *A* finite automaton *A is a 5-tuple (Σ, Q, I, F, E) where Σ is a finite alphabet, Q a finite set of states, $I \subseteq Q$ a set of initial states, $F \subseteq Q$ a set of final states, and $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$ a finite set of* transitions.

Figure 16.1a shows a simple example of a finite automaton. States are represented by circles. A bold circle indicates an initial state, a double circle a final state. Each transition is represented by an arrow from its origin state to its destination state with its label in $\Sigma \cup \{\epsilon\}$.

A path from an initial state to a final state is said to be an *accepting path*. An automaton is said to be *trim* if all of its states are accessible from an initial state and admit a path to a final state, that is, if all of its states lie on an accepting path. A string $x \in \Sigma^*$ is *accepted* by an automaton A iff x labels an accepting path. For convenience, we will say that $x \in \Sigma^*$ is *rejected* by A when it is not accepted. The set of all strings accepted by A defines the *language accepted by* A denoted by $L(A)$. The class of languages accepted by finite automata coincides with the family of *regular languages*, that is, languages that can be described by *regular expressions*.

Any finite automaton admits an equivalent automaton with no ϵ-*transition*, that is, no transition labeled with the empty string: there exists a general ϵ-removal algorithm that takes as input an automaton and returns an equivalent automaton with no ϵ-transition.

An automaton with no ϵ-*transition* is said to be *deterministic* if it admits a unique initial state and if no two transitions sharing the same label leave any given state. A deterministic finite automaton is often referred to by the acronym *DFA*, while the acronym *NFA* is used for arbitrary automata, that is, non-deterministic finite automata. Any NFA admits an equivalent DFA: there exists a general (exponential-time) *determinization* algorithm that takes as input an NFA with no ϵ-transition and returns an equivalent DFA. Thus, the class of languages accepted by DFAs coincides with that of the languages accepted by NFAs, that is regular languages. For any string $x \in \Sigma^*$ and DFA A, we denote by $A(x)$ the state reached in A when reading x from its unique initial state.

A DFA is said to be *minimal* if it admits no equivalent deterministic automaton with a smaller number of states. There exists a general *minimization* algorithm taking as input a deterministic automaton and returning a minimal one that runs in $O(|E| \log |Q|)$. When the input DFA is *acyclic*, that is when it admits no path forming a cycle, it can be minimized in linear time $O(|Q| + |E|)$. Figure 16.1b shows the minimal DFA equivalent to the NFA of figure 16.1a.

16.3 Efficient exact learning

In the *efficient exact learning* framework, the problem consists of identifying a target concept c from a finite set of examples in time polynomial in the size of the representation of the concept and in an upper bound on the size of the representation of an example. Unlike the PAC-learning framework, in this model, there is no stochastic assumption, instances are not assumed to be drawn according to some unknown distribution. Furthermore, the objective is to identify the target concept

exactly, without any approximation. A concept class $\mathcal{C}$ is said to be efficiently exactly learnable if there is an algorithm for efficient exact learning of any $c \in \mathcal{C}$.

We will consider two different scenarios within the framework of efficiently exact learning: a *passive* and an *active learning* scenario. The passive learning scenario is similar to the standard supervised learning scenario discussed in previous chapters but without any stochastic assumption: the learning algorithm *passively* receives data instances as in the PAC model and returns a hypothesis, but here, instances are not assumed to be drawn from any distribution. In the active learning scenario, the learner *actively* participates in the selection of the training samples by using various types of queries that we will describe. In both cases, we will focus more specifically on the problem of learning automata.

16.3.1 Passive learning

The problem of learning finite automata in this scenario is known as the *minimum consistent DFA learning problem* . It can be formulated as follows: the learner receives a finite sample $S = ((x_1, y_1), \ldots, (x_m, y_m))$ with $x_i \in \Sigma^*$ and $y_i \in \{-1, +1\}$ for any $i \in [m]$. If $y_i = +1$, then x_i is an accepted string, otherwise it is rejected. The problem consists of using this sample to learn the smallest DFA A *consistent* with S, that is the automaton with the smallest number of states that accepts the strings of S with label $+1$ and rejects those with label -1. Note that seeking the smallest DFA consistent with S can be viewed as following Occam's razor principle.

The problem just described is distinct from the standard minimization of DFAs. A minimal DFA accepting exactly the strings of S labeled positively may not have the smallest number of states: in general there may be DFAs with fewer states accepting a superset of these strings and rejecting the negatively labeled sample strings. For example, in the simple case $S = ((a, +1), (b, -1))$, a minimal deterministic automaton accepting the unique positively labeled string a or the unique negatively labeled string b admits two states. However, the deterministic automaton accepting the language a^* accepts a and rejects b and has only one state.

Passive learning of finite automata turns out to be a computationally hard problem. The following theorems present several negative results known for this problem.

Theorem 16.2 *The problem of finding the smallest deterministic automaton consistent with a set of accepted or rejected strings is NP-complete.*

Hardness results are known even for a polynomial approximation, as stated by the following theorem.

Theorem 16.3 *If $P \neq NP$, then, no polynomial-time algorithm can be guaranteed to find a DFA consistent with a set of accepted or rejected strings of size smaller than*

a polynomial function of the smallest consistent DFA, even when the alphabet is reduced to just two elements.

Other strong negative results are known for passive learning of finite automata under various cryptographic assumptions.

These negative results for passive learning invite us to consider alternative learning scenarios for finite automata. The next section describes a scenario leading to more positive results where the learner can actively participate in the data selection process using various types of queries.

16.3.2 Learning with queries

The model of *learning with queries* corresponds to that of a (minimal) teacher or oracle and an active learner. In this model, the learner can make the following two types of queries to which an oracle responds:

- *membership queries*: the learner requests the target label $f(x) \in \{-1, +1\}$ of an instance x and receives that label;
- *equivalence queries*: the learner conjectures hypothesis h; it receives the response yes if $h = f$, a counter-example otherwise.

We will say that a concept class $\mathcal{C}$ is *efficiently exactly learnable with membership and equivalence queries* when it is efficiently exactly learnable within this model.

This model is not realistic, since no such oracle is typically available in practice. Nevertheless, it provides a natural framework, which, as we shall see, leads to positive results. Note also that for this model to be significant, equivalence must be computationally testable. This would not be the case for some concept classes such as that of *context-free grammars*, for example, for which the equivalence problem is undecidable. In fact, equivalence must be further efficiently testable, otherwise the response to the learner cannot be supplied in a reasonable amount of time.[21]

Efficient exact learning within this model of learning with queries implies the following variant of PAC-learning: we will say that a concept class $\mathcal{C}$ is *PAC-learnable with membership queries* if it is PAC-learnable by an algorithm that has access to a polynomial number of membership queries.

Theorem 16.4 *Let $\mathcal{C}$ be a concept class that is efficiently exactly learnable with membership and equivalence queries, then $\mathcal{C}$ is PAC-learnable using membership queries.*

Proof: Let $\mathcal{A}$ be an algorithm for efficiently exactly learning $\mathcal{C}$ using membership and equivalence queries. Fix $\epsilon, \delta > 0$. We replace in the execution of $\mathcal{A}$ for learning

[21] For a human oracle, answering membership queries may also become very hard in some cases when the queries are near the class boundaries. This may also make the model difficult to adopt in practice.

target $c \in \mathcal{C}$, each equivalence query by a test of the current hypothesis on a polynomial number of labeled examples. Let $\mathcal{D}$ be the distribution according to which points are drawn. To simulate the tth equivalence query, we draw $m_t = \frac{1}{\epsilon}(\log \frac{1}{\delta} + t \log 2)$ points i.i.d. according to $\mathcal{D}$ to test the current hypothesis h_t. If h_t is consistent with all of these points, then the algorithm stops and returns h_t. Otherwise, one of the points drawn does not belong to h_t, which provides a counter-example.

Since $\mathcal{A}$ learns c exactly, it makes at most T equivalence queries, where T is polynomial in the size of the representation of the target concept and in an upper bound on the size of the representation of an example. Thus, if no equivalence query is positively responded by the simulation, the algorithm will terminate after T equivalence queries and return the correct concept c. Otherwise, the algorithm stops at the first equivalence query positively responded by the simulation. The hypothesis it returns is not an ϵ-approximation only if the equivalence query stopping the algorithm is incorrectly responded positively. By the union bound, since for any fixed $t \in [T]$, $\mathbb{P}[R(h_t) > \epsilon] \leq (1-\epsilon)^{m_t}$, the probability that for some $t \in [T]$, $R(h_t) > \epsilon$ can be bounded as follows:

$$\begin{aligned}\mathbb{P}[\exists t \in [T] \colon R(h_t) > \epsilon] &\leq \sum_{t=1}^{T} \mathbb{P}[R(h_t) > \epsilon] \\ &\leq \sum_{t=1}^{T} (1-\epsilon)^{m_t} \leq \sum_{t=1}^{T} e^{-m_t \epsilon} \leq \sum_{t=1}^{T} \frac{\delta}{2^t} \leq \sum_{t=1}^{+\infty} \frac{\delta}{2^t} = \delta.\end{aligned}$$

Thus, with probability at least $1-\delta$, the hypothesis returned by the algorithm is an ϵ-approximation. Finally, the maximum number of points drawn is $\sum_{t=1}^{T} m_t = \frac{1}{\epsilon}(T \log \frac{1}{\delta} + \frac{T(T+1)}{2} \log 2)$, which is polynomial in $1/\epsilon$, $1/\delta$, and T. Since the rest of the computational cost of $\mathcal{A}$ is also polynomial by assumption, this proves the PAC-learning of $\mathcal{C}$. □

16.3.3 Learning automata with queries

In this section, we describe an algorithm for efficient exact learning of DFAs with membership and equivalence queries. We will denote by A the target DFA and by $\widehat{A}$ the DFA that is the current hypothesis of the algorithm. For the discussion of the algorithm, we assume without loss of generality that A is a minimal DFA.

The algorithm uses two sets of strings, U and V. U is a set of *access strings*: reading an access string $u \in U$ from the initial state of A leads to a state $A(u)$. The algorithm ensures that the states $A(u)$, $u \in U$, are all distinct. To do so, it uses a set V of *distinguishing strings*. Since A is minimal, for two distinct states q and q' of A, there must exist at least one string that leads to a final state from q and not from q', or vice versa. That string helps *distinguish* q and q'. The set of strings V

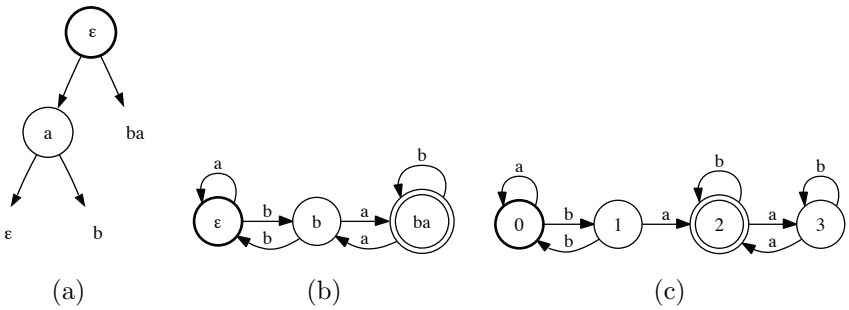

Figure 16.2
(a) Classification tree T, with $U = \{\epsilon, b, ba\}$ and $V = \{\epsilon, a\}$. (b) Current automaton $\widehat{A}$ constructed using T. (c) Target automaton A.

help distinguish any pair of access strings in U. They define in fact a partition of all strings of Σ^*.

The objective of the algorithm is to find at each iteration a new access string distinguished from all previous ones, ultimately obtaining a number of access strings equal to the number of states of A. It can then identify each state $A(u)$ of A with its access string u. To find the destination state of the transition labeled with $a \in \Sigma$ leaving state u, it suffices to determine, using the partition induced by V the access string u' that belongs to the same equivalence class as ua. The finality of each state can be determined in a similar way.

Both sets U and V are maintained by the algorithm via a binary decision tree T similar to those presented in chapter 9. Figure 16.2a shows an example. T defines the partition of all strings induced by the distinguishing strings V. The leaves of T are each labeled with a distinct $u \in U$ and its internal nodes with a string $v \in V$. The decision tree question defined by $v \in V$, given a string $x \in \Sigma^*$, is whether xv is accepted by A, which is determined via a membership query. If accepted, x is assigned to right sub-tree, otherwise to the left sub-tree, and the same is applied recursively with the sub-trees until a leaf is reached. We denote by $T(x)$ the label of the leaf reached. For example, for the tree T of figure 16.2a and target automaton A of figure 16.2c, $T(baa) = b$ since baa is not accepted by A (root question) and $baaa$ is (question at node a). At its initialization step, the algorithm ensures that the root node is labeled with ϵ, which is convenient to check the finality of the strings.

The tentative hypothesis DFA $\widehat{A}$ can be constructed from T as follows. We denote by ConstructAutomaton() the corresponding function. A distinct state $\widehat{A}(u)$ is created for each leaf $u \in U$. The finality of a state $\widehat{A}(u)$ is determined based on the sub-tree of the root node that u belongs to: $\widehat{A}(u)$ is made final iff u belongs

```
QueryLearnAutomata()
 1  t ← MembershipQuery(ε)
 2  T ← T_0
 3  Â ← A_0
 4  while (EquivalenceQuery(Â) ≠ true) do
 5      x ← CounterExample()
 6      if (T = T_0) then
 7          T ← T_1 ▷ nil replaced with x.
 8      else j ← argmin_k A(x[k]) ≠_T Â(x[k])
 9          Split(Â(x[j − 1]))
10      Â ← ConstructAutomaton(T)
11  return Â
```

Figure 16.3
Algorithm for learning automata with membership and equivalence queries. A_0 is a single-state automaton with self-loops labeled with all $a \in \Sigma$. That state is initial. It is final iff $t = \text{true}$. T_0 is a tree with root node labeled with ϵ and two leaves, one labeled with ϵ, the other with nil. the right leaf is labeled with ϵ labels iff $t = \text{true}$. T_1 is the tree obtained from T_0 by replacing nil with x.

to the right sub-tree that is iff $u = \epsilon u$ is accepted by A. The destination of the transition labeled with $a \in \Sigma$ leaving state $\widehat{A}(u)$ is the state $\widehat{A}(v)$ where $v = T(ua)$. Figure 16.2b shows the DFA $\widehat{A}$ constructed from the decision tree of figure 16.2a. For convenience, for any $x \in \Sigma^*$, we denote by $U(\widehat{A}(x))$ the access string identifying state $\widehat{A}(x)$.

Figure 16.3 shows the pseudocode of the algorithm. The initialization steps at lines 1–3 construct a tree T with a single internal node labeled with ϵ and one leaf string labeled with ϵ, the other left undetermined and labeled with nil. They also define a tentative DFA $\widehat{A}$ with a single state with self-loops labeled with all elements of the alphabet. That single state is an initial state. It is made a final state only if ϵ is accepted by the target DFA A, which is determined via the membership query of line 1.

At each iteration of the loop of lines 4–11, an equivalence query is used. If $\widehat{A}$ is not equivalent to A, then a counter-example string x is received (line 5). If T is the tree constructed in the initialization step, then the leaf labeled with nil is replaced with x (lines 6–7). Otherwise, since x is a counter-example, states $A(x)$

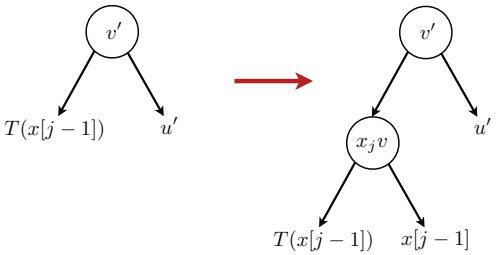

Figure 16.4
Illustration of the splitting procedure SPLIT($\widehat{A}(x[j-1])$).

and $\widehat{A}(x)$ have a different finality; thus, the string x defining $A(x)$ and the access string $U(\widehat{A}(x))$ are assigned to different equivalence classes by T. Thus, there exists a smallest j such that $A(x[j])$ and $\widehat{A}(x[j])$ are not equivalent, that is, such that the prefix $x[j]$ of x and the access string $U(\widehat{A}(x[j]))$ are assigned to different leaves by T. j cannot be 0 since the initialization ensures that $\widehat{A}(\epsilon)$ is an initial state and has the same finality as the initial state $A(\epsilon)$ of A. The equivalence of $A(x[j])$ and $\widehat{A}(x[j])$ is tested by checking the equality of $T(x[j])$ and $T(U(\widehat{A}(x[j])))$, which can be both determined using the tree T and membership queries (line 8).

Now, by definition, $A(x[j-1])$ and $\widehat{A}(x[j-1])$ are equivalent, that is T assigns $x[j-1]$ to the leaf labeled with $U(\widehat{A}(x[j-1]))$. But, $x[j-1]$ and $U(\widehat{A}(x[j-1]))$ must be distinguished since $A(x[j-1])$ and $\widehat{A}(x[j-1])$ admit transitions labeled with the same label x_j to two non-equivalent states. Let v be a distinguishing string for $A(x[j])$ and $\widehat{A}(x[j])$. v can be obtained as the least common ancestor of the leaves labeled with $x[j]$ and $U(\widehat{A}(x[j]))$. To distinguish $x[j-1]$ and $U(\widehat{A}(x[j-1]))$, it suffices to split the leaf of T labeled with $T(x[j-1])$ to create an internal node $x_j v$ dominating a leaf labeled with $x[j-1]$ and another one labeled with $T(x[j-1])$ (line 9). Figure 16.4 illustrates this construction. Thus, this provides a new access string $x[j-1]$ which, by construction, is distinguished from $U(\widehat{A}(x[j-1]))$ and all other access strings.

Thus, the number of access strings (or states of $\widehat{A}$) increases by one at each iteration of the loop. When it reaches the number of states of A, all states of A are of the form $A(u)$ for a distinct $u \in U$. A and $\widehat{A}$ have then the same number of states and in fact $A = \widehat{A}$. Indeed, let $(A(u), a, A(u'))$ be a transition in A, then by definition the equality $A(ua) = A(u')$ holds. The tree T defines a partition of all strings in terms of their distinguishing strings in A. Since in A, ua and u' lead to the same state, they are assigned to the same leaf by T, that is, the leaf labeled with u'. The destination of the transition from $\widehat{A}(u)$ with label a is found by CONSTRUCTAUTOMATON() by determining the leaf in T assigned to ua, that is, u'. Thus, by construction, the same transition $(\widehat{A}(u), a, \widehat{A}(u'))$ is created in $\widehat{A}$.

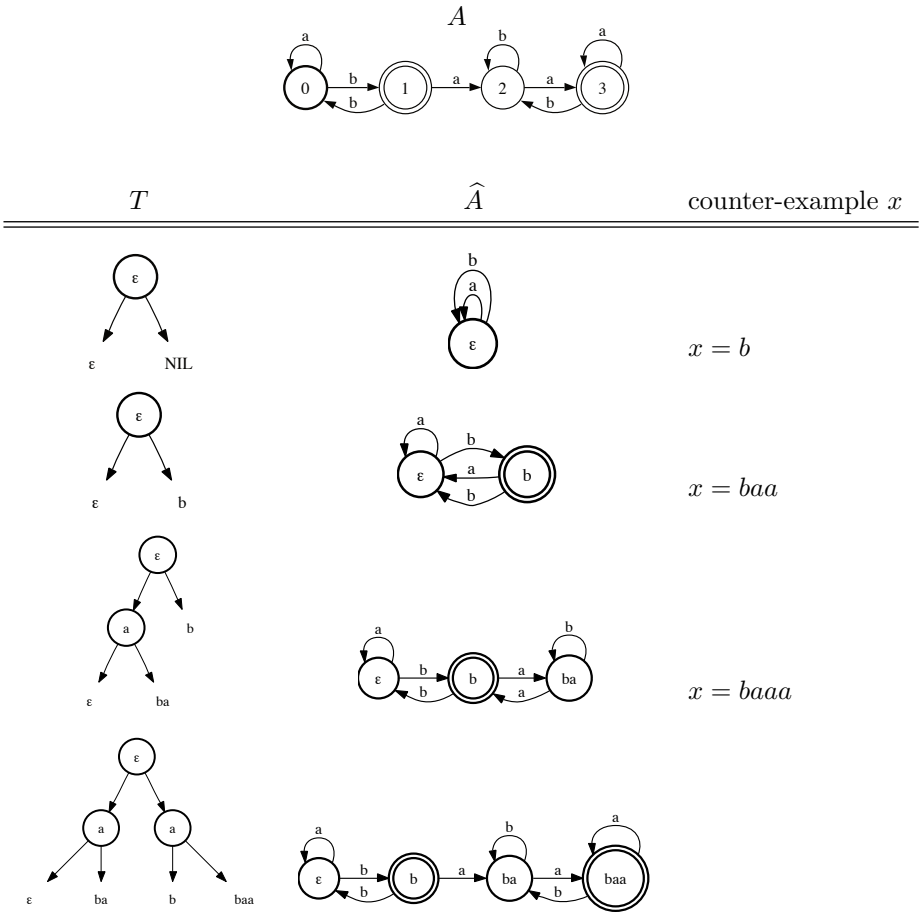

Figure 16.5
Illustration of the execution of Algorithm QUERYLEARNAUTOMATA() for the target automaton A. Each line shows the current decision tree T and the tentative DFA $\widehat{A}$ constructed using T. When $\widehat{A}$ is not equivalent to A, the learner receives a counter-example x indicated in the third column.

Also, a state $A(u)$ of A is final iff u accepted by A that is iff u is assigned to the right sub-tree of the root node by T, which is the criterion determining the finality of $\widehat{A}(u)$. Thus, the automata A and $\widehat{A}$ coincide.

The following is the analysis of the running-time complexity of the algorithm. At each iteration, one new distinguished access string is found associated to a distinct state of A, thus, at most $|A|$ states are created. For each counter-example x, at most $|x|$ tree operations are performed. Constructing $\widehat{A}$ requires $O(|\Sigma||A|)$ tree operations. The cost of a tree operation is $O(|A|)$ since it consists of at most $|A|$

membership queries. Thus, the overall complexity of the algorithm is in $O(|\Sigma||A|^2 + n|A|)$, where n is the maximum length of a counter-example. Note that this analysis assumes that equivalence and membership queries are made in constant time.

Our analysis shows the following result.

Theorem 16.5 (Learning DFAs with queries) *The class of all DFAs is efficiently exactly learnable using membership and equivalence queries.*

Figure 16.5 illustrates a full execution of the algorithm in a specific case. In the next section, we examine a different learning scenario for automata.

16.4 Identification in the limit

In the *identification in the limit framework*, the problem consists of identifying a target concept c exactly after receiving a finite set of examples. A class of languages is said to be *identifiable in the limit* if there exists an algorithm that identifies any language L in that class after examining a finite number of examples and its hypothesis remains unchanged thereafter.

This framework is perhaps less realistic from a computational point of view since it requires no upper bound on the number of instances or the efficiency of the algorithm. Nevertheless, it has been argued by some to be similar to the scenario of humans learning languages. In this framework as well, negative results hold for the general problem of learning DFAs.

Theorem 16.6 *Deterministic automata are not identifiable in the limit from positive examples.*

Some sub-classes of finite automata can however be successfully identified in the limit. Most algorithms for inference of automata are based on a *state-partitioning paradigm*. They start with an initial DFA, typically a tree accepting the finite set of sample strings available and the trivial partition: each block is reduced to one state of the tree. At each iteration, they merge partition blocks while preserving some congruence property. The iteration ends when no other merging is possible and the final partition defines the automaton inferred. Thus, the choice of the congruence fully determines the algorithm and a variety of different algorithms can be defined by varying that choice. A *state-splitting paradigm* can be similarly defined starting from the single-state automaton accepting Σ^*. In this section, we present an algorithm for learning reversible automata, which is a special instance of the general state-partitioning algorithmic paradigm just described.

Let $A = (\Sigma, Q, I, F, E)$ be a DFA and let π be a partition of Q. The DFA defined by the partition π is called the *automaton quotient of A and π*. It is denoted by

A/π and defined as follows: $A/\pi = (\Sigma, \pi, I_\pi, F_\pi, E_\pi)$ with

$$I_\pi = \{B \in \pi \colon I \cap B \neq \emptyset\}$$
$$F_\pi = \{B \in \pi \colon F \cap B \neq \emptyset\}$$
$$E_\pi = \{(B, a, B') \colon \exists (q, a, q') \in E \mid q \in B, q' \in B', B \in \pi, B' \in \pi\}.$$

Let S be a finite set of strings and let $\mathrm{Pref}(S)$ denote the set of prefixes of all strings of S. A *prefix-tree automaton* accepting exactly the set of strings S is a particular DFA denoted by $PT(S) = (\Sigma, \mathrm{Pref}(S), \{\epsilon\}, S, E_S)$ where Σ is the set of alphabet symbols used in S and E_S defined as follows:

$$E_S = \{(x, a, xa) \colon x \in \mathrm{Pref}(S), xa \in \mathrm{Pref}(S)\}.$$

Figure 16.7a shows the prefix-tree automaton of a particular set of strings S.

16.4.1 Learning reversible automata

In this section, we show that the sub-class of *reversible automata* or *reversible languages* can be identified in the limit. In particular, we show that the language can be identified given a *positive presentation*.

A positive presentation of a language L is an infinite sequence $(x_n)_{n \in \mathbb{N}}$ such that $\{x_n \colon n \in \mathbb{N}\} = L$. Thus, in particular, for any $x \in L$ there exists $n \in \mathbb{N}$ such that $x = x_n$. An algorithm identifies L in the limit from a positive presentation if there exists $N \in \mathbb{N}$ such that for $n \geq N$ the hypothesis it returns is L.

Given a DFA A, we define its *reverse* A^R as the automaton derived from A by making the initial state final, the final states initial, and by reversing the direction of every transition. The language accepted by the reverse of A is precisely the language of the reverse (or mirror image) of the strings accepted by A.

Definition 16.7 (Reversible automata) *A finite automaton A is said to be* reversible *iff both A and A^R are deterministic. A language L is said to be* reversible *if it is the language accepted by some reversible automaton.*

Some direct consequences of this definition are that a reversible automaton A has a unique final state and that its reverse A^R is also reversible. Note also that a trim reversible automaton A is minimal. Indeed, if states q and q' in A are equivalent, then, they admit a common string x leading both from q and from q' to a final state. But, by the reverse determinism of A, reading the reverse of x from the final state must lead to a unique state, which implies that $q = q'$.

For any $u \in \Sigma^*$ and any language $L \subseteq \Sigma^*$, let $\mathrm{Suff}_L(u)$ denote the set of all possible suffixes in L for u:

$$\mathrm{Suff}_L(u) = \{v \in \Sigma^* \colon uv \in L\}. \tag{16.1}$$

$\mathrm{Suff}_L(u)$ is also often denoted by $u^{-1}L$. Observe that if L is a reversible language, then the following implication holds for any two strings $u, u' \in \Sigma^*$:

$$\mathrm{Suff}_L(u) \cap \mathrm{Suff}_L(u') \neq \emptyset \implies \mathrm{Suff}_L(u) = \mathrm{Suff}_L(u'). \tag{16.2}$$

Indeed, let A be a reversible automaton accepting L. Let q be the state of A reached from the initial state when reading u and q' the one reached reading u'. If $v \in \mathrm{Suff}_L(u) \cap \mathrm{Suff}_L(u')$, then v can be read both from q and q' to reach the final state. Since A^R is deterministic, reading back the reverse of v from the final state must lead to a unique state, therefore $q = q'$, that is $\mathrm{Suff}_L(u) = \mathrm{Suff}_L(u')$.

Let $A = (\Sigma, Q, \{i_0\}, \{f_0\}, E)$ be a reversible automaton accepting a reversible language L. We define a set of strings S_L as follows:

$$S_L = \{d[q]f[q] : q \in Q\} \cup \{d[q], a, f[q'] : q, q' \in Q, a \in \Sigma\},$$

where $d[q]$ is a string of minimum length from i_0 to q, and $f[q]$ a string of minimum length from q to f_0. As shown by the following proposition, S_L characterizes the language L in the sense that any reversible language containing S_L must contain L.

Proposition 16.8 *Let L be a reversible language. Then, L is the smallest reversible language containing S_L.*

Proof: Let L' be a reversible language containing S_L and let $x = x_1 \cdots x_n$ be a string accepted by L, with $x_k \in \Sigma$ for $k \in [n]$ and $n \geq 1$. For convenience, we also define x_0 as ϵ. Let $(q_0, x_1, q_1) \cdots (q_{n-1}, x_n, q_n)$ be the accepting path in A labeled with x. We show by recurrence that $\mathrm{Suff}_{L'}(x_0 \cdots x_k) = \mathrm{Suff}_{L'}(d[q_k])$ for all $k \in \{0, \ldots, n\}$. Since $d[q_0] = d[i_0] = \epsilon$, this clearly holds for $k = 0$. Now assume that $\mathrm{Suff}_{L'}(x_0 \cdots x_k) = \mathrm{Suff}_{L'}(d[q_k])$ for some $k \in \{0, \ldots, n-1\}$. This implies immediately that $\mathrm{Suff}_{L'}(x_0 \cdots x_k x_{k+1}) = \mathrm{Suff}_{L'}(d[q_k]x_{k+1})$. By definition, S_L contains both $d[q_{k+1}]f[q_{k+1}]$ and $d[q_k]x_{k+1}f[q_{k+1}]$. Since L' includes S_L, the same holds for L'. Thus, $f[q_{k+1}]$ belongs to $\mathrm{Suff}_{L'}(d[q_{k+1}) \cap \mathrm{Suff}_{L'}(d[q_k]x_{k+1})$. In view of (16.2), this implies that $\mathrm{Suff}_{L'}(d[q_k]x_{k+1}) = \mathrm{Suff}_{L'}(d[q_{k+1}])$. Thus, we have $\mathrm{Suff}_{L'}(x_0 \cdots x_k x_{k+1}) = \mathrm{Suff}_{L'}(d[q_{k+1}])$. This shows that $\mathrm{Suff}_{L'}(x_0 \cdots x_k) = \mathrm{Suff}_{L'}(d[q_k])$ holds for all $k \in \{0, \ldots, n\}$, in particular, for $k = n$. Note that since $q_n = f_0$, we have $f[q_n] = \epsilon$, therefore $d[q_n] = d[q_n]f[q_n]$ is in $S_L \subseteq L'$, which implies that $\mathrm{Suff}_{L'}(d[q_n])$ contains ϵ and thus that $\mathrm{Suff}_{L'}(x_0 \cdots x_n)$ contains ϵ. This is equivalent to $x = x_0 \cdots x_n \in L'$. □

Figure 16.6 shows the pseudocode of an algorithm for inferring a reversible automaton from a sample S of m strings $x_1, \ldots, x_m$. The algorithm starts by creating a prefix-tree automaton A for S (line 1) and then iteratively defines a partition π of the states of A, starting with the trivial partition π_0 with one block per state (line 2). The automaton returned is the quotient of A and the final partition π defined.

```
LearnReversibleAutomata(S = (x_1, ..., x_m))
 1  A = (Σ, Q, {i_0}, F, E) ← PT(S)
 2  π ← π_0 ▷ trivial partition.
 3  list ← {(f, f') : f' ∈ F} ▷ f arbitrarily chosen in F.
 4  while list ≠ ∅ do
 5      Remove(list, (q_1, q_2))
 6      if B(q_1, π) ≠ B(q_2, π) then
 7          B_1 ← B(q_1, π)
 8          B_2 ← B(q_2, π)
 9          for all a ∈ Σ do
10              if (succ(B_1, a) ≠ ∅) ∧ (succ(B_2, a) ≠ ∅) then
11                  Add(list, (succ(B_1, a), succ(B_2, a)))
12              if (pred(B_1, a) ≠ ∅ ∧ (pred(B_2, a) ≠ ∅)) then
13                  Add(list, (pred(B_1, a), pred(B_2, a)))
14              Update(succ, pred, B_1, B_2)
15              π ← Merge(π, B_1, B_2)
16  return A/π
```

Figure 16.6
Algorithm for learning reversible automata from a set of positive strings S.

The algorithm maintains a list LIST of pairs of states whose corresponding blocks are to be merged, starting with all pairs of final states (f, f') for an arbitrarily chosen final state $f \in F$ (line 3). We denote by $B(q, \pi)$ the block containing q based on the partition π.

For each block B and alphabet symbol $a \in \Sigma$, the algorithm also maintains a successor $succ(B, a)$, that is, a state that can be reached by reading a from a state of B; $succ(B, a) = \emptyset$ if no such state exists. It maintains similarly the predecessor $pred(B, a)$, which is a state that admits a transition labeled with a leading to a state in B; $pred(B, a) = \emptyset$ if no such state exists.

Then, while LIST is not empty, a pair is removed from LIST and processed as follows. If the pair (q_1, q_1') has not been already merged, the pairs formed by the successors and predecessors of $B_1 = B(q_1, \pi)$ and $B_2 = B(q_2, \pi)$ are added to LIST

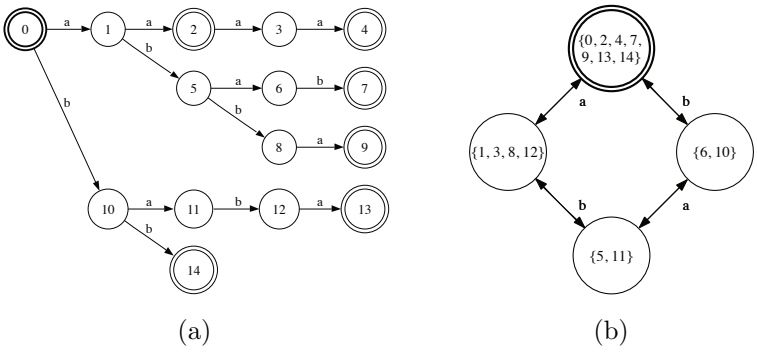

Figure 16.7
Example of inference of a reversible automaton. (a) Prefix-tree $PT(S)$ representing $S = (\epsilon, aa, bb, aaaa, abab, abba, baba)$. (b) Automaton $\widehat{A}$ returned by LearnReversibleAutomata() for the input S. A double-direction arrow represents two transitions with the same label with opposite directions. The language accepted by $\widehat{A}$ is that of strings with an even number of as and bs.

(lines 10–13). Before merging blocks B_1 and B_2 into a new block B' that defines a new partition π (line 15), the successor and predecessor values for the new block B' are defined as follows (line 14). For each symbol $a \in \Sigma$, $succ(B', a) = \emptyset$ if $succ(B_1, a) = succ(B_2, a) = \emptyset$, otherwise $succ(B', a)$ is set to one of $succ(B_1, a)$ if it is non-empty, $succ(B_2, a)$ otherwise. The predecessor values are defined in a similar way. Figure 16.7 illustrates the application of the algorithm in the case of a sample with $m = 7$ strings.

Proposition 16.9 *Let S be a finite set of strings and let $A = PT(S)$ be the prefix-tree automaton defined from S. Then, the final partition defined by* LearnReversibleAutomata() *used with input S is the finest partition π for which A/π is reversible.*

Proof: Let T be the number of iterations of the algorithm for the input sample S. We denote by π_t the partition defined by the algorithm after $t \geq 1$ iterations of the loop, with π_T the final partition.

A/π_T is a reversible automaton since all final states are guaranteed to be merged into the same block as a consequence of the initialization step of line 3 and, for any block B, by definition of the algorithm, states reachable by $a \in \Sigma$ from B are contained in the same block, and similarly for those admitting a transition labeled with a to a state of B.

Let π' be a partition of the states of A for which A/π' is reversible. We show by recurrence that π_T refines π'. Clearly, the trivial partition π_0 refines π'. Assume that π_s refines π' for all $s \leq t$. π_{t+1} is obtained from π by merging two blocks $B(q_1, \pi_t)$ and $B(q_2, \pi_t)$. Since π_t refines π', we must have $B(q_1, \pi_t) \subseteq B(q_1, \pi')$

and $B(q_2, \pi_t) \subseteq B(q_2, \pi')$. To show that π_{t+1} refines π', it suffices to prove that $B(q_1, \pi') = B(q_2, \pi')$.

A reversible automaton has only one final state, therefore, for the partition π', all final states of A must be placed in the same block. Thus, if the pair (q_1, q_2) processed at the $(t+1)$th iteration is a pair of final states placed in LIST at the initialization step (line 3), then we must have $B(q_1, \pi') = B(q_2, \pi')$. Otherwise, (q_1, q_2) was placed in LIST as a pair of successor or predecessor states of two states q_1' and q_2' merged at a previous iteration $s \leq t$. Since π_s refines π', q_1' and q_2' are in the same block of π' and since A/π' is reversible, q_1 and q_2 must also be in the same block as successors or predecessors of the same block for the same label $a \in \Sigma$, thus $B(q_1, \pi') = B(q_2, \pi')$. □

Theorem 16.10 *Let S be a finite set of strings and let A be the automaton returned by* LearnReversibleAutomata() *when used with input S. Then, $L(A)$ is the smallest reversible language containing S.*

Proof: Let L be a reversible language containing S, and let A' be a reversible automaton with $L(A') = L$. Since every string of S is accepted by A', any $u \in \mathrm{Pref}(S)$ can be read from the initial state of A' to reach some state $q(u)$ of A'. Consider the automaton A'' derived from A' by keeping only states of the form $q(u)$ and transitions between such states. A'' has the unique final state of A' since $q(u)$ is final for $u \in S$, and it has the initial state of A', since ϵ is a prefix of strings of S. Furthermore, A'' directly inherits from A' the property of being deterministic and reverse deterministic. Thus, A'' is reversible.

The states of A'' define a partition of $\mathrm{Pref}(S)$: $u, v \in \mathrm{Pref}(S)$ are in the same block iff $q(u) = q(v)$. Since by definition of the prefix-tree $PT(S)$, its states can be identified with $\mathrm{Pref}(S)$, the states of A'' also define a partition π' of the states of $PT(S)$ and thus $A'' = PT(S)/\pi'$. By proposition 16.9, the partition π defined by algorithm LearnReversibleAutomata() run with input S is the finest such that $PT(S)/\pi$ is reversible. Therefore, we must have $L(PT(S)/\pi) \subseteq L(PT(S)/\pi') = L(A'')$. Since A'' is a sub-automaton of A', L contains $L(A'')$ and therefore $L(PT(S)/\pi) = L(A)$, which concludes the proof. □

Theorem 16.11 (Identification in the limit of reversible languages) *Let L be a reversible language, then algorithm* LearnReversibleAutomata() *identifies L in the limit from a positive presentation.*

Proof: Let L be a reversible language. By proposition 16.8, L admits a finite characteristic sample S_L. Let $(x_n)_{n \in \mathbb{N}}$ be a positive presentation of L and let $\mathcal{X}_n$ denote the union of the first n elements of the sequence. Since S_L is finite, there exists $N \geq 1$ such that $S_L \subseteq \mathcal{X}_N$. By theorem 16.10, for any $n \geq N$, LearnReversibleAutomata() run on the finite sample $\mathcal{X}_n$ returns the smallest

reversible language L' containing $\mathcal{X}_n$ a fortiori S_L, which, by definition of S_L, implies that $L' = L$. □

The main operations needed for the implementation of the algorithm for learning reversible automata are the standard FIND and UNION to determine the block a state belongs to and to merge two blocks into a single one. Using a disjoint-set data structure for these operations, the time complexity of the algorithm can be shown to be in $O(n\alpha(n))$, where n denotes the sum of the lengths of all strings in the input sample S and $\alpha(n)$ the inverse of the Ackermann function, which is essentially constant ($\alpha(n) \leq 4$ for $n \leq 10^{80}$).

16.5 Chapter notes

For an overview of finite automata and some related results, see Hopcroft and Ullman [1979] or the more recent Handbook chapter by Perrin [1990], as well as the series of books by M. Lothaire [Lothaire, 1982, 1990, 2005] and the even more recent book by De la Higuera [2010].

Theorem 16.2, stating that the problem of finding a minimum consistent DFA is NP-hard, is due to Gold [1978]. This result was later extended by Angluin [1978]. Pitt and Warmuth [1993] further strengthened these results by showing that even an approximation within a polynomial function of the size of the smallest automaton is NP-hard (theorem 16.3). Their hardness results apply also to the case where prediction is made using NFAs. Kearns and Valiant [1994] presented hardness results of a different nature relying on cryptographic assumptions. Their results imply that no polynomial-time algorithm can learn consistent NFAs polynomial in the size of the smallest DFA from a finite sample of accepted and rejected strings if any of the generally accepted cryptographic assumptions holds: if factoring Blum integers is hard; or if the RSA public key cryptosystem is secure; or if deciding quadratic residuosity is hard. Most recently, Chalermsook et al. [2014] improved the non-approximation guarantee of Pitt and Warmuth [1993] to a tight bound.

On the positive side, Trakhtenbrot and Barzdin [1973] showed that the smallest finite automaton consistent with the input data can be learned exactly from a uniform complete sample, whose size is exponential in the size of the automaton. The worst-case complexity of their algorithm is exponential, but a better average-case complexity can be obtained assuming that the topology and the labeling are selected randomly [Trakhtenbrot and Barzdin, 1973] or even that the topology is selected adversarially [Freund et al., 1993].

Cortes, Kontorovich, and Mohri [2007a] study an approach to the problem of learning automata based on linear separation in some appropriate high-dimensional feature space; see also Kontorovich et al. [2006, 2008]. The mapping of strings to

that feature space can be defined implicitly using the rational kernels presented in chapter 6, which are themselves defined via weighted automata and transducers.

The model of learning with queries was introduced by Angluin [1978], who also proved that finite automata can be learned in time polynomial in the size of the minimal automaton and that of the longest counter-example. Bergadano and Varricchio [1995] further extended this result to the problem of learning weighted automata defined over any field (see also an optimal algorithm by Bisht et al. [2006]). Using the relationship between the size of a minimal weighted automaton over a field and the rank of the corresponding Hankel matrix, the learnability of many other concepts classes such as disjoint DNF can be shown [Beimel et al., 2000]. Our description of an efficient implementation of the algorithm of Angluin [1982] using decision trees is adapted from Kearns and Vazirani [1994].

The model of identification in the limit of automata was introduced and analyzed by Gold [1967]. Deterministic finite automata were shown not to be identifiable in the limit from positive examples [Gold, 1967]. But, positive results were given for the identification in the limit of a number of sub-classes, such as the family of k-reversible languages Angluin [1982] considered in this chapter. Positive results also hold for learning subsequential transducers Oncina et al. [1993]. Some restricted classes of probabilistic automata such as acyclic probabilistic automata were also shown by Ron et al. [1995] to be efficiently learnable.

There is a vast literature dealing with the problem of learning automata. In particular, positive results have been shown for a variety of sub-families of finite automata in the scenario of learning with queries and learning scenarios of different kinds have been introduced and analyzed for this problem. The results presented in this chapter should therefore be viewed only as an introduction to that material.

16.6 Exercises

16.1 Minimal DFA. Show that a minimal DFA A also has the minimal number of transitions among all other DFAs equivalent to A. Prove that a language L is regular iff $Q = \{\text{Suff}_L(u) \colon u \in \Sigma^*\}$ is finite. Show that the number of states of a minimal DFA A with $L(A) = L$ is precisely the cardinality of Q.

16.2 VC-dimension of finite automata.

(a) What is the VC-dimension of the family of all finite automata? What does that imply for PAC-learning of finite automata? Does this result change if we restrict ourselves to learning acyclic automata (automata with no cycles)?

(b) Show that the VC-dimension of the family of DFAs with at most n states is bounded by $O(|\Sigma| n \log n)$.

16.3 PAC learning with membership queries. Give an example of a concept class $\mathcal{C}$ that is efficiently PAC-learnable with membership queries but that is not efficiently exactly learnable.

16.4 Learning monotone DNF formulae with queries. Show that the class of monotone DNF formulae over n variables is efficiently exactly learnable using membership and equivalence queries. (*Hint*: a *prime implicant* t of a formula f is a product of literals such that t implies f but no proper sub-term of t implies f. Use the fact that for monotone DNF, the number of prime implicants is at the most the number of terms of the formula.)

16.5 Learning with unreliable query responses. Consider the problem where the learner must find an integer x selected by the oracle within $[n]$, where $n \geq 1$ is given. To do so, the learner can ask questions of the form $(x \leq m?)$ or $(x > m?)$ for $m \in [n]$. The oracle responds to these questions but may give an incorrect response to k questions. How many questions should the learner ask to determine x? (*Hint*: observe that the learner can repeat each question $2k+1$ times and use the majority vote.)

16.6 Algorithm for learning reversible languages. What is the DFA A returned by the algorithm for learning reversible languages when applied to the sample $S = \{ab, aaabb, aabbb, aabbbb\}$? Suppose we add a new string to the sample, say $x = abab$. How should A be updated to compute the result of the algorithm for $S \cup \{x\}$? More generally, describe a method for updating the result of the algorithm incrementally.

16.7 k-reversible languages. A finite automaton A' is said to be *k-deterministic* if it is deterministic modulo a lookahead k: if two distinct states p and q are both initial, or are both reached from another state r by reading $a \in \Sigma$, then no string u of length k can be read in A' both from p and q. A finite automaton A is said to be *k-reversible* if it is deterministic and if A^R is k-deterministic. A language L is *k-reversible* if it is accepted by some k-reversible automaton.

(a) Prove that L is k-reversible iff for any strings $u, u', v \in \Sigma^*$ with $|v| = k$,

$$\mathrm{Suff}_L(uv) \cap \mathrm{Suff}_L(u'v) \neq \emptyset \implies \mathrm{Suff}_L(uv) = \mathrm{Suff}_L(u'v).$$

(b) Show that a k-reversible language admits a characteristic language.

(c) Show that the following defines an algorithm for learning k-reversible automata. Proceed as in the algorithm for learning reversible automata but with the following merging rule instead: merge blocks B_1 and B_2 if they can be reached by the same string u of length k from some other block and if B_1 and B_2 are both final or have a common successor.