Journal of Computer and System Sciences **56**, 133–152 (1998)
Article No. SS971555

On the Learnability and Usage of Acyclic Probabilistic Finite Automata

Dana Ron

Laboratory of Computer Science, MIT, Cambridge, Massachusetts 02139
E-mail: danar@theory.lcs.mit.edu

Yoram Singer

AT&T Labs, 600 Mountian Avenue, Murray Hill, New Jersey 07974
E-mail: singer@research.att.com

and

Naftali Tishby

Institute of Computer Science and Center for Neural Computation, Hebrew University, Jerusalem 91904, Israel
E-mail: tishby@cs.huji.ac.il

Received November 13, 1995; revised March 3, 1997

We propose and analyze a distribution learning algorithm for a subclass of *acyclic probalistic finite automata* (APFA). This subclass is characterized by a certain distinguishability property of the automata's states. Though hardness results are known for learning distributions generated by general APFAs, we prove that our algorithm can efficiently learn distributions generated by the subclass of APFAs we consider. In particular, we show that the KL-divergence between the distribution generated by the target source and the distribution generated by our hypothesis can be made arbitrarily small with high confidence in polynomial time.

We present two applications of our algorithm. In the first, we show how to model cursively written letters. The resulting models are part of a complete cursive handwriting recognition system. In the second application we demonstrate how APFAs can be used to build multiple-pronunciation models for spoken words. We evaluate the APFA-based pronunciation models on labeled speech data. The good performance (in terms of the log-likelihood obtained on test data) achieved by the APFAs and the little time needed for learning suggests that the learning algorithm of APFAs might be a powerful alternative to commonly used probabilistic models. © 1998 Academic Press

1. INTRODUCTION

An important class of problems that arise in machine learning applications is that of modeling classes of natural sequences with their possibly complex variations. Such sequence models are essential, for instance, in handwriting and speech recognition, natural language processing, and biochemical sequence analysis. Our interest here is specifically in modeling short sequences that correspond to objects such as single handwritten letters, spoken words, or short protein sequences.

In this paper we consider using *acyclic probabilistic finite automata* (APFAs) for modeling distributions on short sequences such as those mentioned above. Probabilistic finite automata[1] (PFA), as referred to in this work, are finite state machines that *generate* strings in the following probabilistic manner: Starting from the start state, at each step an edge going out of the current state is chosen according to the probability assigned to that edge, and the symbol labeling the edge is emitted (where for each state there is at most one outgoing edge labeled by each symbol). We consider in particular PFAs whose underlying graph is acyclic. We present and analyze an efficient and easily implementable learning algorithm for a subclass of APFAs that have a certain *distinguishability* property which is defined subsequently. We describe two applications of our algorithm. In the first application we construct models for cursive handwritten letters, and in the second we build pronunciation models for spoken words. These application use in part an on-line version of our algorithm which is also given in this paper.

The algorithm described in this paper is an efficient algorithm for learning distributions on strings generated by all APFAs M which have the following property. For every pair of states in M, the distance in the L_∞ norm between the distributions generated starting from these two states is non-negligible, namely, this distance is an inverse polynomial in the size of M. We show that for every such target

[1] These automata should not be confused with probabilistic *acceptors* of strings, which are often also called probabilistic automata.

APFA, given a large enough sample (though of size polynomial in the number of states of the target APFA and other relevant parameters), our learning algorithm constructs a hypothesis APFA such that with high probability, the Kullback–Liebler (KL) divergence between the hypothesis APFA and the target APFA is small. The learning algorithm is efficient in the sense that its running time is polynomial in the parameters of the problem.

Our result should be contrasted with the intractability result for learning PFAs proved by Kearns *et al.* [8]. They show that PFAs are not efficiently learnable under the assumption that there is no efficient algorithm for learning parity functions in the presence of noise in the PAC model.[2] Furthermore, the subclass of PFAs which they show are hard to learn are (width two) APFAs in which the distance in the L_1 norm (and hence also the KL-divergence) between the distributions generated starting from every pair of states is large.

One of the key techniques applied in this work is that of using some form of *signatures* of states in order to distinguish between the states of the target automaton. This technique was presented in the pioneering work of Trakhtenbrot and Brazdin' [20] in the context of learning deterministic finite automata (DFAs). The same idea was later applied by Freund *et al.* [6] in their work on learning typical DFAs.[3] In the same work they proposed to apply the notion of *statistical signatures* to learning typical PFAs.

The outline of our learning algorithm is roughly the following. In the course of the algorithm we construct a sequence of directed edge-labeled (leveled) acyclic graphs. The first graph in this sequence, named the *sample tree*, is constructed based on the sample generated by the target APFA, while the last graph in the sequence is the underlying graph of our hypothesis APFA. Each graph in this sequence is transformed into the next graph by *a folding operation* in which a pair of nodes that have passed a certain *similarity test* are merged into a single node (and so are the pairs of their respective successors). With each node we associate a multiset of prefixes of sample strings. The similarity test compares the statistics of multisets of suffixes of sample strings which correspond to these multisets of prefixes. Put another way, we view suffixes of the sample strings as "originating" from particular nodes and associate the statistics of the suffixes with these nodes. If two nodes have similar statistics, then they are assumed to correspond to the same state in the target APFA. In the initial graph, the sample tree, such statistics are reliable only for nodes which are very close to the root of the tree, and hence have relatively many sample strings passing through them. However, as a consequence of the folding operation, nodes which are further away from the root are merged, and so are their respective multisets of originating strings. Thus the folding operation in levels closer to the root enhances the reliability of the statistics that correspond to nodes which are further away from the root. As a result, with high probability, the algorithm only folds pairs of nodes that in fact correspond to the same state, and the nodes which are left unmerged can be shown to contribute little to the error of the hypothesis.

In a previous work [14] we introduced an algorithm for learning distributions (on long strings) generated by ergodic Markovian sources that can be characterized by a different subclass of PFAs which we refer to as *variable memory* PFAs. Our two learning algorithms complement each other. Whereas the variable memory PFAs capture the long-range, stationary, statistical properties of the source, the APFAs capture the short sequence statistics. Together, these algorithm constitute a complete language modeling scheme, which we applied to cursive handwriting recognition [16].

1.1. *Other Related Work*

The most common approaches to the modeling and recognition of sequences such as those studied in this paper are string matching algorithms (e.g., dynamic time warping [15]) and hidden Markov models (in particular *left-to-right* HMMs) [11, 12]. The string matching approach usually assumes the existence of a sequence prototype (reference template) together with a local noise model, from which the probabilities of deletions, insertions, and substitutions can be deduced. The string matching models tend to be too weak for the type of applications we are interested in since they cannot easily capture statistical dependencies which are not local.

HMMs (which PFAs are a special case of) are popular in speech recognition and have better ability than the string-matching-based techniques to capture context-dependent variations. The commonly used training procedure for HMMs which is based on the *forward–backward* algorithm [2] is guaranteed to converge only to a *local* maximum of the likelihood function. Furthermore, there are theoretical results indicating that the problem of learning distributions generated by HMMs is hard [1, 8]. In addition, the successful applications of the HMM approach occur mostly in cases where its full power is not utilized, and the hypothesis constructed is essentially a PFA (or even an APFA). Another drawback of HMMs is that the current HMM training algorithms are neither on-line nor adaptive in the model's topology.

A technique of merging states which is similar to the one used in this paper was also applied by Carrasco and Oncina

[2] The problem of learning parity with noise is closely related to the long standing problem of decoding random linear codes. Additional evidence to the intractability of this problem is provided in [3, 9].

[3] They define typical DFAs to be DFAs in which the underlying graph is arbitrary, but the accept/reject labels on the states are chosen randomly.

[4], and by Stolcke and Omohundro [18]. Carrasco and Oncina give an algorithm which identifies distributions generated by PFAs *in the limit* of infinite examples. Stolcke and Omohundro describe a learning algorithm for HMMs which merges states based on a Bayesian approach and apply their algorithm to build pronunciation models for spoken words. Examples of alternative approaches for modeling multiple pronunciation, such as decision trees, can be found in [5, 13]. For general reviews on cursive handwriting recognition see [10, 19].

1.2. Organization of the Paper

The paper is organized as follows. In Sections 2 and 3 we give several definitions related to APFAs and define our learning model. In Section 4 we present our learning algorithm. In Section 5 we state and prove our main theorem concerning the correctness of the learning algorithm. In Section 6 we give an on-line version of our algorithm, and in Section 7 we describe two applications of the batch and on-line algorithms. We conclude with several suggestions for future research in Section 8.

2. PRELIMINARIES

A *probabilistic finite automaton*[4] (PFA) is an automaton which has a designated starting state and a designated final state. The edges going out of each state are labeled by symbols drawn from an alphabet Σ, where for every state, each outgoing edge is labeled by a different symbol. The states of the PFA are unlabeled. With each edge we associate a probability, where for every state, the probabilities of all outgoing edges sum up to one. All edges pointing at the final state (and those edges only) are labeled by a special final symbol ζ. A PFA can be thought of as mechanism for generating strings in $\Sigma^*\zeta$ in the following straightforward manner. Starting from the starting state and until the final state is reached, at each step an edge going out of the current state is chosen according to the probabilities associated with the edges. The chosen edge is then traversed to the next state, and the symbol labeling the edge is emitted.

More formally, a PFA M is a 7-tuple $(Q, q_0, q_f, \Sigma, \zeta, \tau, \gamma)$ where

- Q is a finite set of *states*;
- $q_0 \in Q$ is the *starting state*;
- $q_f \notin Q$ is the *final state*;
- Σ is a finite *alphabet*;
- $\zeta \notin \Sigma$ is the *final symbol*;
- τ: $Q \times \{\Sigma \cup \{\zeta\}\} \to Q \cup \{q_f\}$ is the *transition function*;
- γ: $Q \times \{\Sigma \cup \{\zeta\}\} \to [0, 1]$ is the *next symbol probability function.*

[4] The definition we use is slightly non-standard in the sense that we assume a final symbol and a final state.

The function γ satisfies the following requirement: for every $q \in Q$, $\sum_{\sigma \in \Sigma \cup \{\zeta\}} \gamma(q, \sigma) = 1$. We allow the transition function τ to be undefined only on states q and symbols σ, for which $\gamma(q, \sigma) = 0$. We require that for every $q \in Q$ such that $\gamma(q, \zeta) > 0$, $\tau(q, \zeta) = q_f$. We also require that q_f can be reached (i.e., with non-zero probability) from *every* state q which can be reached from the starting state, q_0. τ can be extended to be defined on $Q \times \Sigma^*$ (similarly on $Q \times \Sigma^*\zeta$) in the following recursive manner: $\tau(q, s_1 s_2 \cdots s_\ell) = \tau(\tau(q, s_1 \cdots s_{\ell-1}), s_\ell)$.

A PFA M can be thought of as a mechanism for generating strings of finite length ending with the symbol ζ, in the following sequential manner. Starting from q_0, until q_f is reached, if q_i is the current state, then the next symbol is chosen (probabilistically) according to $\gamma(q_i, \cdot)$. If $\sigma \in \Sigma \cup \{\zeta\}$ is the symbol generated, then the next state, q_{i+1}, is $\tau(q_i, \sigma)$. Thus, the probability M generates a string $s = s_1 \cdots s_{\ell-1} s_\ell$, where $s_\ell = \zeta$, denoted by $P^M(s)$ is

$$P^M(s) \overset{\text{def}}{=} \prod_{i=0}^{\ell-1} \gamma(q_i, s_{i+1}). \tag{1}$$

This definition implies that $P^M(\cdot)$ is in fact a probability distribution over strings ending with the symbol ζ, i.e.,

$$\sum_{s \in \Sigma^*\zeta} P^M(s) = 1.$$

For a string $s = s_1 \cdots s_\ell$ where $s_\ell \neq \zeta$ we choose to use the same notation $P^M(s)$ to denote the probability that s is a *prefix* of some generated string $s' = ss''\zeta$. Namely, $P^M(s) = \prod_{i=0}^{\ell-1} \gamma(q_i, s_{i+1})$.

Given a state q in Q, and a string $s = s_1 \cdots s_\ell$ (that does not necessarily end with ζ), let $P_q^M(s)$ denote the probability that s is (a prefix of a string) generated starting from q. Namely,

$$P_q^M(s) \overset{\text{def}}{=} \prod_{i=0}^{\ell-1} \gamma(\tau(q, s_1, ..., s_i), s_{i+1}),$$

where the convention is that for the empty string λ, $\tau(q, \lambda) = q$. The following definition is central to this work.

Definition 2.1. For $\mu > 0$, we say that two states, q_1 and q_2, in Q are *μ-distinguishable* if there exists a string s for which $|P_{q_1}^M(s) - P_{q_2}^M(s)| \geq \mu$. In other words, the distance in the L_∞ norm between the distributions $P_{q_1}^M$ and $P_{q_2}^M$ is at least μ. We say that a PFA M is *μ-distinguishable* if every pair of states in M is *μ-distinguishable*.[5]

We restrict our attention to a subclass of PFAs which have the following property: the underlying graph of every PFA in this subclass is *acyclic*. The *depth* of an acyclic PFA is defined to be the length of the longest path from q_0 to q_f. In particular, we consider *leveled* APFAs. In such an APFA, each state belongs to a single level d, where the starting state, q_0, is the only state in level 0, and the final state, q_f, is the only state in level D, where D is the depth of the APFA. All transitions from a state in level d are to states in level $d+1$, except for transitions labeled by the final symbol, ζ, which can go from any state to the final state. We denote the set of states belonging to level d, by Q_d. In the following lemma we show that every APFA can be transformed to a leveled APFA that is not much larger.

LEMMA 2.1. *For every APFA M having n states and depth D, there exists an equivalent APFA, $\tilde{M}$, that is leveled and has at most $n(D-1)$ states.*

Proof. We define $\tilde{M} = (\tilde{Q}, \Sigma, \zeta, \tilde{\tau}, \tilde{\gamma}, \tilde{q}_0, \tilde{q}_f)$ as follows. For each state q in $Q - \{q_f\}$, we create at most $D-1$ copies of q, each belonging to a different level in $\tilde{M}$. We create a copy of q in level d if there exists some path of length d from q_0 to q in M. If there was an edge from q to $q' \neq q_f$ in M, then for each level d we put an edge from the copy of q in level d (if such a copy exists) to the copy of q' in level $d+1$. If $q' = q_f$ then we have a single copy $\tilde{q}_f$ of q_f in level D of $\tilde{M}$, and an edge labeled by ζ from every copy of q to $\tilde{q}_f$.

More formally, for every state $q \in Q - \{q_f\}$, and for each level d such that there exists a string s of length d for which $\tau(q_0, s) = q$, we have a state $\tilde{q}_d \in \tilde{Q}_d$. For $q = q_0$, $(\tilde{q}_0)_0$ is simply the starting state of $\tilde{M}$, $\tilde{q}_0$, and q_f has a single copy, $\tilde{q}_f \in \tilde{Q}_D$ (which is the final state of $\tilde{M}$). For every level d and for every $\sigma \in \Sigma \cup \{\zeta\}$, $\tilde{\gamma}(\tilde{q}_d, \sigma) = \gamma(q, \sigma)$. For $\sigma \in \Sigma$, $\tilde{\tau}(\tilde{q}_d, \sigma) = \tilde{q}'_{d+1}$ where $q' = \tau(q, \sigma)$. Finally, if $\tau(q, \zeta)$ is defined then $\tilde{\tau}(\tilde{q}_d, \zeta) = \tilde{q}_f$. Every state is copied at most $D-1$ times; therefore the total number of states in $\tilde{M}$ is at most $n(D-1)$. ▌

3. THE LEARNING MODEL

In this section we describe our learning model which is similar to the one introduced by Kearns *et al.* [8]. We start by defining an *ε-good* hypothesis APFA with respect to a given target APFA.

DEFINITION 3.1. Let M be the target APFA and let $M\hat{M}$ be a hypothesis APFA. Let P^M and $P^{\hat{M}}$ be the two probability distributions they generate, respectively. We say that $\hat{M}$ is an **ε-good** hypothesis with respect to M, for $\varepsilon \geqslant 0$, if

$$\mathcal{D}_{KL}[P^M \parallel P^{\hat{M}}] \leqslant \varepsilon,$$

where $\mathcal{D}_{KL}[P^M \parallel P^{\hat{M}}]$ is the *Kullback–Liebler divergence* (also known as the cross-entropy) between the distributions and is defined as

$$\mathcal{D}_{KL}[P^M \parallel P^{\hat{M}}] \overset{\text{def}}{=} \sum_{s \in \Sigma^*\zeta} P^M(s) \log \frac{P^M(s)}{P^{\hat{M}}(s)}.$$

Our learning algorithm for APFAs is given a *confidence* parameter $\delta > 0$ and an *accuracy* parameter $\varepsilon > 0$. We assume the algorithm is given an upper bound n on the number of states in M and a *distinguishability* parameter $\mu > 0$, indicating that the target automaton is μ-distinguishable. The algorithm has access to strings generated by the target APFA, and we ask that it output with probability at least $1-\delta$ an ε-good hypothesis with respect to the target APFA. We also require that the learning algorithm be *efficient*, i.e., that it run in time polynomial in $1/\varepsilon$, $\log 1/\delta$, $|\Sigma|$, n, and $1/\mu$.

4. THE LEARNING ALGORITHM

In this section we describe our algorithm for learning APFAs. An *on-line* version of this algorithm is described in Section 6. We start with a brief informal description of the algorithm and the data structure it maintains.

4.1. *An Informal Description of the Algorithm*

Given a set, or more precisely a multiset, of sample strings, the algorithm starts by building a *sample tree*. Each path from the root of the sample tree to a leaf corresponds to a string in the sample, where the edges of the sample tree are labeled by the corresponding symbols in the string. Each internal node thus corresponds to a prefix of a string from the sample. With each edge the algorithm associates a count which is the number of strings in the sample that pass through this edge. If we wanted to predict accurately the probability that the APFA generates short prefixes of strings, then we could do so using the counts on the edges. For example, a good approximation of the probability that M generates a string starting with a certain symbol, σ, is the count associated with the edge going out of the root node, and labeled by σ, divided by the total sample size. The accuracy of such an approximation can be proven using Chernoff bounds. However, we cannot use the counts in the sample tree to reliably predict the probability that M generates longer strings. Such strings have very few appearances in the sample or may not even appear in it at all (clearly, if the sample is of size polynomial in M, and the

[5] As noted later, in the analysis of our algorithm in Section 5, we can use a slightly weaker version of the above definition, in which we require that only pairs of states with non-negligible weight be distinguishable.

support of P^M is considerably larger, then most long strings do not appear in the sample). In other words, the sample tree cannot serve as a good hypothesis for M. Therefore, we need to use the information in the sample tree in a slightly more sophisticated way.

Given a target APFA M, each node v in the sample tree can be mapped to a single state in M. This state is simply the state reached when following the path defined by the string corresponding to v (in the sample tree), starting from q_0. This mapping is clearly a many-to-one mapping. Assume the algorithm was given this mapping. Then, for each state in M it could merge all nodes which are mapped to that state to a single node. For every symbol σ the algorithm would also merge all edges going out of these nodes and labeled by σ into a single edge, adding up the counts associated with the merged edges. Using the resulting acyclic graph and the counts associated with its edges, the algorithm could define a hypothesis APFA which is a good hypothesis with high probability (for a large enough sample size). Since such a mapping is not given to the algorithm it tries to infer this mapping for as many nodes as possible, using the fact that M is μ-distinguishable.

The learning algorithm has the following structure. Starting from the first level in the sample tree, the algorithm tests whether pairs of nodes which correspond to a large enough number of prefixes of sample strings seem to map to the same state. It does this by comparing the counts on the edges in the two subtrees rooted at these nodes. These counts provide us with approximations to the probabilities that strings are generated starting from the states these nodes map to. The idea is that if two nodes correspond to different states then, since the states are μ-distinguishable, there should be evidence to this difference in the sample. If the algorithm decides that two nodes are mapped to the same state then it merges them and the corresponding nodes in their subtrees. Doing so the algorithm enhances the reliability of the test for pairs of nodes in deeper levels which had low counts (or even zero counts) associated with them prior to any mergings. In our analysis we show that with high probability, in this process, the algorithm does not merge pairs of nodes that are mapped to different states, while it does merge most pairs of nodes that do correspond to the same state, resulting in a reliably accurate hypothesis.

4.2. *A Formal Description of the Algorithm*

We start by describing the data structure used by the algorithm. Let S be a given multiset of sample strings generated by the target APFA M. In the course of the algorithm, a series of directed leveled acyclic graphs $G_0, G_1, ..., G_{N+1}$ are constructed, where the final graph, G_{N+1}, is the underlying graph of the hypothesis automaton. The initial graph G_0 is the *sample tree*, T_S. The edges of T_S are labeled by single symbols, and each node in T_S is associated with a *single* string which is a prefix of a string in S. The root of T_S, v_0, corresponds to the empty string, and every other node, v, is associated with the prefix corresponding to the labeled path from v_0 to v.

In general, in each of the graphs, $G_0, ..., G_{N+1}$, there is one node, v_0, which we refer to as the *starting node*. Every directed edge in a graph G_i is labeled by a symbol $\sigma \in \Sigma \cup \{\zeta\}$. There may be more than one directed edge between a pair of nodes, but for every node, there is at most one outgoing edge labeled by each symbol. If there is an edge labeled by σ connecting a node v to a node u, then we denote it by $v \xrightarrow{\sigma} u$. If there is a labeled (directed) path from v to u corresponding to a string s, then we denote it similarly by $v \overset{s}{\Rightarrow} u$. Each node v is *virtually* associated with a *multiset* of strings $S(v) \subseteq S$. These are the strings in the sample which correspond to the (directed) paths in the graph that *pass* through v when starting from v_0, i.e.,

$$S(v) \overset{\text{def}}{=} \{s\colon \exists s', s'' \text{ s.t. } s = s's'' \in S \text{ and } v_0 \overset{s'}{\Rightarrow} v\}_{multi}.$$

We define an additional, related, multiset, $S_{gen}(v)$, that includes the substrings in the sample which can be seen as *generated* from v. Namely,

$$S_{gen}(v) \overset{\text{def}}{=} \{s''\colon \exists s' \text{ s.t. } s's'' \in S \text{ and } v_0 \overset{s'}{\Rightarrow} v\}_{multi}.$$

By this definition, each string s'' in $S_{gen}(v)$ is a suffix of some string $s = s's''$ in $S(v)$, where its corresponding prefix, s', is such that $v_0 \overset{s'}{\Rightarrow} v$.

For each node v, and each symbol σ, we associate a count, $m_v(\sigma)$, with v's outgoing edge labeled by σ. If v does not have any outgoing edges labeled by σ, then we define $m_v(\sigma)$ to be 0. We denote $\sum_\sigma m_v(\sigma)$ by m_v. Note that $m_v(\sigma)$ equals the number of strings in $S_{gen}(v)$ whose first symbol is σ, and it always holds by construction that $m_v = |S(v)|$ $(= |S_{gen}(v)|)$.

We now describe our learning algorithm. For a more detailed description see the pseudo-code that follows. We would like to stress that the multisets of strings, $S(v)$, are maintained only virtually for the sake of clarity of our algorithm; thus the data structure used along the run of the algorithm is only the current graph, G_i, together with the counts on the edges. For $i = 0, ..., N-1$, we associate with G_i a level, $d(i)$, where $d(0) = 1$, and $d(i) \geqslant d(i-1)$. This is the level in G_i the algorithm operates on in the transformation from G_i to G_{i+1}. We transform G_i into G_{i+1} by what we call a *folding* operation. In this operation the algorithm chooses a pair of nodes u and v, both belonging to $d(i)$, which have the following properties: for a predefined threshold m_0 both $m_u \geqslant m_0$ and $m_v \geqslant m_0$, and the nodes are *similar* in a sense defined below. The algorithm then merges u and v, and all pairs of nodes they reach, respectively. If u and v are merged into a new node, w, then for every σ,

$m_w(\sigma) = m_u(\sigma) + m_v(\sigma)$. The virtual multiset of strings corresponding to w, $S(w)$, is simply the union of the multisets $S(u)$ with $S(v)$. An illustration of the folding operation is depicted in Fig. 1. Note that since the algorithm proceeds level by level, for every G_i, the nodes at level $d(i)$ are all roots of $|\Sigma|$-ary trees, while this is not the case in general in levels $d < d(i)$ where the folding operation altered the original tree structure.

Let G_N be the last graph in this series for which there does not exist such a pair of similar nodes. The algorithm transforms G_N into G_{N+1}, by performing the following operations. First, it merges all leaves in G_N into a single node v_f. Next, for each level d in G_N, it merges all nodes u in level d for which $m_u < m_0$. Let this node be denoted by *small*(d). Finally, for each node u, and for each symbol σ such that $m_u(\sigma) = 0$, if $\sigma = \zeta$, then it adds an edge labeled by ζ from u to v_f, and if $\sigma \in \Sigma$, then it adds an edge labeled by σ from u to *small*$(d+1)$ where d is the level u belongs to.

Finally, the algorithm defines the hypothesis APFA $\widehat{M} = (\hat{Q}, \hat{q}_0, \hat{q}_f, \Sigma, \zeta, \hat{\tau}, \hat{\gamma})$ based on G_{N+1}. We let G_{N+1} be the underlying graph of $\widehat{M}$, where v_0 corresponds to $\hat{q}_0$, and v_f corresponds to $\hat{q}_f$. For every state $\hat{q}$ in level d that corresponds to a node u, and for every symbol $\sigma \in \Sigma \cup \{\zeta\}$, the algorithm defines

$$\hat{\gamma}(\hat{q}, \sigma) = (m_u(\sigma)/m_u)(1 - (|\Sigma| + 1)\,\gamma_{min}) + \gamma_{min}, \qquad (2)$$

where γ_{min} is set in the analysis of the algorithm.

It remains to define the notion of *similar* nodes used in the algorithm. Roughly speaking, two nodes are considered similar if the statistics, according to the sample of the strings which can be seen as generated from these nodes, are similar. Formally, for a given node v and a string s, let $m_v(s) \overset{\text{def}}{=} |\{t\colon t \in S_{gen}(v),\ \ t = st'\}_{multi}|$ be the number of strings viewed as generated from v and contain s as a prefix. We say that a given pair of nodes u and v are *similar* if for *every* string s,

$$|m_v(s)/m_v - m_u(s)/m_u| < \mu/2.$$

As noted before, the algorithm does not maintain the multisets of strings $S_{gen}(v)$. However, the values $m_v(s)/m_v$ and $m_u(s)/m_u$ can be computed efficiently using the counts on the edges of the graphs, as described in the function **Similar** presented at the end of this section.

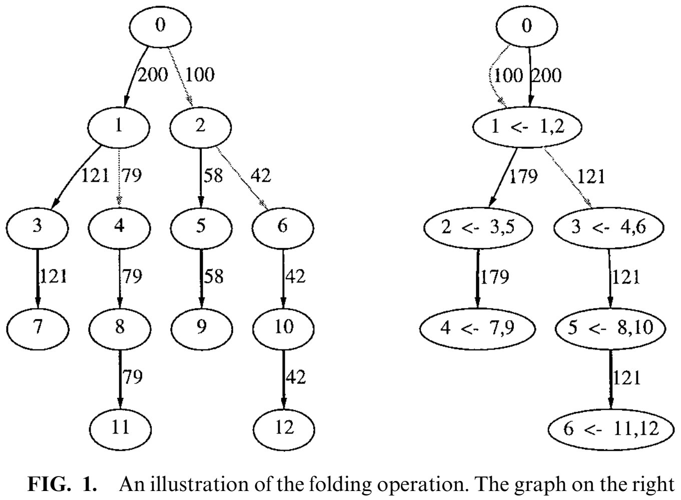


FIG. 1. An illustration of the folding operation. The graph on the right is constructed from the graph on the left by merging nodes 1 and 2. The different edges represent different output symbols: gray is 0, black is 1, and bold black is ζ.

Algorithm Learn-Acyclic-PFA

```
1. Initialize: $i := 0$, $G_0 := T_S$, $d(0) := 1$, $D :=$ depth of $T_S$ ;
2. While $d(i) < D$ do:
   (a) Look for nodes $j$ and $j'$ from level $d(i)$ in $G_i$ which have the following properties:
       i. $m_j \geq m_0$ and $m_{j'} \geq m_0$ ;
       ii. Similar$(j, 1, j', 1)$ = similar ;
   (b) If such a pair is not found let $d(i) := d(i) + 1$ ;
       /* return to while statement */
   (c) Else: /* Such a pair is found: transform $G_i$ into $G_{i+1}$ */
       i. $G_{i+1} := G_i$ ;
       ii. Call Fold$(j, j', G_{i+1})$ ;
       iii. Renumber the states of $G_{i+1}$ to be consecutive numbers in the range $1, \ldots, |G_{i+1}|$ ;
       iv. $d(i+1) := d(i)$ , $i := i + 1$ ;
3. Set $N := i$ ; Call AddSlack$(G_N, G_{N+1}, D)$ ;
4. Call GraphToPFA$(G_{N+1}, \widehat{M})$ .
```

Function Similar(u, p_u, v, p_v)

```
1. If $|p_u - p_v| \geq \mu/2$ Return non-similar ;
2. Else-If $p_u < \mu/2$ and $p_v < \mu/2$ Return similar ;
3. Else $\forall \sigma \in \Sigma \bigcup \zeta$ do:
   (a) $p'_u = p_u \cdot m_u(\sigma)/m_u$ ; $p'_v = p_v \cdot m_v(\sigma)/m_v$ ;
   (b) If $m_u(\sigma) = 0$ $u' :=$ undefined
       else $u' := \tau(u, \sigma)$ ;
   (c) If $m_v(\sigma) = 0$ $v' :=$ undefined
       else $v' := \tau(v, \sigma)$ ;
   (d) If Similar$(u', p'_u, v', p'_v)$ = non-similar
       Return non-similar ;
4. Return similar. /* Recursive calls ended and found similar */
```

```
Subroutine Fold($j, j', G$)
1. For all the nodes $k$ in G and $\forall \sigma \in \Sigma$ such that $k \xrightarrow{\sigma} j'$,
   change the corresponding edge to end at $j$, namely
   set $k \xrightarrow{\sigma} j$;
2. $\forall \sigma \in \Sigma \bigcup \zeta$:
   (a) If $m_j(\sigma) = 0$ and $m_{j'}(\sigma) > 0$, let $k$ be such that
       $j' \xrightarrow{\sigma} k$; set $j \xrightarrow{\sigma} k$;
   (b) If $m_j(\sigma) > 0$ and $m_{j'}(\sigma) > 0$, let $k$ and $k'$ be the
       indices of the states such that $j \xrightarrow{\sigma} k$ , $j' \xrightarrow{\sigma} k'$;
       Recursively fold $k, k'$: call Fold($k, k'$,G);
   (c) $m_j(\sigma) := m_{j'}(\sigma) + m_j(\sigma)$;
3. $G := G - \{j'\}$.
```

```
Subroutine AddSlack($G, G', D$)
1. Initialize: $G' := G$;
2. Merge all nodes in $G'$ which have no outgoing edges,
   into $v_f$ (which is defined to belong to level $D$);
3. For $d := 1, \ldots, D-1$ do: Merge all nodes $j$ in level
   $d$ for which $m_j < m_0$ into $small(d)$;
4. For $d := 0, \ldots, D-1$ and for every $j$ in level $d$ do:
   (a) $\forall \sigma \in \Sigma$: If $m_j(\sigma) = 0$ then add an edge labeled
       $\sigma$ from $j$ to $small(d+1)$ ;
   (b) If $m_j(\zeta) = 0$ then add an edge labeled $\sigma$ from
       $j$ to $v_f$ (set $j \xrightarrow{\zeta} v_f$);
```

```
Subroutine GraphToPFA($G, \widehat{M}$)
1. Let $G$ be the underlying graph of $\widehat{M}$;
2. Let $\hat{q}_0$ be the state corresponding to $v_0$, and let $\hat{q}_f$
   be the state corresponding to $v_f$;
3. For every state $\hat{q}$ in $\widehat{M}$ and for every $\sigma \in \Sigma \bigcup \{\zeta\}$:
   $\hat{\gamma}(\hat{q}, \sigma) := (m_v(\sigma)/m_v)(1 - (|\Sigma| + 1)\gamma_{min}) + \gamma_{min}$ ,
   where $v$ is the node corresponding to $\hat{q}$ in $G$.
```

For the sake of simplicity of the pseudo-code that follows, we associate with each node in a graph G_i, a number in $\{1, ..., |G_i|\}$. The algorithm proceeds level by level. At each level, it searches for pairs of nodes, belonging to that same level, which can be folded. It does so by calling the function **Similar** on every pair of nodes j and j', whose counts, m_j and $m_{j'}$, are above the threshold m_0. The function **Similar** returns *similar* if and only if there is no string s in $S_{gen}(j)$ or $S_{gen}(j')$, such that $|m_j(s)/m_j - m_{j'}(s)/m_{j'}| \geq \mu/2$. If the function returns *similar*, then the algorithm merges j and j' using the routine **Fold**. Each call to **Fold** creates a new (smaller) graph. When level D is reached, the last graph, G_N, is transformed into G_{N+1} as described in the routine **AddSlack**. The final graph, G_{N+1} is then transformed into an APFA while smoothing the transition probabilities (Procedure **GraphToPFA**).

The function **Similar** is implemented as a recursive function. It receives four parameters, u, v, p_u, and p_v, where u and v are two nodes, and $0 \leq p_u, p_v \leq 1$. At the top level of the recursion, **Similar** is always called by **Learn-Acyclic-PFA** for some pair of nodes j, j' which the algorithm is interested in comparing, where p_j and $p_{j'}$ are both set to one. In deeper levels of the recursion, p_u and p_v are the probabilities of reaching u and v starting from j and j', respectively. If for some u and v, which correspond to equivalent paths in the two respective subtrees of j and j', p_u and p_v are found to be very different, the function returns *non-similar* for u and v. Otherwise, the function is called recursively for each pair of respective children of u and v. If some child is missing, then the node is set to be *undefined* (where its corresponding probability, by definition, is 0). Finally, if all respective node pairs are found to be similar, then u and v are declared to be similar.

5. ANALYSIS OF THE LEARNING ALGORITHM

In this section we state and prove our main theorem regarding the correctness and efficiency of the learning algorithm **Learn-Acyclic-PFA**, described in Section 4.

THEOREM 1. *For every given distinguishability parameter $\mu > 0$, for every μ-distinguishable target APFA M, and for every given confidence parameter $\delta > 0$, and accuracy parameter $\varepsilon > 0$, Algorithm* **Learn-Acyclic-PFA** *outputs a hypothesis APFA, $\widehat{M}$, such that with probability at least $1 - \delta$, $\widehat{M}$ is an ε-good hypothesis with respect to M. The running time of the algorithm is polynomial in $1/\varepsilon$, $\log 1/\delta$, $1/\mu$, n, D, and $|\Sigma|$.*

We would like to note that for a given accuracy parameter ε, we may slightly weaken the requirement that M be μ-distinguishable. Let us say that a state q in M is *non-negligible* if the probability at reaching q starting from q_0 is greater than some ε (which is a function of ε, μ, and n). Then it suffices to require that every pair at non-negligible states be μ-distinguishable. For the sake of simplicity, we give our analysis under the slightly stronger assumption.

Based on Lemma 2.1, we may assume without loss of generality that M is a leveled APFA with at most n state in each of its D levels. We add the following notation.

- For a state $q \in Q_d$,

  — $W(q)$ denotes the set of *all* strings in Σ^d which *reach* q; $P^M(q) \overset{\text{def}}{=} \sum_{s \in W(q)} P^M(s)$.

— m_q denotes the number of strings *in the sample* (including repetitions) which *pass through* q, and for a string s, $m_q(s)$ denotes the number of strings in the sample which pass through q and continue with s. More formally,

$$m_q(s) = |\{t\colon t \in S,\ t = t_1 s t_2, \text{ where } \tau(q_0, t_1) = q\}_{multi}|.$$

• For a state $\hat{q} \in \hat{Q}_d$, $W(\hat{q})$, $m_{\hat{q}}$, $m_{\hat{q}}(s)$, and $P^{\hat{M}}(\hat{q})$ are defined similarly. For a node v in a graph G_i constructed by the learning algorithm, $W(v)$ is defined analogously. (Note that m_v and $m_v(s)$ were already defined in Section 4.)

• For a state $q \in Q_d$ and a node v in G_i, we say that v *corresponds* to q, if $W(v) \subseteq W(q)$.

5.1. Motivating Discussion

In order to prove Theorem 1, we first need to define the notion of a *good* sample with respect to a given target (leveled) APFA, M. We prove that with high probability a sample generated by M is good. We then show that if a sample is good then our algorithm constructs a hypothesis APFA which has the properties stated in the theorem. An important ingredient of the proof is the definition of a good sample, which is somewhat complex. We therefore provide below some intuition into what led us to this definition.

The heart of our algorithm is the folding operation and the similarity test that precedes it. We want to show that, on one hand, the algorithm *does not* fold pairs of nodes which correspond to two *different* states and, on the other hand, it *does* fold most pairs of nodes that correspond to the *same* state. By the latter we essentially mean the following. First, every pair of nodes u and v which correspond to the same state and for which $m_u \geqslant m_0$ and $m_v \geqslant m_0$, are folded together. Second, in the final directed graph, which is used to construct our hypothesis, the total fraction of sample strings which pass through nodes u for which $m_u < m_0$ is small. This ensures that for each state q in M for which $P^M(q)$ is non-negligible, there exists a single state $\hat{q}$ in $\hat{M}$, such that:

1. A large fraction of the strings (among all possibly generated strings) that pass through q indeed pass through $\hat{q}$;
2. $\gamma(q, \cdot)$ is very "close" to $\hat{\gamma}(\hat{q}, \cdot)$.

It will then follow that $\hat{M}$ is a good hypothesis with respect to M.

Consider now the similarity test. Whenever the algorithm tests whether two nodes, u and v, are similar, it compares the statistical properties of the corresponding multisets of strings $S_{gen}(u)$ and $S_{gen}(v)$, which "originate" from the two nodes, respectively. Thus, we would like to ensure that if both multisets are of substantial size, then each will be *typical* to the state it was generated from (assuming there exists one such single state for each node). Namely, we ask that the relative weight of any prefix of a string in each of the multisets will not deviate much from the probability assigned to the prefix when starting from the corresponding state. It will then follow (based on the μ-distinguishability of pairs of states) that for every such pair of nodes, the two nodes will be folded if and only if they correspond to the same state. In addition, we need to ensure that most of the multisets $S_{gen}(u)$ are large enough, namely, that the size of most of the multisets is larger than the threshold m_0. The question is how to characterize these multisets of strings so that we can apply a probabilistic argument to prove that they are typical (and large) with high probability.

To gain more intuition concerning the difficulty of this task and to obtain some insight into our solution, consider a particular state q in a given level d of the target automaton and denote by $S_{gen}(q)$ the multiset of suffixes of sample strings which were generated from q. Let G_{i_d} be the first graph in which the algorithm starts folding nodes in level d. Assume that each pair of nodes which were folded previously correspond to the same state (where the notion of correspondence was defined above). Let $v_1, ..., v_k$ be the nodes in G_{i_d} which correspond to q. These nodes induce a partition of $S_{gen}(q)$ into the multisets $S_{gen}(v_1), ..., S_{gen}(v_k)$. It is clear that if $S_{gen}(q)$ is large enough, then, since the strings were generated independently from q, we can apply Chernoff bounds to get that with high probability $S_{gen}(q)$ is a typical sample of strings originating from q. As noted previously, by "typical" we mean that the relative counts of strings in $S_{gen}(q)$ are close to their expectations. However, we want to know that *each* of the $S_{gen}(v_i)$'s is a typical sample of q so that we will not make mistakes when merging nodes in that level. The problem is that the partition of $S_{gen}(q)$ into the $S_{gen}(v_i)$'s depends on previous folding steps of the algorithm, and it is clearly not true that *every* partition of $S_{gen}(q)$ preserves the statistical properties of q. However, the graphs constructed by the algorithm via the folding operation do not induce arbitrary partitions. Furthermore, we are able to characterize the possible partitions in terms of a class of automata which we call M's *reference class* and which we denote by $\mathscr{M}$.

The reference class, $\mathscr{M}$, is a class of automata which are all equivalent to M, in the sense that they define exactly the same probability distribution over strings. However, these automata have different underlying graphs and can be viewed as *redundant* forms of M. In particular, every state of M may have several copies in an automaton $M' \in \mathscr{M}$. The idea is that the nodes in level d of G_{i_d} (i.e., before the algorithm folds nodes in level d) correspond (via a one-to-one mapping) to states in level d of some automaton M' $\in \mathscr{M}$. Given that there exists such a mapping, let us assume that with high probability for every $M' \in \mathscr{M}$ and for every state q' of M' (which is a copy of a state q of M), a sample generated by M is typical of q'. Then, this implies that for every state q of M, the multisets $\{S_{gen}(v_i)\}$ are typical of q, as desired. In our definition, we actually relax the requirement and ask that the sample be typical only for the copies q' of q which are reached in M' with non-negligible probability. These copies, which we refer to as *dominant* copies, correspond to nodes v in G_{i_d} for which m_v is above the threshold m_0 and which the algorithm considers for merging. Analogously, the *non-dominant* copies correspond to nodes whose counts are below the threshold. In our analysis, we show that in the automata corresponding to the graphs generated by the algorithm, the total weight of the non-dominant copies of every state q is small.

In addition to the distinction between dominant and non-dominant copies, we make a distinction between *major* copies and *minor* copies. Major copies can be either dominant or non-dominant, while minor copies are always non-dominant. This distinction corresponds to the difference between nodes in the algorithm's graphs whose immediate ancestors (in the previous level) were merged (i.e., had counts above threshold and hence corresponded to dominant copies) and nodes whose immediate ancestors already had counts below threshold. Intuitively, major copies "have the potential" to be dominant, while minor copies are non-dominant by definition. Furthermore, the choice of the dominant copies among the major copies is what defines an automaton in the reference class. We note that one of the important features of the automata in the reference class is that every level in an automaton $M' \in \mathscr{M}$ is devised to correspond to the same level in a graph constructed by the algorithm *before* any merges were performed in that level, but *after* all merges were performed in the previous level. Thus, these automata do not look exactly like any one of the graphs constructed by the algorithm. However, all dominant major copies of a particular state q have edges going to the same major copies in the next level (thus, in a way they behave like a single merged state), while all non-dominant copies (both major and minor) are roots of (disjoint) $|\Sigma|$-ary trees. See Fig. 2 for an illustration of the different types of copies of states.

5.2. A Good Sample—Formal Definitions

Given a target APFA M, let $\mathscr{M} = \{M' \mid M' = (Q', q'_0, \{q'_f\}, \Sigma, \tau', \gamma', \zeta)\}$ be the set of APFAs which satisfy the following conditions:

1. For each state q in M there exist several *copies* of q in M', each uniquely labeled. q'_0 is the only copy of q_0, and we allow there to be a set of final states $\{q'_f\}$, all copies of q_f. If q' is a copy of q then for every $\sigma \in \Sigma \cup \{\zeta\}$,

   (a) $\gamma'(q', \sigma) = \gamma(q, \sigma)$;

   (b) if $\tau(q, \sigma) = t$, then $\tau'(q', \sigma) = t'$, where t' is a copy of t.

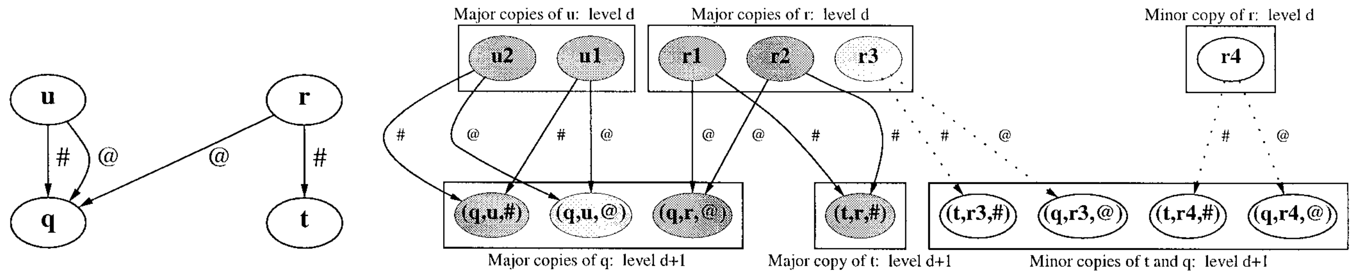
Note that the above restrictions on γ' and τ' ensure that the probability distributions generated by M and M', respectively, are the same. That is, $\forall s \in \Sigma^*\zeta$, $P^{M'}(s) = P^M(s)$.

2. A copy of a state q may be either *dominant* or *non-dominant*. Non-dominant copies are either *major* or *minor*, and dominant copies are *always* major.

3. For each state q, and for every symbol σ and state r such that $\tau(r, \sigma) = q$, there exists a unique major copy of q labeled by (q, r, σ) (where this copy may be either dominant or non-dominant). There are no other major copies of q. Each minor copy of q is labeled by (q, r', σ), where r' is a non-dominant (either major or minor) copy of r (and as before $\tau(r, \sigma) = q$). A state may have no minor copies, and its major copies may be all dominant or all non-dominant.

4. For each dominant copy q' of q and for every $\sigma \in \Sigma \cup \{\zeta\}$, if $\tau(q, \sigma) = t$, then $\tau'(q', \sigma) = (t, q, \sigma)$. Thus, for each symbol σ, *all* σ transitions from the dominant copies of q are to the *same* major copy of t. The starting state q'_0 is always dominant.

5. For each non-dominant (either major or minor) copy q' of q, and for every symbol σ, if $\tau(q, \sigma) = t$ then $\tau'(q', \sigma) = (t, q', \sigma)$, where, as defined in item (2) above, (t, q', σ) is a minor copy of t. Thus, each non-dominant major copy of q is the root of a $|\Sigma|$-*ary* tree, and all its descendants are (non-dominant) minor copies.



u
r
q
t
#
@
@
#
Major copies of u: level d
u2
u1
Major copies of r: level d
r1
r2
r3
Minor copy of r: level d
r4
#
@
#
@
@
@
#
#
#
@
#
@
(q,u,#)
(q,u,@)
(q,r,@)
(t,r,#)
(t,r3,#)
(q,r3,@)
(t,r4,#)
(q,r4,@)
Major copies of q: level d+1
Major copy of t: level d+1
Minor copies of t and q: level d+1

FIG. 2. Left: Part of the original automaton, M, that corresponds to the copies on the right part of the figure. Right: The different types of copies of M's states: the state u has only major copies, u_1 and u_2, that are both dominant and have each an edge labeled by # going to the major copy of q, $(q, u, \#)$, and an edge labeled by @ going to the major copy of q, $(q, u, @)$. State r has three major copies, r_1, r_2, and r_3, and one minor copy, r_4. States r_1 and r_2 are dominant while state r_3 is non-dominant. States r_1 and r_2 have edges going to a major copy of q and to a major copy of t. States r_3 and r_4 each have edges going to different minor copies of q and t.

An illustrative example of the types of copies of states is depicted in Fig. 2.

By the definition above, each APFA in $\mathcal{M}$ is fully characterized by the choices of the sets of dominant copies among the major copies of each state. Namely, given such a choice, it is determined that for each state q, all non-dominant major copies are roots of complete $|\Sigma|$-ary trees (of minor states), and for each symbol σ, all dominant copies of q have edges labeled by σ going to the major copy (r, q, σ) of r, where $r=\tau(q, \sigma)$. Since the number of major copies of a state q is exactly equal to the number of transitions going into q in M, and is thus bounded by $n\,|\Sigma|$, there are at most $2^{n\,|\Sigma|}$ such possible choices for every state. There are at most n states in each level, and hence the size of $\mathcal{M}$ is bounded by $((2^{|\Sigma|\,n})^n)^D = 2^{|\Sigma|\,n^2 D}$. As we show in Lemma 5.2, if the sample is good, then there exists a correspondence between some APFA in $\mathcal{M}$ and the graphs our algorithm constructs. We use this correspondence to prove Theorem 1.

Definition 5.1. A sample S of size m is $(\varepsilon_0, \varepsilon_1, \delta)$-*good* with respect to M if for every $M' \in \mathcal{M}$ and for every state $q' \in Q'$:

1. if $P^{M'}(q') \geqslant 2\varepsilon_0$, then $m_{q'} \geqslant m_0$, where

$$m_0 = \frac{|\Sigma|\, n^2 D^2 + 2D \ln(8(|\Sigma|+1)) + \ln 1/\delta}{\varepsilon_1^2};$$

2. if $m_{q'} \geqslant m_0$, then for every string s,

$$|m_{q',s}/m_{q'} - P_{q'}^{M'}(s)| \leqslant \varepsilon_1.$$

Note that the second item in the above definition is relevant not only to states q', for which $P^{M'}(q') \geqslant 2\varepsilon_0$ (i.e., the condition in the first item holds), but also to states for which this condition does not hold, but nonetheless, their corresponding count, $m_{q'}$, is above the threshold m_0.

Lemma 5.1. *With probability at least* $1-\delta$, *a sample of size*

$$m \geqslant \max\left(\frac{|\Sigma|\, n^2 D + \ln(2D/(\varepsilon_0 \delta))}{2\varepsilon_0^2}, \frac{m_0}{\varepsilon_0}\right)$$

is $(\varepsilon_0, \varepsilon_1, \delta)$-*good with respect to M.*

The proof of Lemma 5.1 is derived by several simple probabilistic arguments and is provided in the Appendix.

Lemma 5.2. *If the sample is* $(\varepsilon_0, \varepsilon_1, \delta)$-*good, for* $\varepsilon_1 < \mu/4$, *then there exists an APFA* $M' \in \mathcal{M}$, $M' = (Q', q'_0, \{q'_f\}, \Sigma, \tau', \gamma', \zeta)$, *for which the following holds. Let* G_{i_d} *denote the first graph in which we consider folding nodes in level d. Then, for every level d, there exists a* one-to one *mapping* Φ_d *from the nodes in the dth level of* G_{i_d}, into Q'_d, *such that for every v in the dth level of* G_{i_d}, $W(v) = W(\Phi_d(v))$. *Furthermore,* $q' \in Q'$ *is a dominant major copy* iff $m_{q'} \geqslant m_0$.

The complete proof of the lemma is given in the Appendix. Below we provide a sketch.

Proof Sketch. The correctness of Lemma 5.2 is proved by induction on d. The automaton M' is constructed in the course of the induction, by choosing for each level d and for each state $q \in Q_d$, the dominant copies of q among its major copies. Recall that such a choice uniquely determines each automaton in $\mathcal{M}$. For the base case of $d=1$, we only need to observe that $G_{i_1} = G_0$ is the sample tree, in which we have at most $|\Sigma|$ nodes in level 1. For each symbol σ, we map the node which is reached from the starting node by an edge labeled σ to the unique major state (q, q_0, σ), where $q=\tau(q_0, \sigma)$. We then choose the dominant copies to match the requirement of the lemma. Namely, the dominant copies are exactly those which are mapped via the reverse mapping Φ_d^{-1} to nodes v for which $m_v \geqslant m_0$.

In the induction step, we consider separately the case in which node v is a result of merges of nodes in level $d-1$ and the case in which it is not. It is not hard to verify that in both cases, based on the folding operation, there is only a single edge (labeled by a single symbol σ) entering v. Consider first the case in which v had immediate ancestors which were merged. Since the nodes that were merged had counts above threshold, by the induction hypothesis they must be mapped to dominant copies of some state r in level $d-1$. Thus, v is mapped to the corresponding major copy (q, r, σ) (where $q=\tau(r, \sigma)$), and this copy is chosen to be dominant only if $m_v \geqslant m_0$. In case v is not a result of any merges in level $d-1$, let u be the node in level $d-1$ from which there is an edge labeled σ going into v. The subcase in which $m_u \geqslant m_0$ is similar to the case treated above (where v is the result of merges in level $d-1$). In the subcase $m_u < m_0$, by the induction hypothesis, u is mapped to a non-dominant copy r' of r, which implies that v should be mapped to the non-dominant (minor) copy (q, r', σ). ▌

5.3. *Proof of Theorem* 1

The proof of Theorem 1 is based on the following lemma, in which we show that for every state q in M there exists a *representative* state $\hat{q}$ in $\hat{M}$ that has significant weight and for which $\hat{\gamma}(\hat{q}, \cdot) \approx \gamma(q, \cdot)$.

Lemma 5.3. *If the sample is* $(\varepsilon_0, \varepsilon_1, \delta)$-*good for*

$$\varepsilon_1 < \min(\mu/4, \varepsilon^2/8(|\Sigma|+1)),$$

then for any $\varepsilon_3 \leqslant 1/(2D)$, *and for any* $\varepsilon_2 \geqslant 2n\,|\Sigma|\,\varepsilon_0/\varepsilon_3$, *we have the following. For every level d and for every state* $q \in Q_d$, *if* $P^M(q) \geqslant \varepsilon_2$ *then there exists a state* $\hat{q} \in \hat{Q}_d$ *such that*:

1. $P^M(W(q) \cap W(\hat{q})) \geqslant (1 - d\varepsilon_3)\, P^M(q)$,
2. *for every symbol* σ, $\gamma(q, \sigma)/\hat{\gamma}(\hat{q}, \sigma) \leqslant 1 + \varepsilon/2$.

The complete proof of the lemma is given in the Appendix. Here we only note that for both claims we rely on the relation that is shown in Lemma 5.2 between the graphs constructed by the algorithm and some APFA M' in $\mathscr{M}$. We show that the total weight in M' of the dominant copies of every state $q \in Q_d$ for which $P^M(q) \geqslant \varepsilon_2$ is at least $1 - d\varepsilon_3$ of the weight of q. This is shown by induction on d. Claim (1) directly follows, and for proving Claim (2) we apply the goodness of the sample.

Proof of Theorem 1. We prove the theorem based on Lemma 5.3. For brevity of the following computation, we assume that M and $\widehat{M}$ generate strings of length exactly D. This can be assumed without loss of generality, since we can require that both APFAs "pad" each shorter string they generate, with a sequence of ζ's, with no change to the KL-divergence between the APFAs.

$$\begin{aligned}
&\mathscr{D}_{KL}(P^M \,\|\, P^{\widehat{M}})\\
&\quad= \sum_{\sigma_1 \cdots \sigma_D} P^M(\sigma_1 \cdots \sigma_D) \log \frac{P^M(\sigma_1 \cdots \sigma_D)}{P^{\widehat{M}}(\sigma_1 \cdots \sigma_D)}\\
&\quad= \sum_{\sigma_1} \sum_{\sigma_2 \cdots \sigma_D} P^M(\sigma_1)\, P^M(\sigma_2 \cdots \sigma_D \mid \sigma_1)\\
&\qquad \times \left[\log \frac{P^M(\sigma_1)}{P^{\widehat{M}}(\sigma_1)} + \log \frac{P^M(\sigma_2 \cdots \sigma_D \mid \sigma_1)}{P^{\widehat{M}}(\sigma_2 \cdots \sigma_D \mid \sigma_1)} \right]\\
&\quad= \sum_{\sigma_1} P^M(\sigma_1) \log \frac{P^M(\sigma_1)}{P^{\widehat{M}}(\sigma_1)} + \sum_{\sigma_1} P^M(\sigma_1)\\
&\qquad \times \mathscr{D}_{KL}(P^M(\sigma_2 \cdots \sigma_D \mid \sigma_1) \,\|\, P^{\widehat{M}}(\sigma_2 \cdots \sigma_D \mid \sigma_1))\\
&\quad= \sum_{\sigma_1} P^M(\sigma_1) \log \frac{P^M(\sigma_1)}{P^{\widehat{M}}(\sigma_1)}\\
&\qquad + \sum_{\sigma_1} P^M(\sigma_1) \sum_{\sigma_2} P^M(\sigma_2 \mid \sigma_1) \log \frac{P^M(\sigma_2 \mid \sigma_1)}{P^{\widehat{M}}(\sigma_2 \mid \sigma_1)}\\
&\qquad + \cdots\\
&\qquad + \sum_{\sigma_1 \cdots \sigma_d} P^M(\sigma_1 + \cdots \sigma_d) \sum_{\sigma_{d+1}} P^M(\sigma_{d+1} \mid \sigma_1 \cdots \sigma_d)\\
&\qquad \times \log \frac{P^M(\sigma_{d+1} \mid \sigma_1 \cdots \sigma_d)}{P^{\widehat{M}}(\sigma_{d+1} \mid \sigma_1 \cdots \sigma_d)}\\
&\qquad + \cdots\\
&\qquad + \sum_{\sigma_1 \cdots \sigma_{D-1}} P^M(\sigma_1 \cdots \sigma_{D-1}) \sum_{\sigma_D} P^M(\sigma_D \mid \sigma_1 \cdots \sigma_{D-1})\\
&\qquad \times \log \frac{P^M(\sigma_D \mid \sigma_1 \cdots \sigma_{D-1})}{P^{\widehat{M}}(\sigma_D \mid \sigma_1 \cdots \sigma_{D-1})}\\
&\quad= \sum_{d=0}^{D-1} \sum_{q \in Q_d} \sum_{\hat{q} \in \hat{Q}_d} P^M(W(q) \cap W(\hat{q}))\\
&\qquad \times \sum_{\sigma} P_q^M(\sigma) \log \frac{P_q^M(\sigma)}{P_{\hat{q}}^{\widehat{M}}(\sigma)}\\
&\quad= \sum_{d=0}^{D-1} \sum_{q \in Q_d} P^M(q) \sum_{\hat{q} \in \hat{Q}_d} P^M(W(q) \cap W(\hat{q}))/P^M(q)\\
&\qquad \times \sum_{\sigma} P_q^M(\sigma) \log \frac{P_q^M(\sigma)}{P_{\hat{q}}^{\widehat{M}}(\sigma)}\\
&\quad\leqslant \sum_{d=0}^{D-1} \sum_{q \in Q_d,\, P^M(q) < \varepsilon_2} P^M(q) \log(1/\gamma_{min})\\
&\qquad + \sum_{d=0}^{D-1} \sum_{q \in Q_d,\, P^M(q) \geqslant \varepsilon_2} P^M(q)\\
&\qquad \times [(1 - d\varepsilon_3) \log(1 + \varepsilon/2) + d\varepsilon_3 \log(1/\gamma_{min})]\\
&\quad\leqslant (nD\varepsilon_2 + D^2\varepsilon_3) \log(1/\gamma_{min}) + \varepsilon/2.
\end{aligned}$$

If we choose ε_2 and ε_3 so that $\varepsilon_2 \leqslant \varepsilon/(4n\, D \log(1/\gamma_{min}))$ and $\varepsilon_3 \leqslant \varepsilon/(4D^2 \log(1/\gamma_{min}))$, then the expression above is bounded by ε, as required. Adding the requirements on ε_2 and ε_3 from Lemma 5.3, we get the following requirement on ε_0:

$$\varepsilon_0 \leqslant \varepsilon^2/(32n^2\, |\Sigma|\, D^3 \log^2(4(|\Sigma| + 1)/\varepsilon)).$$

Applying Lemma 5.1, together with the above bound on ε_0, while using the definition of m_0 provided in Definition 5.1 and the requirement on ε_1 given in Lemma 5.3, we get that a sufficient sample size is

$$\begin{aligned}
m = O(&n^4 D^4\, |\Sigma|^2\, \varepsilon^{-2} \ln(nD\, |\Sigma|\, \varepsilon^{-1}\delta^{-1})\\
&\times \max(\mu^{-2}, |\Sigma|\, \varepsilon^{-4}, n^2\, |\Sigma|^2\, D^3\varepsilon^{-2})).
\end{aligned}$$

The running time of the algorithm when completed successfully can be bounded by $n^2\, |\Sigma|^2\, Dm_0$ as follows. Whenever the algorithm checks whether two nodes are similar, and whenever two nodes are folded, the corresponding procedures consider at most every node in the graph that can be reached from one of the two nodes which are compared/folded. Thus, in every comparing/folding operation, the total number of nodes considered is $O(Dm_0)$. For each level d, if for every state in M, all nodes corresponding to M in level d whose counts are above threshold are merged, then in the next level we have at most $|\Sigma|\, n$ nodes whose counts are above threshold. Since we compare/fold at most every pair of such nodes and there are D levels, we compare/fold at most $n^2 D^2\, |\Sigma|$ pairs of nodes. ∎

6. AN ON-LINE VERSION OF THE ALGORITHM

In this section we describe an *on-line* version of our learning algorithm. We start by defining our notion of *on-line learning* in the context of learning distributions on strings.

6.1. An On-line Learning Model

In the on-line setting, the algorithm is presented with an infinite sequence of *trials*. At each time step, t, the algorithm receives a trial string $s^t = s_1 \cdots s_{\ell-1}\zeta$ generated by the target APFA M, and it should output the probability assigned by its current hypothesis, H_t, to s^t. The algorithm then transforms H_t into H_{t+1}. The hypothesis at each trial need not be an APFA, but may be any data structure which can be used in order to define a probability distribution on strings. In the transformation from H_t into H_{t+1}, the algorithm uses only H_t itself and the new string s^t. The algorithm should be efficient in the sense that the time for computing the probability assigned to s^t by H_t and the time for transforming H_t to H_{t+1} should be bounded by a polynomial in $1/\mu$, n, D, $|\Sigma|$, and $\log(1/\Delta)$ (where Δ is defined shortly). Let the *error* of the algorithm on s^t, denoted by $err_t(s^t)$, be defined as $\log(P^M(s^t)/P^{H_t}(s^t))$. We shall be interested in the *average* error $Err_t \overset{\text{def}}{=} 1/t \sum_{t' \leqslant t} err_{t'}(s^{t'})$.

We allow the algorithm to make an *unrecoverable error* at any stage t, with total probability that is bounded by Δ.[6] We ask that there exist functions $\delta(t, \mu, n, D, |\Sigma|, \Delta)$ and $\varepsilon(t, \mu, n, D, |\Sigma|, \Delta)$, such that the following hold: The function $\delta(t, \mu, n, D, |\Sigma|, \Delta)$ is of the form $\beta_1(\mu, n, D, |\Sigma|, \Delta)\, 2^{t^{-\alpha_1}}$, where β_1 is a polynomial in $1/\mu$, n, D, $|\Sigma|$, and $\log(1/\Delta)$, and $0 < \alpha_1 < 1$; and $\varepsilon(t)$ is of the form $\beta_2(\mu, n, D, |\Sigma|, \Delta)\, t^{-\alpha_2}$, where β_2 is a polynomial in $1/\mu$, n, D, $|\Sigma|$, and $\log(1/\Delta)$, and $0 < \alpha_2 < 1$. Since we are mainly interested in the dependence of these functions on t, let them be denoted for short by $\delta(t)$, and $\varepsilon(t)$. For every trial t, if the algorithm has not made an unrecoverable error prior to that trial, then with probability at least $1-\delta(t)$, the average error is small, namely $Err_t \leqslant \varepsilon(t)$. We thus require that as the algorithm gets more trial strings, it makes better predictions with high probability. (Note though that we allow that algorithm an initial "period of grace" in which it does not have to perform well at all.) Furthermore, we require that the size of the hypothesis H_t be a *sublinear* function of t. This last requirement implies that an algorithm which simply remembers *all* trial strings, and each time constructs a new hypothesis "from scratch," is *not* considered an on-line algorithm.

[6] Allowing the algorithm to make such an unrecoverable error is due to an artifact of our algorithm that we were not able to overcome. As we shall see in the description of the algorithm, a decision to fold two nodes in a graph $G(t)$, which do not correspond to the same state in M, is an *unrecoverable* error. Since the algorithm does not backtrack and "unfold" nodes, the algorithm has no way of recovering from such a decision, and the probability assigned to strings passing through the folded nodes may be erroneous from that point on.

6.2. An On-line Learning Algorithm

We now describe how to modify the *batch* algorithm **Learn-Acyclic-PFA**, presented in Section 4, to make it an on-line algorithm. The pseudo-code for the algorithm follows this description. At each time t, our hypothesis is a graph $G(t)$, which has the same form as the graphs used by the batch algorithm. $G(1)$, the initial hypothesis, consists of a single root node v_0 where for every $\sigma \in \Sigma \cup \{\zeta\}$, $m_{v_0}(\sigma) = 0$ (and hence, by definition, $m_{v_0} = 0$). Given a new trial string s^t, the algorithm checks whether there exists a path corresponding to s^t in $G(t)$. If there are missing nodes and edges on the path, then they are added. The counts corresponding to the new edges and nodes are all set to 0. The algorithm then outputs the probability that a PFA defined based on $G(t)$ would have assigned to s^t. More precisely, let $s^t = s_1 \cdots s_\ell$, and let $v_0 \cdots v_\ell$ be the nodes on the path corresponding to s^t. Then the algorithm outputs the product

$$P^{H_t}(s^t) = \prod_{i=0}^{\ell-1} \left[\frac{m_{v_i}(s_{i+1})}{m_{v_i}} \left(1 - (|\Sigma| + 1)\, \gamma_{min}(t)\right) + \gamma_{min}(t) \right],$$

where $\gamma_{min}(t)$ is a decreasing function of t (whose form is discussed subsequently).

The algorithm increases by one the counts associated with the edges on the path corresponding to s^t in the updated $G(t)$. If for some node v on the path, $m_v \geqslant m_0$, then we execute stage (2) in the batch algorithm, starting from $G_0 = G(t)$, and letting $d(0)$ be the depth of v, and D be the depth of $G(t)$. We let $G(t+1)$ be the final graph constructed by stage (2) of the batch algorithm.

THEOREM 2. *Algorithm* **On-Line-Learn-Acyclic-PFA** *is an efficient on-line learning algorithm for APFAs.*

Proof Idea. The proof of the theorem essentially follows the same arguments used in the proof of Theorem 1 and we therefore present the main new ideas. The first key observation is the following. Consider some trial t. Assume that up until trial t the following was true: for every pair of nodes u and v such that $m_u \geqslant m_0$ and $m_v \geqslant m_0$, u and v are folded into a single node if and only if they correspond to the same state in M. Assume also that the same would be true had we run the batch algorithm on a sample consisting of the first t trial strings (which is essentially a prerequisite for the successful completion of the batch algorithm). Then, $G(t)$ is the same as the graph we would have obtained had we run the batch algorithm on the set of trial strings observed up until trial t. This can easily be verified by induction on the levels of the graph. We thus define a *bad* event to be an event in which the on-line algorithm calls the function **Similar** on two nodes (whose counts are at least m_0) and either the function returns *similar* while the two nodes correspond to different states or the function returns *non-similar* while the

Algorithm On-line-Learn-Acyclic-PFA

```
1. Initialize: $t := 1$, $G(1)$ is a graph with a single node $v_0$, $\forall\sigma \in \Sigma\bigcup\{\zeta\}$, $m_{v_0}(\sigma) = 0$;
2. Repeat:
   (a) Receive the new string $s^t$;
   (b) If there does not exist a path in $G(t)$ corresponding to $s^t$, then add missing edges and nodes
       to $G(t)$, and set their corresponding counts to 0.
   (c) Let $v_0 \dots v_\ell$ be the nodes on the path corresponding to $s^t$ in $G(t)$;
   (d) Output: $P^{H_t}(s^t) = \Pi_{i=0}^{\ell-1}(\frac{m_{v_i}(s_{i+1})}{m_{v_i}}(1-(|\Sigma|+1)\gamma_{min}(t)) + \gamma_{min}(t))$;
   (e) Add 1 to the count of each edge on the path corresponding to $s^t$ in $G(t)$;
   (f) If for some node $v_i$ on the path $m_{v_i} \geq m_0$ then do:
       i. $i := 0$, $G_0 = G(t)$, $d(0) =$ depth of $v_i$, $D =$ depth of $G(t)$;
       ii. Execute step (2) in Learn-Acyclic-PFA;
       iii. $G(t+1) := G_i$, $t := t+1$ .
```

nodes correspond to the same state. Note that the first type of bad event corresponds to an unrecoverable error.

Based on the observation above, if a bad event does not occur until trial t, we can use the result we have for the batch algorithm to get bounds on $\varepsilon(t)$ and $\delta(t)$. In the analysis of the batch algorithm, for every given ε and δ we get a lower bound on m (whose role is taken by t) that ensures error ε with confidence $1-\delta$. Since this bound was polynomial in $1/\varepsilon$ and $\log(1/\delta)$, we can define a pair of functions $\varepsilon(t)$ and $\delta(t)$, which decrease with t as required. To be more precise, $\varepsilon(t)$ is defined to be a bound on the average error Err_t. However, Err_t is an average of random variables $\{err_t\}$, where the expectation of err_t is the KL-divergence between M and H_t. Since these random variables are independent (as the strings s^t are chosen independently), with probability exponential in t, Err_t will not deviate by more than $O(\sqrt{\log(1/t)/t})$ from $\sum_{t' \leqslant t} \mathscr{D}_{KL}(P^M \parallel P^{H_{t'}})$. Finally, by our analysis of Theorem 1, we know that if a bad event does not occur, then for some constants α_1 and α_2, the KL-divergence between M and H_t is $O(t^{-\alpha_2})$ with probability at least $1 - O(2^{t^{\alpha_1}})$ (where the $O(\cdot)$ notation takes into account the dependence on all other parameters of the problem).

The intuition behind this behavior of the algorithm is the following. If no bad event occurs, we expect that as t increases, we will encounter nodes that correspond to states with decreasing weights, and our predictions become more reliable in the sense that $m_v(\sigma)/m_v$ gets closer to its expectation (and the probability of a large error decreases). The parameter $\gamma_{min}(t)$ is defined to have the same functional form with $\varepsilon(t)$ as γ_{min} has with ε in the batch algorithm.

We thus need to show that for an appropriate m_0, the probability that a bad event occurs at any point in time is at most Δ. In the on-line algorithm, as opposed to the batch algorithm, the folding operation does not proceed level by level. It follows that the graphs constructed cannot be mapped to automata in the reference class $\mathscr{M}$ as defined in the batch algorithm. However, we can slightly modify the definition of $\mathscr{M}$ to cope with this problem. We now allow a subset of the dominant major copies of each state to be merged into a single *super* dominant major copy. Recall that automata in the batch algorithm's reference class had the property that their states (copies of states) in level d corresponded to nodes in level d of the algorithm's graph right after all merges were done in level $d-1$ but before any were done in level d. This worked well since we did not perform any foldings in level d after proceeding to level $d+1$. However, in the on-line algorithm, when executing step (2) of the batch algorithm following the receipt of a new trial string, we need to account for nodes (in level d) that are the result of merges (in the same level) which were performed in previous executions of step (2). Namely, we might consider merging nodes u and v, where u is the outcome of a merging of nodes u_1 and u_2 in level d (in a previous execution of step (2)). In such a case we map u to a super major copy in an automaton M'. It is easy to verify that by definition of our algorithm, there cannot be more than one such node corresponding to each state q.

Thus, we let each state q have at most one super dominant major copy which is labeled by a subset $\{(q, r, \sigma)\}$, where r and σ are such that $\tau(r, \sigma) = q$. The non-super major copies are labeled as before by triples (q, r, σ) (that do not belong to the subset labeling the super copy). The edges going out of this super copy are defined in the same way as for dominant major copies. Edges that previously entered major copies that belong to a super copy now enter the super copy. Thus, each automaton is now defined not only by the choice of dominant copies among the major copies of each state but also by the choice of a subset of the dominant major copies which defines the super copy. The size of this new $\mathscr{M}$ is therefore bounded by the square of the size of $\mathscr{M}$ as defined for the batch algorithm and is at most $2^{2|\Sigma| n^2 D}$. Note that M is the automaton in $\mathscr{M}$ in which each state has a super copy labeled by the set of all major copies and no other copies.

In order to bound the probability that a bad event occurs, we need to show that with probability at least $1-\Delta$, at any time t, for every $M' \in \mathscr{M}$, and for every state $q' \in Q'$, if $m_{q'} \geqslant m_0$, then for every string s, $|m_{q',s}/m_{q'} - P_{q'}^{M'}(s)| \leqslant \mu/4$. Let

$$m_0 = \frac{32(|\Sigma|\, n^2 D^2 + D \ln(16(|\Sigma|+1)) + \ln(16/(\Delta\mu^2)))}{\mu^2}.$$

Then similarly to the proof of Lemma 5.1, for a given q', if $m_{q'} = m_0 + x$ for some integer $x \geqslant 0$, then the probability that for some s, $|m_{q',s}/m_{q'} - P_{q'}^{M'}(s)| > \mu/4$ is at most

$$2^{-(x\mu^2/16)} \cdot \frac{\Delta}{8(|\Sigma|+1)^D\, 2^{2|\Sigma| n^2 D} (16/\mu^2)}.$$

Let us say in this case that q' is bad (otherwise it is good). After a new trial string is added to the current hypothesis, either $m_{q'}$ does not change or it increases by 1. If $m_{q'}$ does not change and prior to the receipt of the new string, q' was good, then it remains good. Otherwise, we upper bound the probability that it turned bad by the expression above. Hence, every time $m_{q'}$ increases by $16/\mu^2$, the probability that q' turned bad given that it was good before the new $(16/\mu^2)$ trial strings were added decreases by a factor of 1/2. If we sum these probabilities over an infinite sequence of trials to bound the probability that q' turns bad following any trial, we get that this probability is at most $\Delta/(4(|\Sigma|+1)^D\, 2^{2|\Sigma| n^2 D})$. Multiplying by the number of states in each automaton and the number of automata in $\mathscr{M}$ as done in Lemma 5.1, we get the desired bound.

We now bound the size of the hypotheses constructed by the algorithm and the running time of the algorithm (per trial). Let a node v be called *reliable* if $m_v \geqslant m_0$. As claimed above, with probability $1-\Delta$ we fold all reliable nodes which correspond to the same state. Thus, the number of reliable nodes is never larger than Dn. From every reliable node there are edges going to at most $|\Sigma|$ unreliable nodes. Each unreliable node is a root of a tree in which there are at most Dm_0 additional unreliable nodes. We thus get a bound of $O(D^2 n m_0)$ on the number of nodes in $G(t)$ which is *independent* of t. Since for every v and σ in $G(t)$, $m_v(\sigma) \leqslant t$, the counts on the edges contribute a factor of $\log t$ to the total size of the hypothesis. It remains to show that the algorithm is efficient. When transforming a hypothesis H_t to the next hypothesis H_{t+1}, in the worst case we consider folding all pairs of nodes from each level. Thus, if each operation on two rational numbers takes a single time step, then the bound on the size of the hypotheses gives us a polynomial bound on the time for computing each new hypothesis. The time for computing the probability assigned to each new trial node is $O(|\Sigma|\, D)$ since we need only consider nodes on the path corresponding to the trial string and their successors. ▮

7. APPLICATIONS

A slightly modified version of our learning algorithm was applied and tested on various problems involving modeling and analysis of spoken and written natural language. This modified version of the algorithm allows folding states from different levels, and hence the resulting hypothesis is more compact. One more small modification is that the algorithm folds nodes with small counts into the graph itself (instead of adding the extra nodes, *small*(d)). Here we give a brief overview of the usage of APFAs and their learning scheme for the following applications: (a) a part of a complete cursive handwriting recognition system; (b) pronunciation models for spoken words.

7.1. *Building Stochastic Models for Cursive Handwriting*

In [17], a dynamic encoding scheme for cursive handwriting based on an oscillatory model of handwriting was proposed and analyzed. The process described in [17] performs mapping from continuous pen trajectories to strings over a discrete set of symbols which efficiently encode cursive letters. These symbols are named *motor control commands*. There are 36 different motor control commands representing different characteristics of pen trajectories during the writing process. One of the symbols represents zero pen movement and it is used to denote "pen-ups" and end-of writing activity. This symbol serves as the final symbol (ζ) for building the APFAs for cursive letters as described subsequently.

Different Roman letters map to different sequences over these symbols. Moreover, since there are different writing styles and due to the existence of noise in the human motor system, the same cursive letter can be written in many

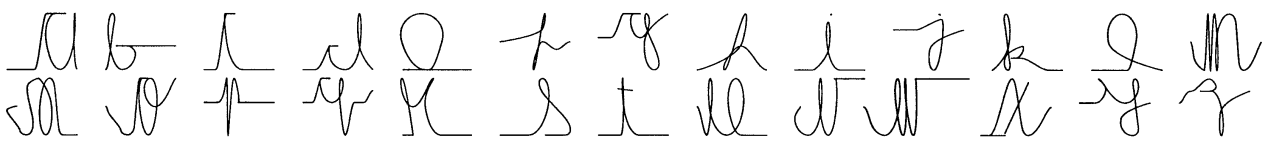


FIG. 3. Synthetic cursive letters created by random walks on the 26 letter APFAs.

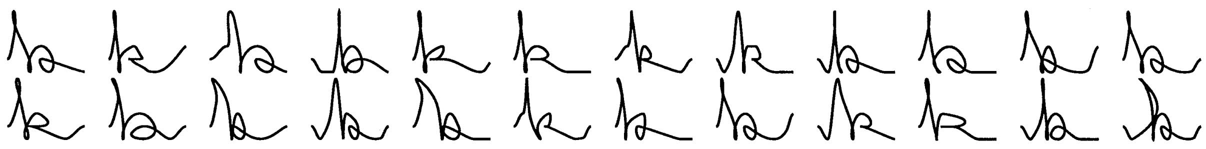


FIG. 4. Synthetic cursive letters created by random walks using the APFA that represents the letter k.

different ways. This results in different symbol sequences that represent the same letter. The first step in our cursive handwriting recognition system that is based on the above encoding is to construct stochastic models which approximate the distributions on sequences for each cursive letter. Given hundreds of examples of segmented cursive letters we applied the modified version of our algorithm to train 26 APFAs, one for each lower-case cursive English letter. In order to verify that the resulting APFAs have indeed learned the distributions on strings that represent the cursive letters, we performed a simple sanity check. Random walks on each of the 26 APFAs were used to synthesize motor control commands. An inverse oscillatory model was then used to translate these synthetic strings into pen trajectories. This process, known as *analysis-by-synthesis*, is widely used for testing the quality of speech models. A typical result of such random walks on the corresponding APFAs is given in Fig. 3. All the synthesized letters are clearly intelligible. The distortions are partly due to the compact representation of the dynamic model and not a failure of the learning algorithm.

We also performed a synthesis test which consisted of several different random walks using the same APFA. Typical results are shown in Fig. 4, where several synthetic letters, created using the APFA that represents the cursive letter k (which has a rather complex spatial structure), are depicted. The random walks created varying drawings which are all intelligible. Moreover, the letters start and end in several different ways. This indicates that the APFAs also capture effects of neighboring letters—a phenomenon similar to the coarticulation effects between phonemes in speech.

It is also interesting to look at the intermediate graphs built along the run of the APFA learning algorithm. Several of the intermediate graphs that represent the cursive letter l, and that were built when the algorithm was trained on segmented data, are shown in Fig. 5. The number of training sequences in this example is 195, and the initial graph has 209 nodes. The symbol labeling each edge in the figure is one of the possible motor control commands. The number on each edge is the count associated with the edge, that is, the number of times the edge was traversed in the training data. The top left graph in the figure is the initial sample tree, where all of its leaves are connected to a final node with an edge labeled by the final symbol. The intermediate graphs are drawn at every 10th iteration, left to right and top to bottom. The final graph, which is the result of 41 merging iterations, is drawn at the bottom part of the figure. The intermediate graphs at the start of the merging process are very "bushy," with no apparent structure. After 20 iterations, when more merges have been performed, a compact structure starts to appear. Finally, the resulting automaton has only 12 states with an interesting structure. All the outgoing edges from state 4 and the incoming edges into state 5 are labeled by symbols of the form $5 \times x$, $x \in \{0, 1, 2, 3, 4, 5\}$. Since all the paths from the start state to the final state must pass through either state 4 or state 5, it implies that a symbol of the form $5 \times x$ must be generated by *any* random walk using one of the existing paths in the automaton. This symbol corresponds to a high vertical modulation value (the top part of the letter l). Therefore, states 4 and 5 "encode" the fact that the letter l is characterized by a high vertical modulation value.

Given the set of 26 APFAs, representing the different cursive English letters, we can perform tasks such as segmentation of cursive words and recognition of unlabeled words. Here we briefly demonstrate how a new word can be broken into its different letter constituents. Recognition of completely unlabeled data is more involved, but can be performed efficiently using a higher-level language model (see [14] for an example of such a model). A complete description of the cursive handwriting recognition system is given in [16].

When a transcription of a cursively written word (i.e., the letters that constitute the word) is given, we find the most likely segmentation[7] of that word as follows. We first calculate the probability of each subsequence to be generated by each of the APFAs representing the letters appearing in the transcription. Then, using dynamic programming we efficiently find the most likely indices where the switches from one APFA to the next occur. An example of the result of a segmentation is depicted in Fig. 6, where the cursive word impossible, reconstructed from the motor control commands, is shown with its most likely segmentation. Note that the segmentation is temporal and hence letters are sometimes cut in the "middle" though the segmentation is correct.

The above segmentation procedure can be incorporated into an on-line learning setting as follows. We start with an initial stage where a relatively reliable set of APFAs for the cursive letters is constructed from *segmented* data. We then

[7] The segmentation partitions the motor control commands into non-overlapping segments, where each segment corresponds to a different letter.

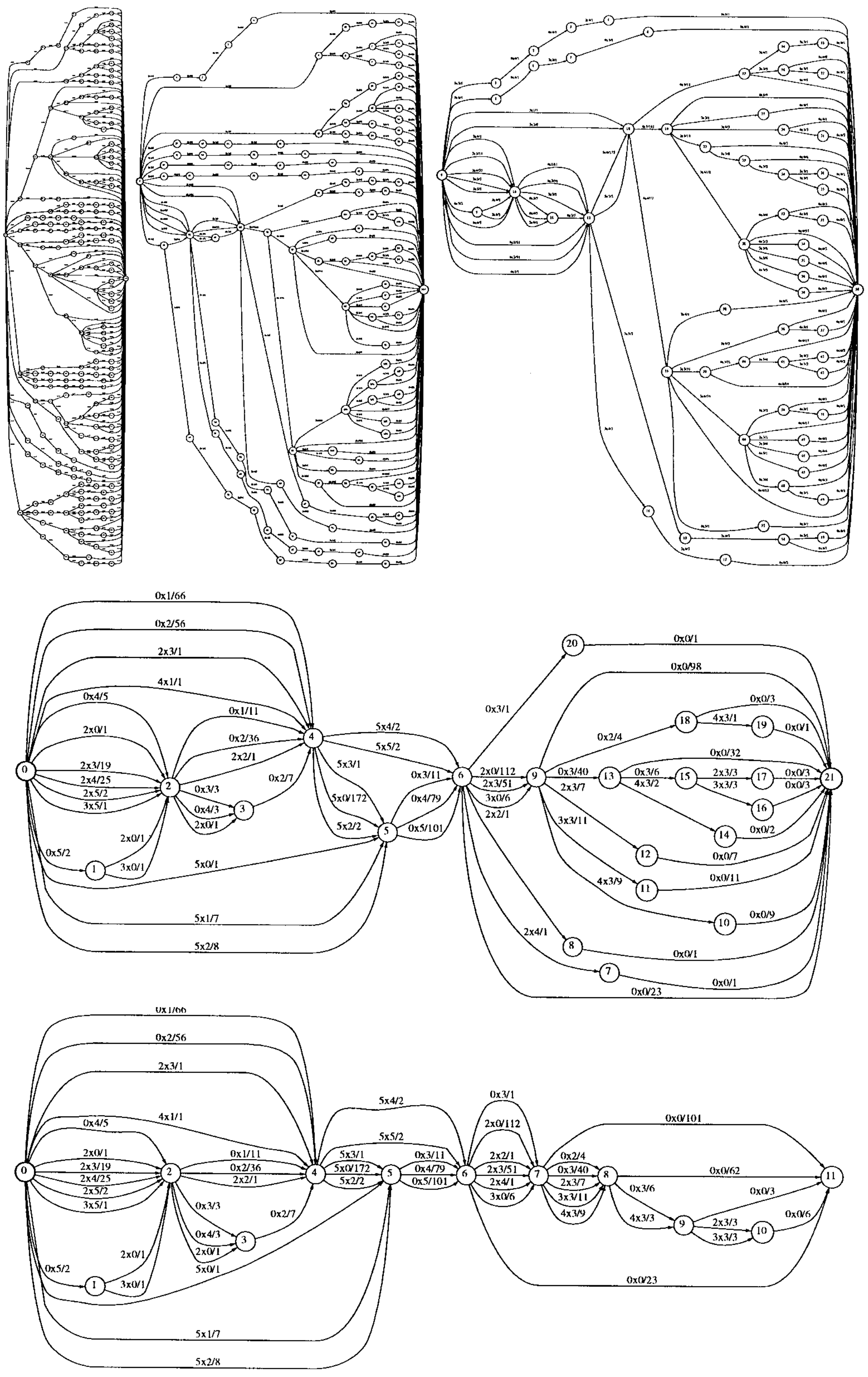


FIG. 5. Several of the intermediate graphs built along the run of the APFA learning algorithm.

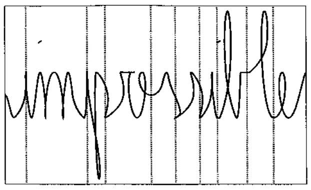


FIG. 6. *Temporal* segmentation of the word `impossible`. The segmentation is performed by evaluating the probabilities of the APFAs which correspond to the letter constituents of the word. These probabilities are evaluated for each possible subsequence of the motor control commands. The most likely segmentation is then found using dynamic programming.

continue with an on-line setting in which we employ the probabilities assigned by the automata to segment new unsegmented words and "feed" the segmented subsequences back as inputs to the corresponding APFAs.

7.2. Building Pronunciation Models for Spoken Words

In natural speech, a word might be pronounced differently by different speakers. For example, the phoneme `t` in `often` is often omitted, and the phoneme `d` in the word `muddy` can be pronounced in a few different ways. One possible approach to model such pronunciation variations is to construct stochastic models that capture the distributions of the possible pronunciations for words in a given database. The models should reflect not only the alternative pronunciations but also the a priori probability of a given phonetic transcription of the word. This probability depends on the distribution of the different speakers that uttered the words in the training set. Such models can be used as a component in a speech recognition system. The same problem was studied by Stolcke and Omohundro [18]. Here, we briefly discuss how our algorithm for learning APFAs can be used to efficiently build probabilistic pronunciation models for words.

We used the TIMIT (Texas Instruments–MIT) database. This database contains the acoustic waveforms of continuous speech with phone labels from an alphabet of 62 phones that constitute a temporally aligned phonetic transcription to the uttered words. For the purpose of building pronunciation models, the acoustic data were ignored and we partitioned the phonetic labels according to the words that appeared in the data. We then built an APFA for each word in the data set. Examples of the resulting APFAs for the words `have`, `had`, and `often` are shown in Fig. 7. The symbol labeling each edge is one of the possible 62 phones or the final symbol, ζ, represented in the figure by the string `End`. The number on each edge is the count associated with the edge, i.e., the number of times the edge was traversed in the training data. The figure shows that the resulting models indeed capture the different pronunciation styles. For instance, all the possible pronunciations of the word `often` contain the phone `f` and there are paths that

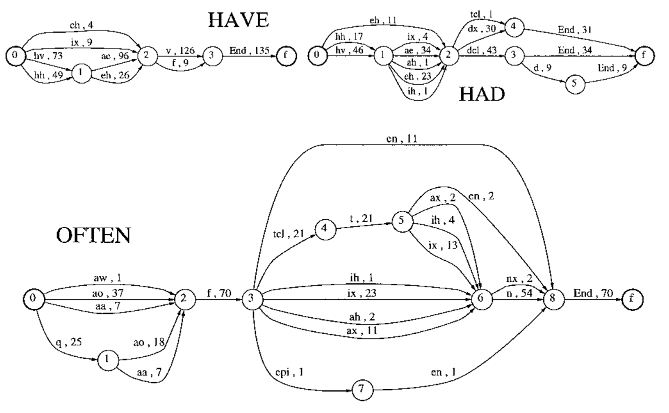


FIG. 7. An example of pronunciation models based on APFAs for the words `have`, `had`, and `often` trained from the TIMIT database.

share the optional `t` (the phones `tcl t`) and paths that omit it. Similar phenomena are captured by the models for the words `have` and `had` (the optional semivowels `hh` and `hv` and the different pronunciations for `d` in `had` and for `v` in `have`).

In order to quantitatively check the performance of the models, we filtered and partitioned the data in the same way as in [18]. That is, words occurring between 20 and 100 times in the data set were used for training and evaluation according to the following partition. Seventy-five percent of the occurrences of each word were used as training data for the learning algorithm and the remaining 25% were used for evaluation. The models were evaluated by calculating the log probability (likelihood) of the respective model on the phonetic transcription for each word in the test set. The results are summarized in Table 1. The performance of the resulting APFAs is surprisingly good, compared to the performance of the hidden Markov model reported by Stolcke and Omohundro [18]. To be cautious, we note that it is not certain whether the better performance (in the sense that the likelihood of the APFAs on the test data is higher) indeed indicates better performance in terms of recognition error rate. Yet, the much smaller time needed for the learning suggests that our algorithm might be the tool of choice for this problem when large amounts of training data are presented.

TABLE 1

The Performance of APFAs compared to Hidden Markov Models (HMM) as Reported in [18] by Stolcke and Omohundro.

Model	*Log-likelihood*	*Perplexity*	*States*	*Transitions*	*Training time*
APFA	−2142.8	1.563	1398	2197	23 sec
HMM [18]	−2343.0	1.849	1204	1542	29: 49 min

Note: *Log-likelihood* is the logarithm of the probability induced by the two classes of models on the test data. *Perplexity* is the average number of phones that can follow in any given context within a word.

8. DIRECTIONS FOR FURTHER RESEARCH

As mention in Section 7, in our implementation of the learning algorithm we allow the algorithm to merge nodes from different levels, and thus the resulting hypothesis is more compact than a leveled APFA. We believe that the learning algorithm remains correct with this modification, but were not able to come up with a proof. Furthermore, we believe that the same idea of merging states which are statistically similar should work for PFAs which have cycles in them (but are μ-distinguishable for non-negligible μ). Allowing cycles, and in particular self-loops, seems to be important in applications for speech analysis.

Another interesting generalization of our result is to give an algorithm which is a good learning algorithm in an agnostic setting. Namely, when the sample is generated according to a source which is not an APFA (or not even a PFA), does there exist a learning algorithm that finds a hypothesis which is close to the APFA which best approximates the target source?

APPENDIX: PROOFS OF TECHNICAL LEMMAS

Proof of Lemma 5.1. In order to prove that the sample has the first property with probability at least $1-\delta/2$, we show that for every $M' \in \mathscr{M}$, and for every state $q' \in Q'$, $m_{q'}/m \geqslant P^{M'}(q') - \varepsilon_0$. In particular, it follows that for every state q' in any given APFA M', if $P^{M'}(q') \geqslant 2\varepsilon_0$, then $m_{q'}/m \geqslant \varepsilon_0$, and thus $m_{q'} \geqslant \varepsilon_0 m \geqslant m_0$. For a given $M' \in \mathscr{M}$, and a state $q' \in Q'$, if $P^{M'}(q') \leqslant \varepsilon_0$, then necessarily $m_{q'}/m \geqslant P^{M'}(q') - \varepsilon_0$. There are at most $1/\varepsilon_0$ states in each of the D levels for which $P^{M'}(q') \geqslant \varepsilon_0$, and hence, using Hoeffding's inequality [7] and the fact that $m \geqslant 1/(2\varepsilon_0^2)\ \ln((2D/(\varepsilon_0\delta)) \cdot 2^{|\Sigma|\, n^2 D})$, with probability at least $1-(\delta/2)\, 2^{-(|\Sigma|\, n^2 D)}$, for each such q', $m_{q'}/m \geqslant P^{M'}(q') - \varepsilon_0$. Since the size of $\mathscr{M}$ is bounded by $2^{|\Sigma|\, n^2 D}$, the above holds with probability at least $1-\delta/2$ for every M'.

And now for the second property, we first observe that by definition of m_0,

$$m_0 = \frac{|\Sigma|\, n^2 D + 2D \ln(8(|\Sigma|+1)) + \ln(1/\delta)}{\varepsilon_1^2} \tag{3}$$

$$> \frac{1}{\varepsilon_1^2} \ln \frac{8(|\Sigma|+1)^{2D}\, 2^{|\Sigma|\, n^2 D}}{\delta}. \tag{4}$$

We next observe that for any automaton M' and any state q' in M', the number of strings that can be generated starting from q' is less than $2(|\Sigma|+1)^D$. Therefore, for a given M' and a given q', if $m_{q'} \geqslant m_0$, then using Hoeffding's inequality with probability larger than

$$1 - \frac{\delta}{4(|\Sigma|+1)^D\, 2^{|\Sigma|\, n^2 D}},$$

$|m_{q',s}/m_{q'} - P^{M'}_{q'}(s)| \leqslant \varepsilon_1$, for every s. Since there are at most $2(|\Sigma|+1)^D$ states in M' (a general bound on the size of the full tree of degree $|\Sigma+1|$), and using our bound on $|\mathscr{M}|$, the second property holds with probability at least $1-\delta/2$, as well. ∎

Proof of Lemma 5.2. We prove the claim by induction on d. M' is constructed in the course of the induction, where for each d we choose the dominant copies among the major copies of the states in Q_d, and define the transition function, τ', accordingly.

For $d=1$, G_{i_1} is G_0 (which is the sample tree, T_S). Based on the definition of $\mathscr{M}$, for every $M' \in \mathscr{M}$, for every $q \in Q_1$, and for every σ such that $\tau(q_0, \sigma) = q$, there exists a copy of q, (q, q_0, σ) in Q'_1. Thus, for every v in the first level of G_0, there is a single edge labeled by some symbol σ entering v, and we let $\Phi_1(v) = (q, q_0, \sigma)$, where $q = \tau(q_0, \sigma)$. Clearly, no two vertices are mapped to the same state in M'. Since all states in Q'_1 are major copies by definition, we can choose the dominant copies of each state $q \in Q_1$ to be all copies q' for which there exists a node v such that $\Phi_1(v) = q'$, and $m_v (= m_{\Phi_1(v)}) \geqslant m_0$.

Assume the claim is true for d' such that $1 \leqslant d' < d$; we prove it for d. Though M' is only partially defined, we allow ourselves to use the notation $W(q')$ for states q' which belong to the levels of M' that have already been constructed. Let $q \in Q_{d-1}$, let $\{q'_i\} \subset Q'_{d-1}$ be its copies, and for each i such that $\Phi^{-1}_{d-1}(q'_i)$ is defined, let $u_i = \Phi^{-1}_{d-1}(q'_i)$. Based on the *goodness* of the sample and our requirement on ε_1, for each u_i such that $m_{u_i} \geqslant m_0$, and for every string s, the difference between $P^{M'}_{q'_i}(s) (= P^M_q(s))$ and $m_{u_i}(s)/m_{u_i}$ is less than $\mu/4$. Hence, if a pair of nodes, u_i and u_j, mapped to q'_i and q'_j, respectively, are tested for similarity by the algorithm, then the procedure **Similar** returns *similar*, and they are folded into one node v. It follows that for every s,

$$m_v(s)/m_v = \frac{m_{u_i}(s) + m_{u_j}(s)}{m_{u_i} + m_{u_j}} \tag{5}$$

$$\leqslant \frac{(P^M_q(s) + \mu/4) \cdot m_{u_i} + (P^M_q(s) + \mu/4) \cdot m_{u_j}}{m_{u_i} + m_{u_j}} \tag{6}$$

$$= P^M_q(s) + \mu/4. \tag{7}$$

Similarly, $m_v(s)/m_v \geqslant P^M_q(s) - \mu/4$. Therefore,

$$|m_v(s)/m_v - P^M_q(s)| < \mu/4, \tag{8}$$

and the same is true for any possible node that is the result of folding some subset of the u_i's that satisfy $m_{u_i} \geqslant m_0$. Since the target automaton is μ-distinguishable, none of these nodes is folded with any node w such that $\Phi_{d-1}(w) \notin \{q'_i\}$. Note that by the induction hypothesis, for every u_i such that $m_{q'_i} = m_{u_i} \geqslant m_0$, q'_i is a dominant copy of q.

Let v be a node in the dth level of G_{i_d}. We first consider the case where v is a result of folding nodes in level $d-1$ of $G_{i_{d-1}}$. Let these nodes be $\{u_1, ..., u_\ell\}$. Since these nodes were folded, necessarily, $m_{u_j} \geqslant m_0$ for every $j \in \{1, ..., \ell\}$. By the induction hypothesis these are mapped to states in Q'_{d-1} which are all dominant major copies of some state $r \in Q_{d-1}$. Note that by the definition of the folding operation, at any point prior to the folding of v itself with other nodes in its level, there is only a single edge (labeled by a single symbol) entering v and this situation can change only when we fold v with another node u. This is the case since whenever we fold two nodes, they are roots of trees which are subgraphs of the current graph constructed by the algorithm, and after their folding the new merged node is still a root of a tree (however, the merged node now might have several incoming edges labeled by different symbols). Thus, let σ be the label of the edge entering v. Then

$$W(v) = \bigcup_{j=1}^{\ell} W(u_j) \circ \sigma \tag{9}$$

$$= \bigcup_{j=1}^{\ell} W(\Phi_{d-1}(u_j)) \circ \sigma \tag{10}$$

$$= W((q, r, \sigma)), \tag{11}$$

where $q = \tau(r, \sigma)$. We thus set $\Phi_d(v) = q'$, where $q' = (q, r, \sigma)$ is a major copy of q in Q'_d. If $m_v \geqslant m_0$, we choose q' to be a dominant copy of q. Otherwise, it is non-dominant. In either case, in accordance with the definition of automata in $\mathscr{M}$, we have for every u_j that $\tau'(\Phi_{d-1}(u_j), \sigma) = q'$ ($= (q, r, \sigma)$). If v is not a result of any such merging in the previous level, then let $u \in G_{i_d}$ be such that $u \xrightarrow{\sigma} v$. We have,

$$W(v) = W(u) \circ \sigma = W(\Phi_{d-1}(u)) \circ \sigma. \tag{12}$$

If $m_u \geqslant m_0$, then $\Phi_{d-1}(u)$ is a (single) dominant major copy of some state $r \in Q_{d-1}$. This case is very similar to the previous case (in which there are several dominant major copies of r), and we let $\Phi_d(v) = (q, r, \sigma)$ be a major copy of q, where $q = \tau(r, \sigma)$. If $m_v \geqslant m_0$, we choose (q, r, σ) to be a dominant copy of q (and otherwise, it is non-dominant). In any case, we set $\tau(\Phi_{d-1}(u), \sigma) = (q, r, \sigma)$. Since $\Phi_{d-1}(u)$ is a single dominant copy of r, then $W(\Phi_d(v)) = W(\Phi_{d-1})) \circ \sigma$, which by Eq. (12) equals $W(v)$. If $m_u < m_0$, then $\Phi_{d-1}(u)$ is a non-dominant copy of some r in Q_{d-1}. In such a case we set $\Phi_d(v) = (q, \Phi_{d-1}(u), \sigma)$ (where as before $q = \tau(r, \sigma)$). By definition, $\Phi_d(v)$ is a minor copy, but since the only edge entering v goes out of u, $m_v \leqslant m_u < m_0$, which is consistent with the lemma statement. Finally, since there is only a single edge (which is labeled σ) entering $\Phi_d(v)$ (going out of $\Phi_{d-1}(u)$), we have that $W(\Phi_d(v)) = W\Phi_{d-1}(u)) \circ \sigma$, which by Eq. (12) equals $W(v)$. ∎

Proof of Lemma 5.3. In order to prove Claim (1) we show (using an inductive argument) that the total weight in M' of the dominant copies of every state $q \in Q_d$ for which $P^M(q) \geqslant \varepsilon_2$ is at least $1 - d\varepsilon_3$ of the weight of q.

For $d = 1$: The number of copies of each state in Q_1 is at most $|\Sigma|$. By the goodness of the sample, for each copy q' whose weight is greater than $2\varepsilon_0$, $m_{q'} \geqslant m_0$, and thus by Lemma 5.2, q' is dominant. Hence the total weight of the dominant copies is at least $\varepsilon_2 - 2|\Sigma|\,\varepsilon_0$. Based on our choice of ε_2 and ε_3, the total weight is therefore at least $(1-\varepsilon_3)\,\varepsilon_2$.

For $d > 1$: By the induction hypothesis, the total weight of the dominant major copies of a state r in Q_{d-1} is at least $(1-(d-1)\,\varepsilon_3)\,P^M(r)$. For $q \in Q_d$, the total weight of the major copies of q is thus at least

$$\sum_{r,\,\sigma:\, r \xrightarrow{\sigma} q} (1-(d-1)\,\varepsilon_3)\,P^M(r) \cdot \gamma(r, \sigma)$$

$$= (1-(d-1)\,\varepsilon_3)\,P^M(q). \tag{13}$$

There are at most $n\,|\Sigma|$ major copies of q, and hence the total weight of the non-dominant ones is at most $2n\,|\Sigma|\,\varepsilon_0 < \varepsilon_3\varepsilon_2$ and the claim follows.

We next prove Claim (2). We break the analysis into two cases. If $\gamma(q, \sigma) \leqslant \gamma_{min} + \varepsilon_1$, then since $\hat{\gamma}(\hat{q}, \sigma) \geqslant \gamma_{min}$ by definition, and $\varepsilon_1 \leqslant \varepsilon^2/(8(|\Sigma|+1))$, if we choose $\gamma_{min} = \varepsilon/(4(|\Sigma|+1))$, then $\gamma(q, \sigma)/(\hat{\gamma}(\hat{q}, \sigma) \leqslant 1 + \varepsilon/2$, as required.

If $\gamma(q, \sigma) > \gamma_{min} + \varepsilon_1$, then let $x \overset{\text{def}}{=} \gamma(q, \sigma) - \gamma_{min} - \varepsilon_1 > 0$. Based on our choice of ε_2 and ε_3, for every $d \leqslant D$, $\varepsilon_2(1 - d\varepsilon_3) \geqslant 2\varepsilon_0$. By the goodness of the sample, and the definition of $\hat{\gamma}(\cdot, \cdot)$, we have that

$$\hat{\gamma}(\hat{q}, \sigma) \geqslant (\gamma(q, \sigma) - \delta_1)(1 - (|\Sigma|+1)\,\gamma_{min}) + \gamma_{min} \tag{14}$$

$$= (x + \gamma_{min})(1 - \varepsilon/4) + \gamma_{min} \tag{15}$$

$$\geqslant \frac{x + \gamma_{min}(1 + \varepsilon/2)}{1 + \varepsilon/2} \geqslant \frac{\gamma(q, \sigma)}{1 + \varepsilon/2}. \tag{16}$$

∎

ACKNOWLEDGMENTS

We thank an anonymous COLT'95 committee member for her/his careful reading and very helpful comments. Special thanks to Andreas Stolcke for helpful comments and for pointing us to Ref. [4]. We also thank Ilan Kremer, Yoav Freund, Mike Kearns, Ronitt Rubinfeld, and Rob Schapire for helpful discussions. This research has been supported in part by the Israeli Ministry of Sciences and Arts and by the Bruno Goldberg endowment fund. Dana Ron thanks the Eshkol fellowship and the National Science Foundation for their support. Yoram Singer acknowledges the Clore Foundation for its support. Most of this research was done while Dana Ron and Yoram Singer were at the Hebrew University.

REFERENCES

1. N. Abe and M. Warmuth, On the computational complexity of approximating distributions by probabilistic automata, *Mach. Learning* **9** (1992), 205–260.
2. L. E. Baum and T. Petrie, Statistical inference for probabilistic functions of finite state Markov chains, *Ann. Math. Statist.* **37** (1966).
3. A. Blum, M. Furst, M. J. Kearns, and R. J. Lipton, Cryptographic primitives based on hard learning problems, *in* "Pre-Proceedings of CRYPTO '93," pp. 24.1–24.10, 1993.
4. R. C. Carrasco and J. Oncina, Learning stochastic regular grammars by means of a state merging method, *in* "The 2nd International Colloquium on Grammatical Inference and Applications," pp. 139–152, 1994.
5. F. R. Chen, Identification of contextual factors for pronounciation networks, *in* "Proceedings of IEEE Conference on Acoustics, Speech and Signal Processing," pp. 753–756, 1990.
6. Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. E. Schapire, and L. Sellie, Efficient learning of typical finite automata from walks, *in* "Proceedings of the 24th Annual ACM Symposium on Theory of Computing," pp. 315–324, 1993.
7. W. Hoeffding, Probability inequalities for sums of bounded random variables, *Amer. Statist. Assoc. J.* **58** (1963), 13–30.
8. M. Kearns, Y. Mansour, D. Ron, R. Rubinfeld, R. E. Schapire, and L. Sellie, On the learnability of discrete distributions, *in* "The 25th Annual ACM Symposium on Theory of Computing," 1994.
9. M. J. Kearns, Efficient noise-tolerant learning from statistical queries, *in* "Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing," pp. 392–401, 1993.
10. R. Plamondon, C. Y. Suen, and M. L. Simner, Eds., "Computer Recognition and Human Production of Handwriting," World Scientific, Singapore, 1989.
11. L. R. Rabiner, A tutorial on hidden Markov models and selected applications in speech recognition, *Proc. IEEE* (1989).
12. L. R. Rabiner and B. H. Juang, An introduction to hidden Markov models, *IEEE ASSP Mag.* **3** (1986), 4–16.
13. M. D. Riley, A statistical model for generating pronounciation networks, *in* "Proceedings of IEEE Conference on Acoustics, Speech and Signal Processing," pp. 737–740, 1991.
14. D. Ron, Y. Singer, and N. Tishby, Learning probabilistic automata with variable memory length, *in* "Proceedings of the Seventh Annual Workshop on Computational Learning Theory," 1994; *Mach. Learning*, to appear.
15. D. Sankoff and J. B. Krusbal, "Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison," Addison–Wesley, Reading, MA, 1983.
16. Y. Singer, "What Has Been Will Be Again: Machine Learning Approach to the Analysis of Natural Language," Ph.D. thesis, The Hebrew University of Jerusalem, 1995.
17. Y. Singer and N. Tishby, Dynamical encoding of cursive handwriting, *Biol. Cybern.* **71** (1994), 227–237.
18. A. Stolcke and S. Omohundro, Hidden Markov model induction by Bayesian model merging, *in* "Advances in Neural Information Processing Systems," Vol. 5, Morgan Kaufmann, San Mateo, CA, 1992.
19. C. C. Tappert, C. Y. Suen, and T. Wakahara, The state of art in on-line handwriting recognition, *IEEE Trans. Pattern Annal. Mach. Int.* **12** (1990), 787–808.
20. B. A. Trakhtenbrot and Ya. M. Brazdin', "Finite Automata: Behavior and Synthesis," North-Holland, Amsterdam, 1973.