On the Learnability and Usage of Acyclic Probabilistic Finite Automata

(Working Draft)

Dana Ron
MIT Laboratory of Computer Science
Cambridge, MA 02139
danar@theory.lcs.mit.edu

Yoram Singer
AT&T Bell Laboratories
Murray Hill, NJ 07974
singer@research.att.com

Naftali Tishby
Institute of Computer Science and
Center for Neural Computation
Hebrew University, Jerusalem 91904, Israel
tishby@cs.huji.ac.il

Abstract

We propose and analyze a distribution learning algorithm for a subclass of *Acyclic Probabilistic 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 indeed 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 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 incredibly small amount of time needed for learning suggests that the learning algorithm of APFAs might be a powerful alternative to commonly used probabilistic models.

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* (*APFA*s) for modeling distributions on short sequences such as those mentioned above. These automata seem to capture well the context dependent variability of such sequences. 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 online version of our algorithm which is also given in this paper.

In a previous work [15] we introduced an algorithm for learning distributions (on long strings) generated by ergodic Markovian sources that can be described by a different subclass of probabilistic finite automata (PFAs) which we refer to as *Variable Memory* PFAs. Our two learning algorithm 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 [18].

The algorithm described in this paper is an efficient algorithm for learning distributions on strings generated by a subclass of APFAs which have the following property. For every pair of states in an automaton M belonging to this class, 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 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 described by Kearns *et. al.* [8]. They show that PFAs are not efficiently learnable under the widely acceptable assumption that there is no efficient algorithm for learning parity functions in the presence of noise in the PAC model. 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' [21] 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[1]. 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

[1] 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.

we maintain a sequence of directed edge-labeled acyclic graphs. The first graph in this sequence, named the *sample tree*, is constructed based on the a 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).

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 [16]) on the one hand, and Hidden Markov Models (in particular *left-to-right* HMMs) on the other hand [12, 13]. 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 main weakness of this approach is that it does not treat any context dependent, or non-local variations, without making the noise model much more complex. This property is unrealistic for many of the above applications due to phenomena such as *coarticulation* in speech and handwriting, or long range chemical interactions (due to geometric effects) in biochemistry.

HMMs (which PFAs are a special case of) are popular in speech recognition and have better ability than the string matching based models to capture context dependent variations, However, 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 online 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 [4], and by Stolcke and Omohundro [19]. Carrasco and Oncina give an algorithm which identifies *in the limit* distributions generated by PFAs. 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 reviews on practical models and algorithms for multiple-pronunciation can be found in [5, 14], and for cursive handwriting recognition in [11, 10, 20, 3].

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 online version of our algorithm, and in Section 7 we describe two applications of the algorithm. We conclude with several suggestion for future research in Section 8.

2 Preliminaries

A *Probabilistic Finite Automaton*[2] (*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 in an alphabet Σ, where for every state, each outgoing edge must be 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 must sum up to one. We require that all edges going into the final state (and those edges only), be labeled by a special final symbol ζ. A PFA generates 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*;
- $\tau : Q \times \Sigma \bigcup \{\zeta\} \to Q \bigcup \{q_f\}$ is the *transition function*;
- $\gamma : Q \times \Sigma \bigcup \{\zeta\} \to [0, 1]$ is the *next symbol probability function*.

The function γ must satisfy 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 $Q \times \Sigma^*\{\mathsf{e} \cup \zeta\}$ (where e denoted the empty string) in the following recursive manner: $\tau(q, s_1 s_2 \ldots s_l) = \tau(\tau(q, s_1 \ldots s_{l-1}), s_l)$, and $\tau(q, \mathsf{e}) = q$.

A PFA M generates 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 $s = s_1 \ldots s_{l-1} s_l$, where $s_l = \zeta$, denoted by $P^M(s)$ is

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

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

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

For a string $s = s_1 \ldots s_l$ where $s_l \neq \zeta$ 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, $P^M(s) = \prod_{i=0}^{l-1} \gamma(q_i, s_{i+1})$.

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

Given a state q in Q, and a string $s = s_1 \dots s_l$ (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,

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

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 the distance in the L_∞ norm between the distributions $P_{q_1}^M$ and $P_{q_2}^M$ is at least μ. Namely, q_1 and q_2 are μ-distinguishable if there exists a string s for which $\left|P_{q_1}^M(s) - P_{q_2}^M(s)\right| \geq \mu$. We say that a PFA M is* **μ-distinguishable**, *if every pair of states in M are* **μ-distinguishable**.[3]

We shall 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 (*APFA*) 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 must be to states in level $d+1$, except for transitions labeled by the final symbol, ζ, which need not be restricted in this way. 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 not much larger leveled APFA.

Lemma 2.1 *For every APFA M having n states and depth D, there exists an equivalent leveled APFA, $\overline{M}$, with at most $n(D-1)$ states.*

Proof: We define $\overline{M} = (\overline{Q}, \Sigma, \zeta, \overline{\tau}, \overline{\gamma}, \overline{q_0}, \overline{q_f})$ 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 $\overline{M}$. 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 $\overline{q_f}$ of q_f in level D of $\overline{M}$, and an edge labeled by ζ from every copy of q to $\overline{q_f}$.

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 $\overline{q}_d \in \overline{Q}_d$. For $q = q_0$, $(\overline{q_0})_0$ is simply the starting state of $\overline{M}$, $\overline{q_0}$, and q_f has a single copy, $\overline{q_f} \in \overline{Q}_D$ (which is the final state of $\overline{M}$. For every level d and for every $\sigma \in \Sigma \bigcup \{\zeta\}$, $\overline{\gamma}(\overline{q}_d, \sigma) = \gamma(q, \sigma)$. For $\sigma \in \Sigma$, $\overline{\tau}(\overline{q}_d, \sigma) = \overline{q'}_{d+1}$, where $q' = \tau(q, \sigma)$, and $\overline{\tau}(\overline{q}_d, \zeta) = \overline{q_f}$. Every state is copied at most $D-1$ times, therefore the total number of states in $\overline{M}$ 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 in [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 $\widehat{M}$ be a hypothesis APFA. Let P^M and $P^{\widehat{M}}$ be the two probability distributions they generate respectively. We say that $\widehat{M}$ is an* **ϵ-good** *hypothesis with respect to M, for $\epsilon \geq 0$, if*

$$\mathcal{D}_{KL}[P^M || P^{\widehat{M}}] \leq \epsilon \quad ,$$

[3] As noted 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.

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

$$\mathcal{D}_{KL}[P^M||P^{\widehat{M}}] \stackrel{\text{def}}{=} \sum_{s \,\in\, \Sigma^*\zeta} P^M(s) \log \frac{P^M(s)}{P^{\widehat{M}}(s)} \quad .$$

Our learning algorithm for APFAs is given a *confidence* parameter $\delta > 0$, and an *approximation* parameter $\epsilon > 0$. 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.[4] 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 runs in time polynomial in $\frac{1}{\epsilon}$, $\log \frac{1}{\delta}$, $|\Sigma|$, n, and $\frac{1}{\mu}$.

4 The Learning Algorithm

In this section we describe our algorithm for learning APFAs. An *online* 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 some string in the sample. With each edge we also associate 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 is proven using Chernoff bounds. However, we cannot use the counts in the sample tree to reliably predict the probability that M generates longer strings, which 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 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. What we need to do is to use the information in the sample tree in a slightly more sophisticated way.

Given 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 taking the walk corresponding to the path from the root of the sample tree to v, starting from q_0. This mapping is clearly a many to one mapping. Assume we were given this mapping. Then, for each state in M we could merge all nodes which are mapped to that state to a single node. For every symbol σ we 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

[4]These last two assumption can be removed by *searching* for n and μ. This search is performed by testing the hypotheses the algorithm outputs when it runs with growing values of n, and decreasing values of μ. Such a test can be done by comparing the log-likelihood of the hypotheses on additional test data.

edges. Using the resulting acyclic graph and the counts associated with its edges, we could define a hypothesis APFA for which it is not hard to show is a good hypothesis with high probability (for a large enough sample size). Since such a mapping is not given to the algorithm we try and infer this mapping for as many nodes as possible, using the fact that M is μ-distinguishable.

Starting from the first level in the sample tree, we test if pairs of nodes which correspond to a large enough number of prefixes of sample strings, seem to map to the same state. We do so 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 ideas is that if two nodes correspond to different states then, since the states are μ-distinguishable, we should see evidence to this difference in the sample. If we decide that two nodes are mapped to the same state then we merge them and the corresponding nodes in their subtrees. Doing so we enhance the reliability of the test for pairs of nodes in deeper levels which had low counts (or even no counts) associated with them prior to any mergings. In our analysis we show that with high probability, in this process, we do not merge pairs of nodes that are mapped to different states, while we do merge most pairs of nodes that do correspond to the same state, resulting in a reliable 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 we maintain a series of directed leveled acyclic graphs $G_0, G_1, \ldots, G_{N+1}$, 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, $G_0, \ldots, G_{N+1}$, 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 $\sigma \in \Sigma \bigcup \{\zeta\}$. 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 $v \stackrel{\sigma}{\rightarrow} u$. If there is a labeled (directed) path from v to u corresponding to a string s, then we denote it similarly by $v \stackrel{s}{\Rightarrow} u$. 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.,

$$S(v) \stackrel{\rm def}{=} \{s : s = s's'' \in S, \; v_0 \stackrel{s'}{\Rightarrow} v\}_{multi} \; .$$

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,

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

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 $v_0 \stackrel{s'}{\Rightarrow} v$.

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, and it always holds by construction that $m_v = |S(v)|$ $(= |S_{gen}(v)|)$, and $m_v(\sigma)$ equals the number of strings in $S_{gen}(v)$ whose first symbol is σ.

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, 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 $i = 0, \ldots, N-1$, we associate with G_i a level, $d(i)$, where $d(0) = 1$, and $d(i) \geq d(i-1)$. This is the level in G_i we plan to operate 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 we choose a pair of nodes u and v, both belonging to $d(i)$, which have the following properties: for a predefined threshold m_0 (that is set in the analysis of the algorithm) both $m_u \geq m_0$ and $m_v \geq m_0$, and the nodes are *similar* in a sense defined subsequently. We then merge 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, we let $m_w(\sigma) = m_u(\sigma) + m_v(\sigma)$. The virtual multiset of strings corresponding to w, $S(w)$, is simply the union of $S(u)$ with $S(v)$. An illustration of the folding operation is depicted in Figure 1. Note that since the algorithm proceeds level by level, for every G_i, the nodes in 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.

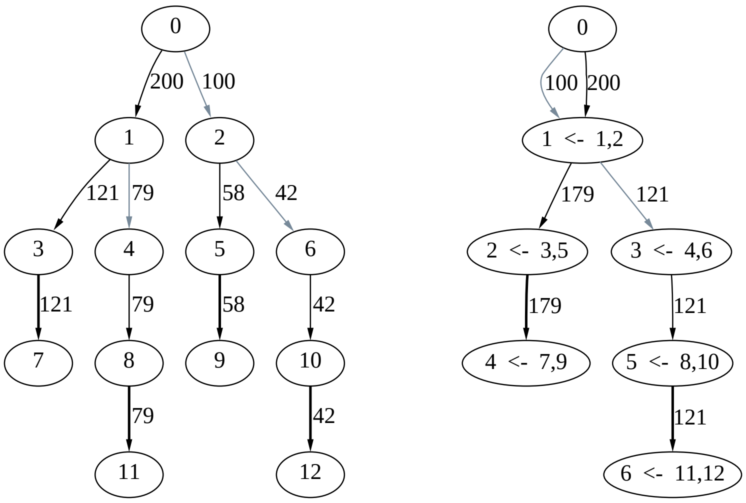

Figure 1: An illustration of the folding operation. The graph on the right is constructed from the graph on the left by merging the nodes 1 and 2. The different edges represent different output symbols: gray is `0`, black is `1` and bold black is ζ.

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

Finally, we define our 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 $\sigma \in \Sigma \bigcup \{\zeta\}$, we define

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

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, is similar. More formally, for a given node v and a string s, let $m_v(s) \stackrel{\text{def}}{=} |\{t : t \in S_{gen}(v),\ t = st'\}_{multi}|$, be the number of strings viewed as generated from v which s is a prefix of. We say that a given pair of nodes u and v, are *similar* if for *every* string s,

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

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 **Similiar** presented in the end of this section.

For sake of simplicity of the pseudo-code that follows, we associate with each node in a graph G_i, a number in $\{1, \ldots, |G_i|\}$. 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. 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 highest level of the recursion, **Similar** is always called by **Learn-Acyclic-PFA** for some pair of nodes j, j' which we are interested in comparing, where p_j and $p_{j'}$ are both set to one. In general, 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 always correspond to the same path in the two respective subtrees of j and j', p_u and p_v are found to be very different, then the function returns *non-similar*, and this value propagates up to the highest level of the recursion. Otherwise, the functions is called with pairs of respective children of u and v. If some child is missing, then we set the node to be *undefined* (where its corresponding probability, by definition, is 0).

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 σ from j to *small*(d) ;
   (b) If $m_j(\zeta) = 0$ then add an edge labeled σ 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 $\sigma \in \Sigma \bigcup \{\zeta\}$:

$$\hat{\gamma}(\hat{q}, \sigma) := (m_v(\sigma)/m_v)(1 - (|\Sigma| + 1)\gamma_{min}) + \gamma_{min} \quad ,$$

   where v is the node corresponding to $\hat{q}$ in G.

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 security parameter $\delta > 0$, and approximation parameter $\epsilon > 0$, 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 $\frac{1}{\epsilon}$, $\log\frac{1}{\delta}$, $\frac{1}{\mu}$, n, D, and $|\Sigma|$.*

We would like to note that for a given approximation parameter ϵ, we may slightly weaken the requirement that M be μ-distinguishable. It suffices that we require that every pair of states q_1 and q_2 in M such that both the probability of reaching q_1 starting from q_0, and the probability of reaching q_2 starting from q_0, and are greater than some ϵ' (which is a function of ϵ, μ and n), q_1 and q_2 be μ-distinguishable. For sake of simplicity, we give our analysis under the slightly stronger assumption.

Without loss of generality, (based on Lemma 2.1) we may assume that M is a leveled APFA with at most n state in each of its D levels. We add the following notations.

- For a state $q \in Q_d$,
  - $W(q)$ denotes the set of *all* strings in Σ^d which *reach* q; $P^M(q) \stackrel{\text{def}}{=} \sum_{s\in W(q)} P^M(s)$.
  - 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,

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

- For a state $\hat{q} \in \hat{Q}_d$, $W(\hat{q})$ $m_{\hat{q}}$, $m_{\hat{q}}(s)$, and $P^{\widehat{M}}(\hat{q})$ 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 $W(v) \subseteq W(q)$.

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. We prove that with high probability a sample generated by the target APFA 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.

5.1 A *Good* Sample

In order to define when a sample is good in the sense that it has the statistical properties required to ensure the successful completion of our algorithm, we introduce a class of APFAs $\mathcal{M}$, which is defined below. The reason for introducing this class is roughly the following. The heart of our algorithm is in the folding operation, and the similarity test that precedes it. We want to show that, on one hand, we do not fold pairs of nodes which correspond to two different states, and on the other hand, we fold most pairs of nodes that do correspond to the same state. By *most* we essentially mean that in our final hypothesis, the weight of the *small* states (which correspond to the unfolded nodes whose counts are small and thus cannot be used to reliably compute the probability of strings that pass through them) is in fact small.

Whenever we perform the *similarity* test between two nodes u and v, we compare 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 sets are of substantial size, then each will be in some sense *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 sets will not deviate much from the probability it was generated 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 an only if they correspond to the same state.

For a given level d, let G_{i_d} be the first graph in which we start folding nodes in level d. Consider some specific state q in level d of the target automaton. Let $S(q) \subseteq S$ be the subset of sample strings which pass through q. Let $v_1, \ldots, v_k$ be the nodes in G which correspond to q, in the sense that each string in $S(q)$ passes through one of the v_i's. Hence, these nodes induce a partition of $S(q)$ into the sets $S(v_1), \ldots, S(v_k)$. It is clear that if $S(q)$ is large enough, then, since the strings were generated independently, we can apply Chernoff bounds to get that with high probability $S(q)$ is typical to q. But we want to know that *each* of the $S(v_i)$'s is typical to q. It is clearly not true that *every* partition of $S(q)$ preserves the statistical properties of q. However, the graphs constructed by the algorithm do not induce arbitrary partitions, and we are able to characterize the possible partitions in terms of the automata in $\mathcal{M}$. This characterization also helps us bound the weight of the small states in our hypothesis.

Given a target APFA M, let $\mathcal{M}$ be the set of APFAs $\{M' = (Q', q'_0, \{q'_f\}, \Sigma, \tau', \gamma', \zeta)\}$ 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 $\sigma \in \Sigma \bigcup \{\zeta\}$,

   (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, i.e., $\forall s \in \Sigma^* \zeta$, $P^{M'}(s) = P^M(s)$.

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

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). 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 major copy q' of q and for every $\sigma \in \Sigma \bigcup \{\zeta\}$, if $\tau(q, \sigma) = t$, then $\tau'(q', \sigma) = (t, q, \sigma)$. Thus, for each symbol σ, *all* σ transitions from the dominant major 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 it's descendants are (non-dominant) minor copies.

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

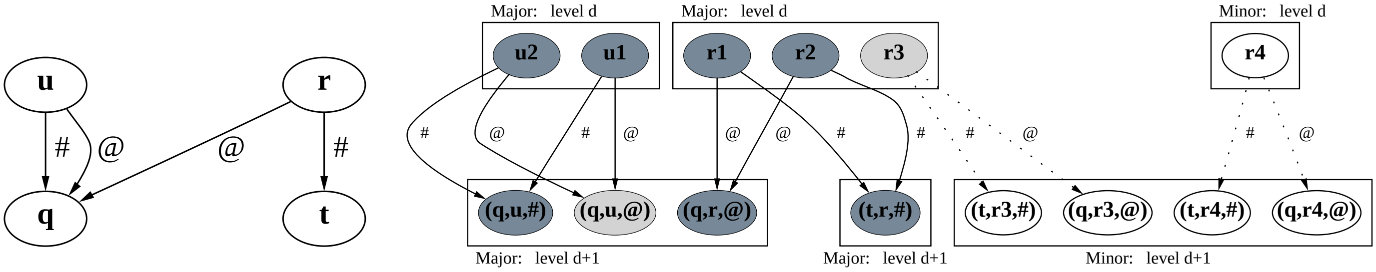

Figure 2: Left: Part of the original automaton, M, that corresponds to the copies on the right part of the figure. Right: The different types of copies of M's states: u has only major copies, u_1 and u_2, who are both dominant and have each an edge labeled by # going to the major copy of q, $(q, u, \#)$,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

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 5.3, 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* $(\boldsymbol{\epsilon_0, \epsilon_1})$-good *with respect to M if for every $M' \in \mathcal{M}$ and for every state $q' \in Q'$:*

1. *If $P^{M'}(q') \geq 2\epsilon_0$, then $m_{q'} \geq m_0$, where*

$$m_0 = \frac{|\Sigma|\, n^2\, D^2 \,+\, 2D \ln\left(8(|\Sigma| + 1)\right) \,+\, \ln \frac{1}{\delta}}{\epsilon_1^2} \ ;$$

2. *If $m_{q'} \geq m_0$, then for every string s,*

$$|m_{q',s}/m_{q'} - P_{q'}^{M'}(s)| \leq \epsilon_1 \ ;$$

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

$$m \geq \max\left(\frac{|\Sigma|\, n^2\, D \,+\, \ln \frac{2D}{\epsilon_0 \delta}}{2\epsilon_0^2}, \frac{m_0}{\epsilon_0}\right) \ ,$$

is (ϵ_0, ϵ_1)-good with respect to M.

Proof: In order to prove that the sample has the first property with probability at least $1-\delta/2$, we show that for every $M' \in \mathcal{M}$, and for every state $q' \in M'$, $m_{q'}/m \geq P^{M'}(q') - \epsilon_0$. In particular, if follows that for every state q' in any given APFA M', if $P^{M}(q') \geq 2\epsilon_0$, then $m_{q'}/m \geq \epsilon_0$, and thus $m_{q'} \geq \epsilon_0 m \geq m_0$. For a given $M' \in \mathcal{M}$, and a state $q' \in M'$, if $P^{M'}(q') \leq \epsilon_0$, then necessarily $m_{q'}/m \geq P^{M'}(q') - \epsilon_0$. There are at most $1/\epsilon_0$ states in each level for which $P^{M'}(q') \geq \epsilon_0$, and hence, using Hoeffding's inequality [7] and the fact that $m \geq (1/\epsilon_0^2)\ln\left((2D/(\epsilon_0\delta)) \cdot 2^{|\Sigma|n^2D}\right)$, with probability at least $1-(\delta/2)2^{-(|\Sigma|n^2D)}$, for each such q', $m_{q'}/m \geq P^{M'}(q') - \epsilon_0$. Since the size of $\mathcal{M}$ is bounded by $2^{|\Sigma|n^2D}$, the above holds with probability at least $1-\delta/2$ for every M'.

And now for the second property. Since

$$m_0 = \frac{|\Sigma|\, n^2\, D \,+\, 2D\ln\left(8(|\Sigma|+1)\right) + \ln\frac{1}{\delta}}{\epsilon_1^2} \tag{3}$$

$$> \frac{1}{\epsilon_1^2}\ln\frac{8(|\Sigma+1)^{2D}\, 2^{|\Sigma|n^2D}}{\delta} \quad , \tag{4}$$

for a given M', and a given q', if $m_{q'} \geq m_0$ then using Hoeffding's inequality, and since there are less than $2(|\Sigma|+1)^D$ strings that can be generated starting from q', with probability larger than

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

for every s, $|m_{q',s}/m_{q'} - P^{M'}_{q'}(s)| \leq \epsilon_1$. Since there are at most $2(|\Sigma|+1)^D$ states in M' (a bound on the size of the full tree of degree $|\Sigma+1|$), and using our bound on $|\mathcal{M}|$, we have the second property with probability at least $1-\delta/2$, as well. ■

5.2 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 $\widehat{M}$, which has significant weight, and for which $\hat{\gamma}(\hat{q},\cdot) \approx \gamma(q,\cdot)$.

Lemma 5.2 *If the sample is (ϵ_0,ϵ_1)-good for*

$$\epsilon_1 < \min(\mu/4, \epsilon^2/8(|\Sigma|+1)) \ ,$$

then for any $\epsilon_3 \leq 1/(2D)$, and for any $\epsilon_2 \geq 2n|\Sigma|\epsilon_0/\epsilon_3$, we have the following. For every level d and for every state $q \in Q_d$, if $P^M(q) \geq \epsilon_2$ then there exists a state $\hat{q} \in \hat{Q}_d$ such that:

1. $P^M\left(W(q) \bigcap W(\hat{q})\right) \geq (1-d\epsilon_3)P^M(q)$,
2. *for every symbol σ, $\gamma(q,\sigma)/\hat{\gamma}(\hat{q},\sigma) \leq 1+\epsilon/2$.*

The proof of Lemma 5.2 is derived based on the following lemma in which we show a relationship between the graphs constructed by the algorithm and an APFA in $\mathcal{M}$.

Lemma 5.3 *If the sample is (ϵ_0,ϵ_1)-good, for $\epsilon_1 < \mu/4$, 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 d'th level of G_{i_d},* into *Q_d', such that for every v in the d'th level of G_{i_d}, $W(v) = W(\Phi_d(v))$. Furthermore, $q' \in M'$ is a dominant major copy* iff *$m_{q'} \geq m_0$.*

Proof: 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.

For $d = 1$, G_{i_1} is G_0 (which is the sample tree, T_S). Based on the definition of $\mathcal{M}$, for every $M' \in \mathcal{M}$, 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 $(= m_{\Phi_1(v)}) \geq m_0$.

Assume the claim is true for $1 \leq d' < d$, 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 are already constructed. Let $q \in Q_{d-1}$, let $\{q'_i\} \subset Q'_{d-1}$ be its copies, and for each i such that $\Phi_{d-1}^{-1}(q'_i)$ is defined, let $u_i = \Phi_{d-1}^{-1}(q'_i)$. Based on the *goodness* of the sample and our requirement on ϵ_1, for each u_i such that $m_{u_i} \geq m_0$, and for every string s, the difference between $P^{M'}_{q'_i}(s)$ 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, than the procedure Similar returns *similar*, and they are folded into one node v. For every s, since

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

then

$$|m_v(s)/m_v - P^M_q(s)| < \mu/4 \ , \tag{6}$$

and the same is true for any possible node that is the result of folding some subset of the u_i's that satisfy $m_{u_i} \geq m_0$. Since the target automaton is μ-distinguishable, none of these nodes are 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 $m_{q'_i} = m_{u_i} \geq m_0$, q'_i is a dominant copy of q.

Let v be a node in the d'th 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 $\{u_1, \ldots, u_\ell\}$. By the induction hypothesis they are mapped to states in Q'_{d-1} which are all dominant major copies of some state $r \in Q_{d-1}$. Let σ be the label of the edge entering v. Then

$$\begin{aligned} W(v) &= \bigcup_{j=1}^{\ell} W(u_j) \circ \sigma & (7)\\ &= \bigcup_{j=1}^{\ell} W(\Phi_{d-1}(u_j)) \circ \sigma & (8)\\ &= W((q, r, \sigma)), & (9) \end{aligned}$$

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 $m_v \geq m_0$, we choose q' to be a dominant copy of q. Otherwise, it is non-dominant. If v is not a cause of any such merging in the previous level, then let $u \in G_{i_d}$ be such that $u \xrightarrow{\sigma} v$. Then

$$W(v) \ = \ W(u) \circ \sigma \ = \ W(\Phi_{d-1}(u)) \circ \sigma \ = \ W(\tau'(\Phi_{d-1}(u), \sigma)) \, , \tag{10}$$

and we simply set

$$\Phi_d(v) = \tau'(\Phi_{d-1}(u), \sigma) \ .$$

If $m_u \geq m_0$, then $\Phi_{d-1}(u)$ is a (single) dominant major copy of some state $r \in Q_{d-1}$, and $q' = \Phi_d(v)$ is a major copy. In this case, if $m_v \geq m_0$, we choose q' to be a dominant copy of q. If $m_u < m_0$, then $\Phi_{d-1}(u)$ is a non-dominant copy of some r in Q_{d-1}. $\Phi_d(v)$ is a minor copy by definition, but since the only edge entering v goes out of u, $m_v \leq m_u < m_0$, which is consistent with the lemma statement. ■

Proof of Lemma 5.2 : For both claims we rely on the relation that is shown in Lemma 5.3, between the graphs constructed by the algorithm and some APFA M' in $\mathcal{M}$. We show that the weight in M' of the dominant copies of every state $q \in Q_d$ for which $P^M(q) \geq \epsilon_2$ is at least $1 - d\epsilon_3$ of the weight of q. Claim (1) directly follows, and for proving Claim (2) we apply the goodness of the sample as explained in more detail subsequently. Our proof regarding the weight of the dominant copies the the states in Q is by induction on d.

For $d = 1$: The number of copies of each state in Q_d^1 is at most $|\Sigma|$. By the goodness of the sample, for each copy q' whose weight is greater than $2\epsilon_0$, $m_{q'} \geq m_0$, and thus by Lemma 5.3, q' is dominant. Hence the total weight of the dominant copies is at least $\epsilon_2 - 2|\Sigma|\epsilon_0$ which based on our choice of ϵ_2 and ϵ_3, is at least $(1 - \epsilon_3)\epsilon_2$.

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 $(1 - (d-1)\epsilon_3)P^M(r)$. For $q \in Q_d$, The total weight of the major copies of q is thus at least

$$\sum_{r,\sigma:\, r \xrightarrow{\sigma} q} (1 - (d-1)\epsilon_3)P^M(r) \cdot \gamma(r,\sigma) \;=\; (1 - (d-1)\epsilon_3)P^M(q) \;. \tag{11}$$

There are at most $n|\Sigma|$ major copies of q, and hence the weight of the non-dominant ones is at most $2n|\Sigma|\epsilon_0 < \epsilon_3\epsilon_2$ and the claim follows.

We next prove Claim (2). We break the analysis into two cases. If $\gamma(q,\sigma) \leq \gamma_{min} + \epsilon_1$, then since $\hat{\gamma}(\hat{q},\sigma) \geq \gamma_{min}$ by definition, and $\epsilon_1 \leq \epsilon^2/(8(|\Sigma|+1))$, if we choose $\gamma_{min} = \epsilon/(4|(\Sigma|+1))$, then $\gamma(q,\sigma)/\hat{\gamma}(\hat{q},\sigma) \leq 1 + \epsilon/2$, as required.

If $\gamma(q,\sigma) > \gamma_{min} + \epsilon_1$, then let $\gamma(q,\sigma) = \gamma_{min} + \epsilon_1 + x$, where $x > 0$. Based on our choice of ϵ_2 and ϵ_3, for every $d \leq D$, $\epsilon_2(1 - d\epsilon_3) \geq 2\epsilon_0$. By the goodness of the sample, and the definition of $\hat{\gamma}(\cdot,\cdot)$, we have that

$$\begin{aligned} \hat{\gamma}(\hat{q},\sigma) &\geq (\gamma(q,\sigma) - \epsilon_1)(1 - (|\Sigma|+1)\gamma_{min}) + \gamma_{min} & (12)\\ &= (x + \gamma_{min})(1 - \epsilon/4) + \gamma_{min} & (13)\\ &\geq \frac{x + \gamma_{min}(1+\epsilon/2)}{1+\epsilon/2} \geq \frac{\gamma(q,\sigma)}{1+\epsilon/2} \;. & (14) \end{aligned}$$

■

Proof of Theorem 1: We prove the theorem based on Lemma 5.2. 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.

$$\begin{aligned} \mathcal{D}_{KL}(\,P^M \| P^{\widehat{M}}\,) &= \sum_{\sigma_1 \ldots \sigma_D} P^M(\sigma_1 \ldots \sigma_D) \log \frac{P^M(\sigma_1 \ldots \sigma_D)}{P^{\widehat{M}}(\sigma_1 \ldots \sigma_D)} \\ &= \sum_{\sigma_1} \sum_{\sigma_2 \ldots \sigma_D} P^M(\sigma_1) P^M(\sigma_2 \ldots \sigma_D|\sigma_1) \left[\log \frac{P^M(\sigma_1)}{P^{\widehat{M}}(\sigma_1)} + \log \frac{P^M(\sigma_2 \ldots \sigma_D|\sigma_1)}{P^{\widehat{M}}(\sigma_2 \ldots \sigma_D|\sigma_1)} \right] \\ &= \sum_{\sigma_1} P^M(\sigma_1) \log \frac{P^M(\sigma_1)}{P^{\widehat{M}}(\sigma_1)} \end{aligned}$$

$$
\begin{aligned}
+ \quad & \sum_{\sigma_1} P^M(\sigma_1) \cdot \mathcal{D}_{KL}\left(P^M(\sigma_2 \ldots \sigma_D|\sigma_1) \| P^{\widehat{M}}(\sigma_2 \ldots \sigma_D|\sigma_1)\right) \\
= \quad & \sum_{\sigma_1} P^M(\sigma_1) \log \frac{P^M(\sigma_1)}{P^{\widehat{M}}(\sigma_1)} + \sum_{\sigma_1} P^M(\sigma_1) \sum_{\sigma_2} P^M(\sigma_2|\sigma_1) \log \frac{P^M(\sigma_2|\sigma_1)}{P^{\widehat{M}}(\sigma_2|\sigma_1)} \\
+ \quad & \ldots \\
+ \quad & \sum_{\sigma_1 \ldots \sigma_d} P^M(\sigma_1 \ldots \sigma_d) \sum_{\sigma_{d+1}} P^M(\sigma_{d+1}|\sigma_1 \ldots \sigma_d) \log \frac{P^M(\sigma_{d+1}|\sigma_1 \ldots \sigma_d)}{P^{\widehat{M}}(\sigma_{d+1}|\sigma_1 \ldots \sigma_d)} \\
+ \quad & \ldots \\
+ \quad & \sum_{\sigma_1 \ldots \sigma_{D-1}} P^M(\sigma_1 \ldots \sigma_{D-1}) \sum_{\sigma_D} P^M(\sigma_D|\sigma_1 \ldots \sigma_{D-1}) \log \frac{P^M(\sigma_D|\sigma_1 \ldots \sigma_{D-1})}{P^{\widehat{M}}(\sigma_D|\sigma_1 \ldots \sigma_{D-1})} \\
= \quad & \sum_{d=0}^{D-1} \sum_{q \in Q_d} \sum_{\hat{q} \in \hat{Q}_d} P^M\left(W(q) \bigcap W(\hat{q})\right) \sum_{\sigma} P_q^M(\sigma) \log \frac{P_q^M(\sigma)}{P_{\hat{q}}^{\widehat{M}}(\sigma)} \\
= \quad & \sum_{d=0}^{D-1} \sum_{q \in Q_d} P^M(q) \sum_{\hat{q} \in \hat{Q}_d} P^M\left(W(q) \bigcap W(\hat{q})\right) / P^M(q) \sum_{\sigma} P_q^M(\sigma) \log \frac{P_q^M(\sigma)}{P_{\hat{q}}^{\widehat{M}}(\sigma)} \\
\leq \quad & \sum_{d=0}^{D-1} \sum_{q \in Q_d,\ P^M(q) < \epsilon_2} P^M(q) \log(1/\gamma_{min}) \\
+ \quad & \sum_{d=0}^{D-1} \sum_{q \in Q_d,\ P^M(q) \geq \epsilon_2} P^M(q)[(1 - d\epsilon_3)\log(1 + \epsilon/2) + d\epsilon_3 \log(1/\gamma_{min})] \\
\leq \quad & (nD\epsilon_2 + D^2\epsilon_3)\log(1/\gamma_{min}) + \epsilon/2
\end{aligned}
$$

If we choose ϵ_2 and ϵ_3 so that $\epsilon_2 \leq \epsilon/(4nD\log(1/\gamma_{min}))$ and $\epsilon_3 \leq \epsilon/(4D^2\log(1/\gamma_{min}))$, then the expression above is bounded by ϵ, as required. Adding the requirements on ϵ_2 and ϵ_3 from Lemma 5.2, we get the following requirement on ϵ_0:

$$
\epsilon_0 \leq \epsilon^2/(32n^2|\Sigma|D^3\log^2(4(|\Sigma|+1)/\epsilon)) \ ,
$$

from which we can derive a lower bound on m by applying Lemma 5.1. ■

6 An Online Version of the Algorithm

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

6.1 An Online Learning Model

In the online setting, the algorithm is presented with an infinite sequence of *trials*. At each time step, t, the algorithm receives a trial string $s^t = s_1 \ldots s_{\ell-1}\zeta$ 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 $\log(P^M(s^t)/P_t(s^t))$. We shall be interested in the *average* error $Err_t \stackrel{\mathrm{def}}{=} \frac{1}{t}\sum_{t'\leq t} err_t(s^t)$.

We allow the algorithm to err an *unrecoverable error* at some stage t, with total probability that is bounded by Δ.[5] We ask that there exist functions $\delta(t,\mu,n,D,|\Sigma|,\Delta)$, and $\epsilon(t,\mu,n,D,|\Sigma|,\Delta)$, such that the following hold. $\delta(t,\mu,n,D,|\Sigma|,\Delta)$ 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 $\epsilon(t)$ 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 $\epsilon(t)$. For every trial t, if the algorithm has not erred an unrecoverable error prior to that trial, then with probability at least $1-\delta(t)$, the average error is small, namely $Err_t \leq \epsilon(t)$. We thus require that as the algorithm gets more trial strings, it makes better predictions with high probability. 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 online algorithm.

6.2 An Online Learning Algorithm

We now describe how to modify the *batch* algorithm Learn-Acyclic-PFA, presented in Section 4, to become an online 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 $\sigma \in \Sigma\bigcup\{\zeta\}$, $m_{v_0}(\sigma)=0$ (and hence, by definition, $m_{v_0}=0$). Given a new trial string s^t, the algorithm checks if 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 $s^t=s_1\ldots s_\ell$, and let $v_0\ldots v_\ell$ be the nodes on the path corresponding to s^t. Then the algorithm outputs the following product:

$$P_t(s^t) = \Pi_{i=0}^{\ell-1}\left(\frac{m_{v_i}(s_{i+1})}{m_{v_i}}(1-(|\Sigma|+1)\gamma_{min}(t)) + \gamma_{min}(t)\right),$$

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

The algorithm adds s^t to $G(t)$, and 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, $m_v \geq m_0$, 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.

[5] Allowing the algorithm to err such an unrecoverable error is an artifact of the 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.

Algorithm Online-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 $v_0 \ldots v_\ell$ be the nodes on the path corresponding to s^t in $G(t)$;
   (d) Output: $P_t(s^t) = \Pi_{i=0}^{\ell-1}(\frac{m_{v_i}(s_{i+1})}{m_{v_i}}(1-(|\Sigma|+1)\gamma_{min}(t)) + \gamma_{min}(t))$;
   (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 $m_{v_i} = m_0$ 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. $G(t+1) := G_i$, $t := t+1$.

Theorem 2 *Algorithm* Online-Learn-Acyclic-PFA *is an efficient online learning algorithm for APFAs.*

Proof Idea: The proof of the theorem essentially follows the same arguments used in the proof of Theorem 1 and so we present the main new ideas. The first key observation is the following. Consider some trial t. Assume that up till trial t the following was true: for every pair of nodes u and v such that $m_u \geq m_0$ and $m_v \geq m_0$, u and v are folded 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 get had we run the batch algorithm on the set of trial strings observed up till 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 online 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 nodes do 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 up till trial t, we can use the result we have for the batch algorithm to get bounds on $\epsilon(t)$ and $\delta(t)$. In the analysis of the batch algorithm, for every given ϵ and δ we got a lower bound on m (whose role is taken by t) that ensured error ϵ with confidence $1-\delta$. Since this bound was polynomial in $1/\epsilon$ and $\log(1/\delta)$, we can define a pair of functions $\epsilon(t)$ and $\delta(t)$ which decrease with t as required[6]. The intuition behind this behavior of the algorithm is that if no bad event occurs, we expect that as t increases, we both 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). As for the parameter $\gamma_{min}(t)$, it is defined to have the same functional relation with $\epsilon(t)$ as γ_{min} has with ϵ in the batch algorithm.

[6]To be a little more precise, $\epsilon(t)$ 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 H_t and M. Since the expectation of err_t decreases with t (in a polynomial rate and with exponentially increasing probability), so does the expected value of Err_t. As t increases, the deviation of err_t from it's expectation decreases with exponentially increasing probability.

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 online 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 $\mathcal{M}$ as defined in the batch algorithm. However, we can slightly modify the definition of $\mathcal{M}$ to answer this problem. We now allow a subset of the dominant major copies of each state to be merged into single *super* dominant major copy. Each state q can 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 the same as for dominant major copies. Edges that previously entered major copies that belong to a super copy, now enter the super copy. Each automata 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 $\mathcal{M}$ is therefore bounded by the square of the size of $\mathcal{M}$ as defined for the batch algorithm and is at most $2^{|\Sigma|n^2 D}$. Note that M is the automaton in $\mathcal{M}$ 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 $M' \in \mathcal{M}$, and for every state $q' \in Q'$, if $m_{q'} \geq m_0$, then for every string s, $|m_{q',s}/m_{q'} - P_{q'}^{M'}(s)| \leq \mu/4$. Let

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

Then similarly to the proof of Lemma 5.1, for a given q', if $m_{q'} = m_0 + x$ for some integer $x \geq 0$, 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, $m_{q'}$ either remains the same or grows by 1. If it remains the same 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 $m_{q'}$ grows 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 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 $\mathcal{M}$ 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 $m_v \geq m_0$. 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 $D\,n$. 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 $D\,m_0$ 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 $G(t)$, $m_v(\sigma) \leq t$, the counts on the edges contribute a factor of $\log t$ to the total size 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 trail string, and their successors. $\square$

7 Applications

A slightly modified version of our learning algorithm was applied and tested on various problems such as: stochastic modeling of cursive handwriting [18], locating noun phrases in natural English text, and building multiple-pronunciation models for spoken words from their phonetic transcription. This modified version of the algorithm allows folding states from different levels, thus the resulting hypothesis is more compact. We also chose to fold 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 acyclic PFAs 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], the second and the third authors proposed a dynamic encoding scheme for cursive handwriting based on an oscillatory model of handwriting. The process described in [17] performs inverse mapping from continuous pen trajectories to strings over a discrete set of symbols which efficiently encode cursive handwriting. These symbols are named *motor control commands*. Using a forward model the motor control commands can be transformed back into pen trajectories and the handwriting can be reconstructed (without the "noise" that was eliminated by the inverse mapping). Each possible control command is composed of a cartesian product of the form $\mathcal{X} \times \mathcal{Y}$ where $\mathcal{X}, \mathcal{Y} \in \{0, 1, 2, 3, 4, 5\}$, hence the alphabet consists of 36 different symbols. These symbols represents quantized horizontal and vertical amplitude modulation and their phase-lags. The symbol 0×0 represents zero modulation 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 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 of 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 of the 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. The forward dynamic 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 Figure 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.

Given the above set of APFAs, 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

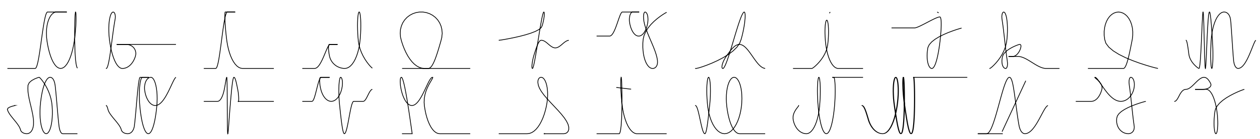

Figure 3: Synthetic cursive letters, created by random walks on the 26 APFAs.

its different letter constituents. Recognition of completely unlabeled data is more involved, but can be performed efficiently using a higher level language model (see [15] for an example of such a model). A complete description of the cursive handwriting recognition system is given in [18].

When a transcription of a cursively written word (i.e., the letters that constitute the word) is given, we find the most likely segmentation of that word as follows. The segmentation partitions the motor control commands into non-overlapping segments, where each segment corresponds to a different letter. Denote the control commands of a word by $s = s_1, s_2, \ldots, s_L$ and the letters that constitute a word by $\sigma_1, \sigma_2, \ldots, \sigma_K$. A segmentation is a set of $K+1$ indices, denoted by $\mathcal{I} = i_0, i_1, \ldots, i_K$, such that $i_0 = 1$, $i_K = L+1$, and for all $0 \leq j < K : i_j < i_{j+1}$. We associate with each cursive letter an APFA that approximates the distribution of the possible motor control commands which may represent that letter. Let the probability of a string s to be produced by a model corresponding to the letter σ be denoted by $P^{\sigma}(s)$. The probability of a segmentation, $\mathcal{I}$, for a sequence s and its transcription $\sigma_1, \sigma_2, \ldots, \sigma_K$, given a set of APFAs, is

$$P(\mathcal{I} | (\sigma_1, \sigma_2, \ldots, \sigma_K), (s_1, \ldots, s_L)) = \prod_{k=1}^{K} P^{\sigma_k}(s_{i_{k-1}}, \ldots, s_{i_k - 1}) \, . \tag{15}$$

The most likely segmentation for a transcribed word can be found efficiently by using a dynamic programming scheme as follows. Define $Seg(n,k)$ to be the most likely partial segmentation of the prefix of s, $s_1, \ldots, s_n$, by the k letters prefix of the word, $\sigma_1, \ldots, \sigma_k$. $Seg(n,k)$ is calculated recursively through,

$$Seg(n,k) = \max_{n'} \; Seg(n', k-1) \times P^{\sigma_k}(s_{n'+1}, \ldots, s_n) \, . \tag{16}$$

The probability of the most likely segmentation is $Seg(L,K)$. The most likely segmentation itself is found by keeping the indices that maximized Equation (16) for all possible n and k and backtracking these indices from $S(L,K)$ back to $S(0,0)$. An example of the result of such a segmentation is depicted in Figure 4, 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 online 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 continue with an online 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, the phoneme `d` in the word `muddy` might be flapped, *etc.* One possible approach to model such pronunciation variations is to construct stochastic models that

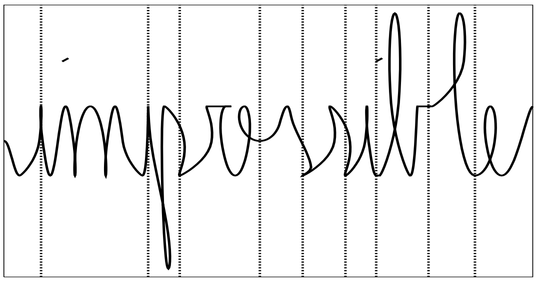

Figure 4: *Temporal* segmentation of the word `impossible`. The segmentation is performed by evaluating the probabilities of the APFAs which correspond to the letter constituents of the word.

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 apriori 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 in [19]. 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 was 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 Figure 5. 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 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 [19]. That is, words occurring between 20 and 100 times in the data set were used for training and evaluation according to the following partition. 75% 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 proper model on the phonetic transcription of 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 in [19]. 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 method of choice for this problem when large amounts of training data are presented.

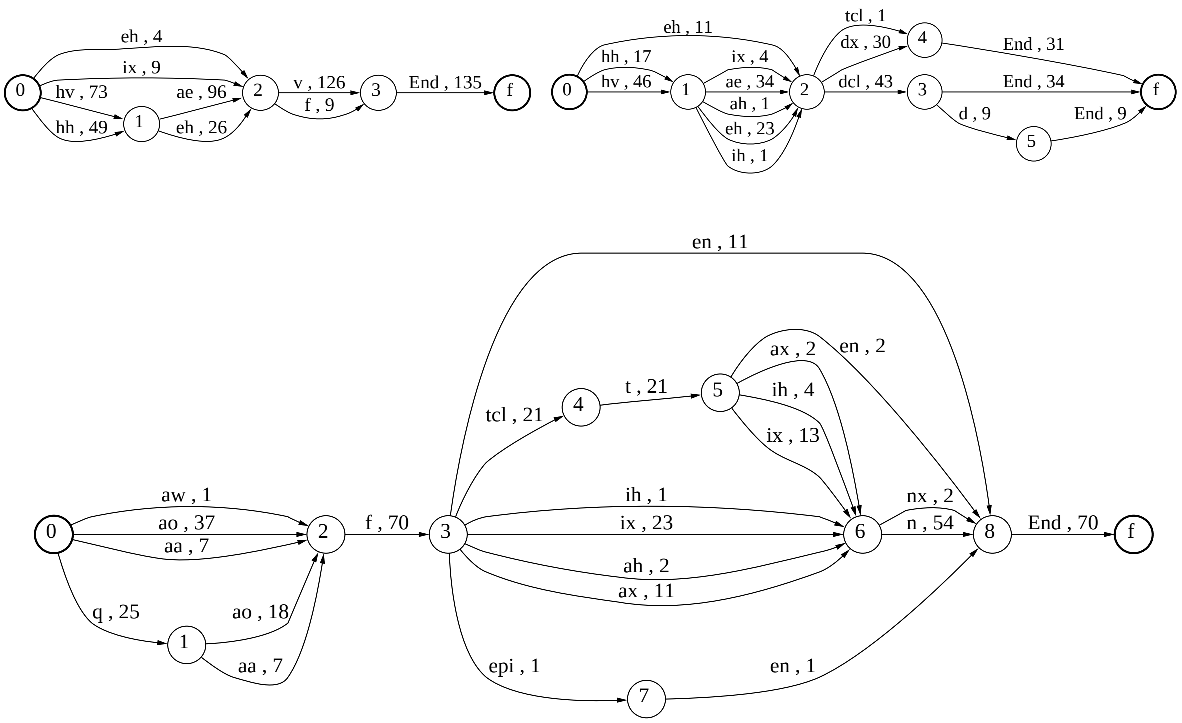

Figure 5: An example of pronunciation models based on APFAs for the words `have, had` and `often` trained from the TIMIT database.

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 statisticly similar should work for PFAs which have cycles in them (but are μ-distinguishable for non-negligble 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 that when the sample is generated according to a source which is not an APFA (or not even a PFA), then the learning algorithm finds a hypothesis which

Model	*Log-Likelihood*	*Perplexity*	*States*	*Transitions*	*Training Time*
APFA	-2142.8	1.563	1398	2197	23 **seconds**
HMM [19]	-2343.0	1.849	1204	1542	29:49 **minutes**

Table 1: The performance of APFAs compared to Hidden Markov Models (HMM) as reported in [19] by Stolcke and Omohundro. *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.

is close to the APFA which best approximates the target source.

Finally, it might be of value to find other subclasses of probabilistic automata which can serve as good models for various families of natural sequences and have efficient learning algorithms.

Acknowledgements

We would like to 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 reference [4]. We would also like to 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 would like to thank the support of the Eshkol fellowship. Yoram Singer would like to thank the Clore Foundation for its support.

References

[1] N. Abe and M. Warmuth. On the computational complexity of approximating distributions by probabilistic automata. *Machine Learning*, 9:205–260, 1992.

[2] L.E Baum and T. Petrie. Statistical inference for probabilistic functions of finite state markov chains. *Annals of Mathematical Statistics*, 37, 1966.

[3] Y. Bengio, Y. le Cun, and D. Henderson. Globally trained handwritten word recognizer using spatial representation, convolutional neural networks, and hidden Markov models. In *Advances in Neural Information Processing Systems*, volume 6. Morgan Kaufmann, 1993.

[4] R.C. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *The 2nd Intl. Collo. on Grammatical Inference and Applications*, pages 139–152, 1994.

[5] F.R. Chen. Identification of contextual factos for pronounciation networks. In *Proc. of IEEE Conf. on Acoustics, Speech and Signal Processing*, pages 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 random walks. In *Proceedings of the 24th Annual ACM Symp. on Theory of Computing*, pages 315–324, 1993.

[7] W. Hoeffding. Probability inequalities for sums of bounded random variables. *American Statistical Association Journal*, 58:13–30, 1963.

[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 Symp. on Theory of Computing*, 1994.

[9] N. Merhav and Y. Ephraim. Maximum likelihood hidden Markov modeling using a dominant sequence of states. *IEEE Trans. on ASSP*, 39(9):2111–2115, 1991.

[10] R. Plamondon and C.G Leedham, editors. *Computer Processing of Handwriting.* World Scientific, 1990.

[11] R. Plamondon, C.Y Suen, and M.L. Simner, editors. *Computer Recognition and Human Production of Handwriting.* World Scientific, 1989.

[12] L.R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proc. of the IEEE*, 1989.

[13] L.R. Rabiner and B. H. Juang. An introduction to hidden markov models. *IEEE ASSP Magazine*, 3(1):4–16, January 1986.

[14] M.D. Riley. A statistical model for generating pronounication networks. In *Proc. of IEEE Conf. on Acoustics, Speech and Signal Processing*, pages 737–740, 1991.

[15] 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. (To appear in Machine Learning).

[16] D. Sankoff and J.B. Kruskal. *Time warps, string edits and macromolecules: the theory and practice of sequence comparison.* Addison-Wesley, Reading Mass, 1983.

[17] Y. Singer and N. Tishby. Dynamical encoding of cursive handwriting. *Biological Cybernetics*, 71(3):227–237, 1994.

[18] Y. Singer and N. Tishby. An adaptive cursive handwriting recognition system. Technical Report CS-TR-22, Hebrew University, 1995.

[19] A. Stolcke and S. Omohundro. Hidden Markov model induction by Bayesian model merging. In *Advances in Neural Information Processing Systems*, volume 5. Morgan Kaufmann, 1992.

[20] C.C. Tappert, C.Y. Suen, and T. Wakahara. The state of art in on-line handwriting recognition. *IEEE Trans. on Pat. Anal. and Mach. Int.*, 12(8):787–808, 1990.

[21] B. A. Trakhtenbrot and Ya. M. Brazdin'. *Finite Automata: Behavior and Synthesis.* North-Holland, 1973.