On the Learnability and Usage of Acyclic Probabilistic Finite Automata

Dana Ron Yoram Singer Naftali Tishby
Institute of Computer Science and
Center for Neural Computation
Hebrew University, Jerusalem 91904, Israel
Email: {danar,singer,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 short 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 "words" in a language or short protein sequences.

The common approaches to the modeling and recognition of such sequences 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 [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 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.

On the other hand, HMMs, which are popular in speech recognition and have better ability to capture context dependent variations, suffer from both practical and theoretical drawbacks. The commonly used training procedure for HMMs is based on the *forward-backward* algorithm [2]. This algorithm 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, 7]. In addition, the successful applications of the HMM approach occur mostly in cases where its full power is not utilized. Namely, there is one, most probable, state sequence (the Viterbi sequence) which captures most of the likelihood of the model given the observations, so that practically the states are not truly hidden [8]. Another drawback of HMMs is that the current HMM training algorithms are neither online nor adaptive in the model's topology. These weak aspects of the hidden Markov model motivate our present modeling technique.

The alternative we consider here is using *Acyclic Probabilistic Finite Automata* (APFA) 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

COLT''95 Santa Cruz, CA USA© 1995 ACM 0-89723-5/95/0007..$3.50

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 [14] we introduced an algorithm for learning distributions (on long strings) generated by ergodic Markovian sources which can be described by a different subclass of 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 and similar problems [18].

More formally, we present an algorithm for efficiently 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.

Our result should be contrasted with the intractability result for learning PFAs described by Kearns *et. al.* [7]. They show that PFAs are not efficiently learnable under the widely acceptable assumption that there is no efficient algorithm for learning noisy parity functions 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 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).

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 our main theorem concerning the correctness of the learning algorithm, and give a skeleton of its proof. The full proofs of our main theorem and all additional lemmas are given in [15]. In Section 6 we describe two applications of our algorithm, and in Section 7 we give an online version of the algorithm.

Other Related Work: A similar technique of merging states 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 and reviews of practical models and algorithms for multiple-pronunciation can be found in [5, 13], and for cursive handwriting recognition in [10, 9, 20, 3].

2 Preliminaries

A *Probabilistic Finite Automaton (PFA)* M is a 7-tuple $(Q, q_0, q_f, \Sigma, \zeta, \tau, \gamma)$[2] 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*;
- $\tau : Q \times \Sigma \bigcup \{\zeta\} \rightarrow Q \bigcup \{q_f\}$ is the *transition function*;
- $\gamma : Q \times \Sigma \bigcup \{\zeta\} \rightarrow [0, 1]$ is the *next symbol probability function*.

The function γ must satisfy the following requirement: for every $q \in Q$, $\sum_{\sigma \in \Sigma} \gamma(q, \sigma) = 1$. 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 $Q \times \Sigma^*$ 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 $\tau(q, \mathtt{e}) = q$ where $\mathtt{e}$ is the empty string.

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 $\sigma \in \Sigma$ 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

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

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

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

[1] DFAs in which the underlying graph is arbitrary, but the accept/reject labels on the states are chosen randomly.

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

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

Given a state q in Q, and a string $s = s_1 \ldots s_l$ (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. More formally

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

The following definition is central to this work.

Definition 1 *For $0 \leq \mu \leq 1$, we say that two states, q_1 and q_2 in Q are* μ-distinguishable, *if there exists a string s for which $|P_{q_1}^M(s) - P_{q_2}^M(s)| \geq \mu$. 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 is defined to be the length of the longest path from q_0 to q_f. In particular, we consider *leveled* acyclic PFAs. In such a PFA, 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 PFA. 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. The following claim can easily be verified.

Lemma 2 *For every acyclic PFA M having n states and depth D, there exists an equivalent leveled acyclic PFA, $\tilde{M}$, with at most $n(D-1)$ states.*

3 The Learning Model

In this section we describe our learning model which is similar to the one introduced in [7]. We start by defining an *ϵ-good* hypothesis PFA with respect to a given target PFA.

Definition 3 *Let M be the target PFA and let $\widehat{M}$ be a hypothesis PFA. 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*

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

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:*

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

[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-negligble weight be distinguishable

Our learning algorithm for PFAs is given a *confidence* parameter $0 < \delta \leq 1$, and an *approximation* parameter $\epsilon > 0$. The algorithm is also given an upper bound n on the number of states in M, and a *distinguishability* parameter $0 < \mu \leq 1$, indicating that the target automaton is μ-distinguishable.[4] The algorithm has access to strings generated by the target PFA, and we ask that it output with probability at least $1 - \delta$ an ϵ-good hypothesis with respect to the target PFA. We also require that the learning algorithm be *efficient*, i.e., that it runs in time polynomial in $\frac{1}{\epsilon}$, $\log \frac{1}{\delta}$, $|\Sigma|$, and in the bounds on $\frac{1}{\mu}$ and n.

4 The Learning Algorithm

In this section we describe our algorithm for learning acyclic PFAs. An *online* version of this algorithm is described in Section 7.

Let S be a given multiset of sample strings generated by the target PFA 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. In each of these graphs, 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., $S(v) \stackrel{\text{def}}{=} \{s : s = s's'' \in S,\ v_0 \stackrel{s'}{\Rightarrow} v\}_{multi}$. 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, $S_{gen}(v) \stackrel{\text{def}}{=} \{s'' : \exists s' \text{ s.t. } s's'' \in S \text{ and } v_0 \stackrel{s'}{\Rightarrow} v\}_{multi}$. 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.

The initial graph G_0 is the *sample tree*, T_S. 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.

[4] These last two assumption can be removed by *searching* for an upper bound on n and a lower bound on 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

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.

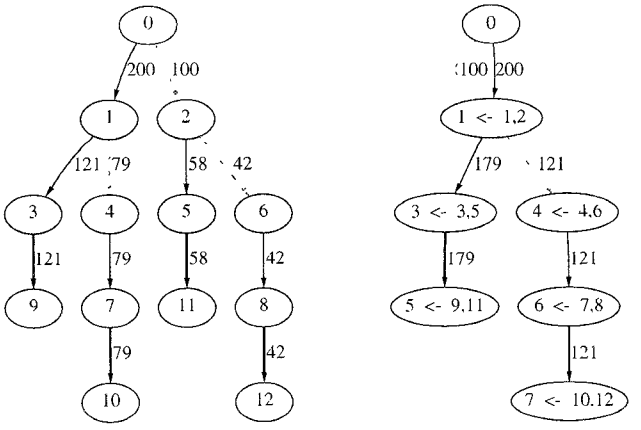

Figure 1: An illustration of the folding operation. The graph on the right is constructed from the graph on the left by merging the nodes v_1 and v_2. The different edges represent different output symbols: gray is 0, black is 1 and bold black edge is ζ.

Let G_N be the last graph in this series for which there does not exist such a pair of 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 PFA $\widehat{M}$ based on G_{N+1}. We let G_{N+1} be the underlying graph of $\widehat{M}$, where v_0 corresponds to q_0, and v_f corresponds to q_f. For every state q in level d that corresponds to a node u, and for every symbol $\sigma \in \Sigma \bigcup \{\zeta\}$, we define

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

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 $m_v(s) \stackrel{\text{def}}{=} |\{t : t \in S_{gen}(v),\ t = st'\}_{multi}|$. We say that a given pair of nodes u and v, are *similiar* 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 below.

For sake of simplicity of the pseudo-code below, 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 u and v, whose counts, m_u and m_v, are above the threshold m_0. If the function returns *similar*, then the algorithm merges u and v 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 below in the routine **AddSlack**. The final graph, G_{N+1} is then transformed into a PFA while smoothing the transition probabilities (Procedure **GraphToPFA**).

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) $p'_u = p_u m_u(\sigma)/m_u$; $p'_v = p_v m_v(\sigma)/m_v$;
   (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 our main theorem regarding the correctness and efficiency of the learning algorithm Learn-Acyclic-PFA, described in Section 4, and give a skeleton of its proof. The full proofs of the main theorem and the technical lemmas appear in [15].

Theorem 4 *For every given distinguishability parameter* $0 < \mu \leq 1$, *for every* μ*-distinguishable target acyclic PFA* M, *and for every given security parameter* $0 < \delta \leq 1$, *and approximation parameter* $\epsilon > 0$, *Algorithm* Learn-Acyclic-PFA *outputs a hypothesis PFA,* $\widehat{M}$, *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 $P^M(q_1)$ and $P^M(q_2)$ 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) we may assume that M is a leveled acyclic PFA 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 4, we first need to define the notion of a *good* sample with respect to a given target (leveled) PFA. We prove that with high probability a sample generated by the target PFA is good. We then show that if a sample is good then our algorithm constructs a hypothesis PFA 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 by our algorithm, we introduce a class of PFAs $\mathcal{M}$, 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) 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.

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 PFA M let $\mathcal{M}$ be the set of PFAs $\{M' = (Q', q'_0, \{q'_f\}, \Sigma, \tau', \gamma', \zeta)\}$ 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 $M' \equiv M$, 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.

By the definition above, each PFA in $\mathcal{M}$ is fully characterized by the choices of the sets of dominant copies among the major copies of each state. 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 8, if the sample is good, then there exists a correspondence between some PFA in $\mathcal{M}$ and the graphs our algorithm constructs. We use this correspondence to prove Theorem 4.

Definition 5 *A sample S of size m is (ϵ_0, ϵ_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 6 *With probability at least $1 - \delta$, a sample of size*
$$m \geq \max\left(\frac{|\Sigma|\, n^2 D + \ln \frac{2D}{\epsilon_0 \delta}}{\epsilon_0^2}, \frac{m_0}{\epsilon_0}\right) ,$$
is (ϵ_0, ϵ_1)-good with respect to M.

5.2 Technical Lemmas

The proof of Theorem 4 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 7 *If the sample is (ϵ_0, ϵ_1)-good for*
$$\epsilon_1 < \min(\mu/4, \epsilon^2/8(|\Sigma|+1)) ,$$
then for $\epsilon_3 \leq 1/(2D)$, and for $\epsilon_2 \geq 2n|\Sigma|\epsilon_0/\epsilon_3$, 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 7 can be derived based on the following lemma in which we show a relationship between the graphs constructed by the algorithm and a PFA in $\mathcal{M}$.

Lemma 8 *If the sample is (ϵ_0, ϵ_1)-good, for $\epsilon_1 < \mu/4$, then there exists a PFA $M' \in \mathcal{M}$, $M' = (Q', q'_0, \{q'_f\}, \Sigma, \tau', \gamma', \zeta)$, 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 *$m_{q'} \geq m_0$.*

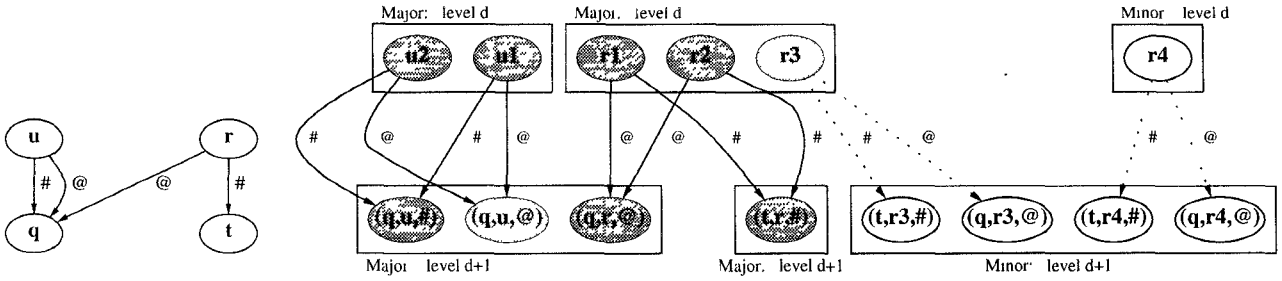

Figure 2: Left: Part of the original automaton, M, that corresponds to the copies on the right part of the figure. Right: The different types of copies of M's states: copies of a state are of two types *major* and *minor*. A subset of the major copies of every state is chosen to be *dominant* (dark-gray nodes). The major copies of a state in the next level are the next states of the dominant states in the current level.

6 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.

6.1 Stochastic Models for Cursive Scripts

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.

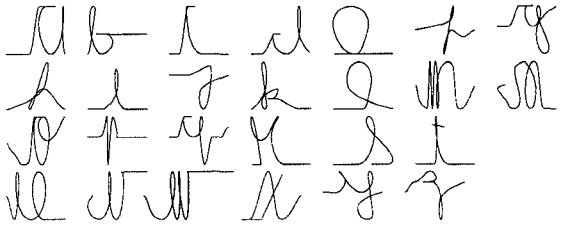

Figure 3: Synthetic cursive letters, created by random walks on the 26 APFAs.

Given the above set of APFAs, we can perform tasks such as segmentation of cursive words and recognition of unlabeled words. 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. 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 [18].

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

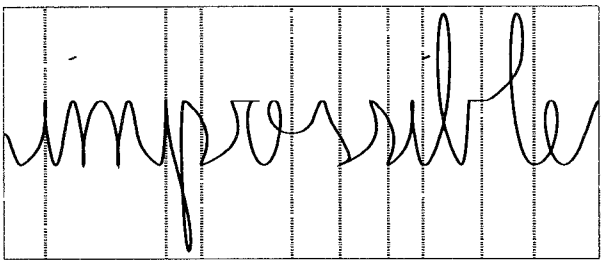

Figure 4: *Temporal* segmentation of the word impossible. The segmentation is performed by evaluating the probabilities of the APFAs which correspond to the letter constituents of the word.

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.

6.2 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 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 evaluation. Of these, 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.

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

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.

7 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.

7.1 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 machine, 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 a PFA, 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. 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 cumulative* error,

$$Err_t \stackrel{\text{def}}{=} \frac{1}{t}\sum_{t' \le t} err_t(s^t) \ .$$

We allow the algorithm to err an *unrecoverable error* at some stage t, with total probability that is bounded by Δ. We ask

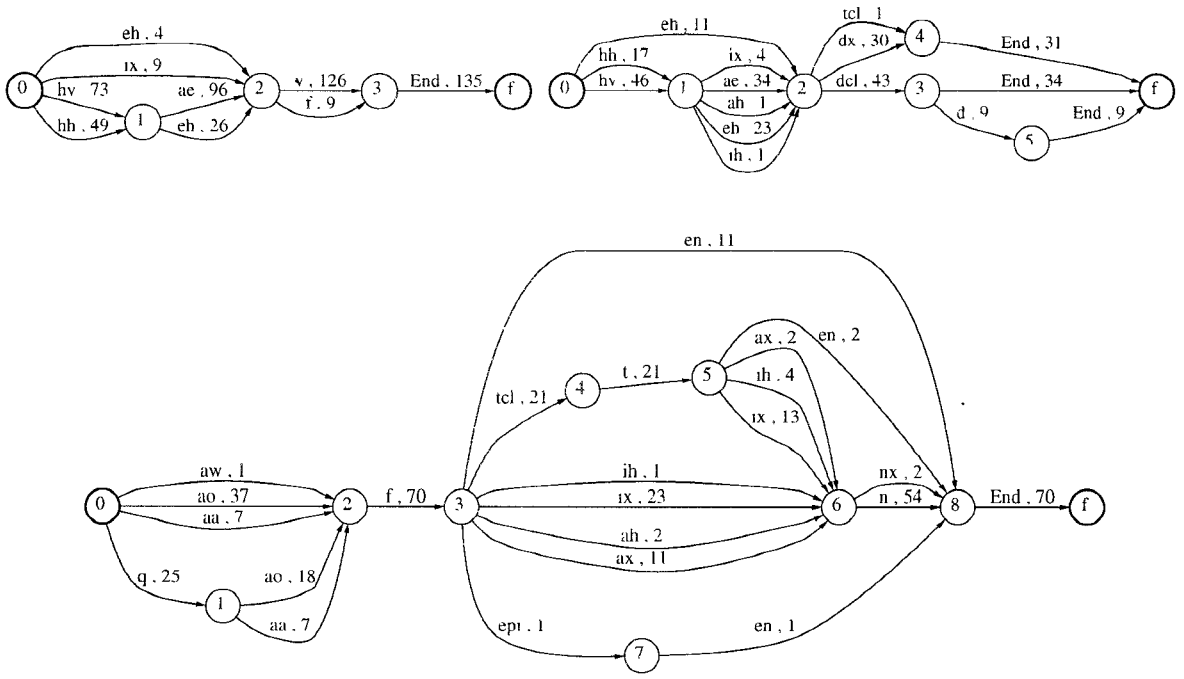

Figure 5: An example of pronunciation models based on APFAs for the words `have`, `had` and `often` trained from the TIMIT database.

that there exist functions $\delta(t, \mu, n, D, |\Sigma|)$, and $\epsilon(t, \mu, n, D, |\Sigma|)$, such that the following hold. $\delta(t, \mu, n, D, |\Sigma|)$ is of the form $\beta_1(\mu, n, D, |\Sigma|)2^{t^{-\alpha_1}}$, where β_1 is a polynomial in μ, n, D, and $|\Sigma|$, and $0 < \alpha_1 < 1$, and $\epsilon(t)$ is of the form $\beta_2(\mu, n, D, |\Sigma|)t^{-\alpha_2}$, where β_2 is a polynomial in μ, n, D, and $|\Sigma|$, and $0 < \alpha_2 < 1$. Since we are mainly interested in the dependence of the 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 cumulative error is small, namely $Err_t \leq \epsilon(t)$. 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 PFA "from scratch" is not considered an online algorithm.

7.2 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 is presented at the end of the section. 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:

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

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

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.

In the algorithm described above, as in the batch 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. Similarly to the analysis in the batch algorithm, it can be shown that for an appropriate choice of m_0, the probability that we perform such a merge at any time in the algorithm, is bounded by Δ. If we never perform such merges, 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). A more detailed analysis can give precise bounds on $\delta(t)$ and $\epsilon(t)$.

What about the size of our hypotheses? Let a node v be called *reliable* if $m_v \geq m_0$. Using the same argument needed for showing that with probability at least $1 - \Delta$ we never merge nodes that correspond to different states, we get that with the same probability we merge every pair of 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 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.

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 1.
   (c) Let $v_0 \ldots v_\ell$ be the nodes on the path corresponding to s^t in $G(t)$;
   (d) Output: $P_t(s^t) =$

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

   (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. $G(t+1) := G_i$, $t := t+1$.

Acknowledgements

We would like to thank an anonymous 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] 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.

[8] 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.

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

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

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

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

[13] 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.

[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. (To appear in Machine Learning).

[15] D. Ron, Y. Singer, and T. Tishby. On the learnability and usage of acyclic probabilistic finite automata. Technical Report CS-TR-23, Hebrew University, 1995.

[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.