Passively Learning Finite Automata

Kevin P. Murphy*

16 November 1995

Abstract

We provide a survey of methods for inferring the structure of a finite automaton from passive observation of its behavior. We consider both deterministic automata and probabilistic automata (similar to Hidden Markov Models). While it is computationally intractible to solve the general problem exactly, we will consider heuristic algorithms, and also special cases which are tractible. Most of the algorithms we consider are based on the idea of building a tree which encodes all of the examples we have seen, and then merging equivalent nodes to produce a (near) minimal automaton.

Contents

*murphyk@cs.ucdavis.edu

APFA Acyclic deterministic Probabilistic Finite Automaton. A DPFA in which the underlying topology is acyclic.

DFA Deterministic Finite Automaton.

DPFA Determinisitic Probabilistic Finite Automaton. This is a DFA in which states emit symbols probabilistically, but the next state is uniquely determined by the current state and the symbol.

FSM Finite State Machine. This is a DFA which converts input strings to output strings. Also called a transducer. There are two main kinds: a Moore machine associates a (deterministic) emission function with each state, and a Mealy machine associates a (deterministic) emission function with each arc.

HMM Hidden Markov Model. This is a special case of an NPFA in which the transition probabilities are independent of the symbol which is emitted; hence an HMM is a Markov Chain in which states emit symbols probabilistically. We usually specify the emission and transition probability distributions separately.

NFA Non-deterministic Finite Automaton. Like a DFA, except that there may be several possible next states given the current state and symbol.

NPFA Non-deterministic Probabilistic Finite Automaton. This is an NFA in which states emit symbols probabilistically, and the next state is chosen probabilistically (conditioned on the symbol). We usually specify the probability of emitting a symbol and transitioning to the next state jointly.

PFA Probabilistic Finite Automaton.

PSA Probabilistic Suffix Automaton. This is a special case of a DPFA in which each state label is a string of at most length L. It can be used to model a Markov Chain in which some states have order L, and other states have lower order.

Table 1: A list of the acronyms used in this paper.

1 Introduction

In their excellent survey paper, Angluin and Smith [AS83] define "inductive inference" as the "process of hypothesizing a general rule from examples". We are interested in the special case where the "rule" takes the form of a Deterministic or Probabilistic Finite Automaton (DFA or PFA), and the examples are (assumed to be) drawn from a (stochastic) regular language. (See Table 1 for a list of the acronyms used in this paper; the different models that we use will be defined more fully later.) We can think of this as the problem of trying to learn the structure inside of some "black box", which is continously emitting symbols.[1] Thus no experimentation or oracle queries are allowed. We concentrate on recent results which are not mentioned in the 1982 survey on inductive inference by Angluin and Smith and the 1990 survey on grammatical inference by Miclet [Mic90]. (See also the book by Fu [Fu82].)

1.1 Applications of automaton inference

Most of the work on automaton inference (especially the early work) is theoretical in nature: finite automata are simple, so we can characterize more easily how hard it is to learn them, both in terms of how large the

[1] We will consider two cases: one in which the target automaton (the one we are trying to learn) returns to its start state after it emits each string, and (in the case of PFAs only) one in which it runs continuously (i.e., there is no final state). We also assume that it is minimal, i.e., has no redundant states.

training set must be, and how much time is needed – that is, the sample and time complexity.

However, finite automaton inference also has several "real world" applications. In the electrical engineering community, there is considerable interest in synthesising Finite State Machines (FSMs are DFAs with output) from a partial specification of their behavior; see for example [OE95] and the references therein. DFAs have also been proposed as a model of players (in a game-theoretic sense) with bounded rationality; learning to play optimally against such players is very similar to the inference problem defined above; see for example [FKM+95] and references therein. Finally, we can imagine the problem of a robot trying to learn the structure of its environment, which can be modelled as a DFA; see [RS87, RS89, MB91].

The applications of PFAs, of which Hidden Markov Models (HMMs) are a special case, are much more extensive. HMMs have been very widely used in the speech and handwriting recognition communities (see e.g., [Rab89, RST95]), and have also recently been applied to recognizing patterns in biological sequences such as DNA and proteins (see e.g., [KMH93, KBM+94, BCHM94]). Most work on HMMs assumes the sructure or topology is specified in advance, and the learning procedure (which is usually called training in this context) merely consists of finding the right transition probabilities. (A similar comment can be made about neural networks.[2]) We are interested in the more general problem of learning the topology of the model, as well as the transition probabilities, starting from scratch. (The even more general problem of choosing a model class is discussed in [Cru91].) The hope is that PFAs can then be applied to domains where we have little or no prior knowledge as to what the correct structure should look like. As an example of this, Crutchfield and Young [CY89, You91] propose learning a PFA from a series of measurements taken from a non-linear dynamical system, and using the "complexity" of the resulting PFA as a measure of the complexity of the original system. (See also [Li90].)

1.2 Why PFAs instead of other probabilistic models?

We could consider the problem of learning probabilistic models which are more powerful than PFAs, such as belief (Bayesian) networks [Pea88, HGC94] or stochastic Context Free Grammars [JLM92], but learning PFAs will prove to be hard enough. In fact, the problem is so hard (i.e., time-consuming) that we will mostly concentrate on a special case, which we call "deterministic" PFAs (DPFAs). These are less powerful than HMMs, and hence easier to learn. In fact, experimental results in [RST95] show that it is 10–100 times faster to learn a pronunciation model for spoken words using a (certain kind of) DPFA than it is to learn a corresponding HMM, and yet the performance of the DPFA is actually slightly better. Also, it is possibly to come up with provably efficient algorithms to optimally learn this kind of DPFA, whereas optimally learning HMMs is provably hard — the algorithms used in practice only find a local optimum.

Another kind of DPFA, called a Probabilistic Suffix Automaton (PSA), can be viewed as a Markov chain in which each state has order at most n. This is much smaller than an explicit Markov chain in which every state has order exactly n; such a model has $O(|\Sigma|^n)$ states (where Σ is the alphabet), and hence requires an exponential amount of data to learn the transition probabilities. (A Markov chain of order n is also called an n-gram model.) Experimental results in [RST94] show it is possible to learn a DPFA that performs well on a task which involves correcting errors in text, but it is not yet clear if this method is better than other sparse n-gram methods such as hash tables. Also, the results of this model applied to a task which involved locating genes in *E. coli* DNA were inferior to the results of an HMM with a hand-crafted topology (see [KMH93]), although this may be because of the postprocessing they perform to cope with overlapping genes.

[2] See [HKP91] for a good book on neural nets.

1.3 The input to/output from the algorithms

The input to the algorithms will usually be a set or multi-set of finite-length strings. Some of the algorithms accept a single string of length m (assumed to be generated by a PFA with no final state), but they convert this to a multiset of $m - \ell + 1$ strings by sliding a window of length ℓ across it. If the strings are unlabelled, we assume they are positive examples of the target (stochastic) regular language we are trying to learn. Unfortunately, in the deterministic case, it is not possible to learn a language from positive data alone [Gol76], so often the strings will be labelled as either in the language (positive examples) or not in the language (negative examples). When trying to learn FSMs, rather than just presenting strings with their + or - labels, we give the corresponding output strings they should produce; this is sometimes called a (partial) behavioral specification.

In much of the literature, the goal has been to learn the right rule (i.e., the rule believed to be generating the examples) in the limit of infinite data. This notion is called *identification in the limit*, and was invented by Gold [Gol76]. In the case of DFAs, the "right rule" means that the inferred model should be as small as possible, and contain all the strings in the set of positive examples and none of the strings in the set of negative examples. If no negative examples are provided, the inferred language should contain as few strings as possible which are not in the positive example set.

In the case of PFAs, our goal might be to learn the target probability distribution with probability 1 in the limit of infinite data. However, in practice we only have a finite amount of data, so a more useful goal would be to approximate the target distribution to within some precision given only a finite training sample. We will define this more precisely later.

1.4 Batch vs. online algorithms

An online algorithm receives a new input string x_t (along with its label) at every time step t, and returns its best guess H_t so far about what the target automaton is. H_t should be computed only from H_{t-1} and x_t. Usually we impose the additional constraint that the size of H_t be a sublinear function of t, so that an algorithm which remembers all the strings so far, and each time constructs a new model "from scratch", is not considered on-line.

A batch algorithm, in contrast, waits until it has received the complete input set I, and then produces its best guess. Many batch algorithms can be converted to on-line algorithms by alternating between "growing" and "shrinking" phases: in the growth stage, new strings are added to the current model, and whenever it becomes too big, the state merging algorithm is invoked (see [RST95, SO94] for examples).

1.5 Bottom-up vs. top-down algorithms

Most of the batch algorithms we study work in roughly the same way, namely: construct some canonical, tree-shaped automaton which represents the input set (see Figure 2 for an example), and then merge equivalent states to get a smaller model. (We will explain the details later.) We call this a "bottom-up" approach. A "top-down" approach consists of starting with a 1-state model, and then trying to "fit" new strings to the model, and splitting it apart into new states/ transitions whenever necessary.

There is no consensus on which method is better. In the case of learning FSMs from an incomplete sample, we will study a top-down algorithm which runs in time exponential in n (the number of states in the final, minimal machine), and a bottom-up algorithm which runs in time exponential in t (the number of nodes in the tree). Experimental results [OE95] verify that the top-down approach is faster in practice. However, the exponential in t behavior arises because (1) the input is incomplete, and (2) we insist on an exact answer. Removing either of these conditions will yield a much faster (polynomial time) solution, as we will see.

Ron, Singer and Tishby [RST94] present a top-down algorithm for learning a PSA (a Probabilistic Suffix Automaton, which can be thought of as a variable-order Markov chain). They claim that their top-down scheme is equivalent to a bottom-up scheme, but that the top-down scheme is "somewhat more intuitive, simpler to implement, [and] more easily adapted to an online setting". However, Stolcke and Omohundro [SO94] say that, "our experience has been that modelling approaches based on splitting tend to fit the structure of a domain less well than those based on merging". Intuitively the reason for this is that top-down approaches have seen less of the data, and have to make decisions too early, whereas bottom-up approaches have more and more data to work with as states get merged.[3]

The most general approach would be to think of the problem as searching through model space, and allow both the splitting and merging operators. As far as we know, no one has tried this.

2 Finite automata defined

The model classes we shall consider are deterministic finite automata (DFAs) and probabilistic finite automata (PFAs). We shall define these more precisely, and then show how PFAs relate to Markov Chains and Hidden Markov Models (HMMs). DFAs are discussed more fully in [AU72, HU79] and PFAs in [Paz71]; for a good tutorial article on HMMs, see [Rab89].

2.1 Deterministic Finite Automata

Definition 1 *A* Deterministic Finite Automaton *(DFA) is a tuple $M = (\Sigma, Q, q_0, F, \delta)$, where Σ is the input alphabet, Q is the set of states, q_0 is the initial or start state, $F \subseteq Q$ is the set of final states, and $\delta : \Sigma \times Q \to Q$ is the transition function. A* Non-deterministic Finite Automaton *(NFA) is the same as a DFA except that $\delta \subseteq \Sigma \times Q \times Q$ is now a relation; intuitively, for each state q and input letter a, there may be several states that can be reached on the next step (hence the next state is* not fully determined*).*

The above definition of a DFA/NFA views it as an *acceptor* of strings. A string $x = x_1 x_2 \ldots x_m$ is said to be *accepted* by a DFA/NFA if there is at least one path edge-labelled by x which starts in the initial state and ends in a final state. (This definition covers the case of an NFA in which we may not know which edge to follow — conceptually, we are allowed to try them all, as long as at least one of them leads to a final state.) The set of strings accepted by a DFA/NFA M is called the language accepted by M, and is denoted $L(M)$. Any language accepted by a DFA/NFA is called *regular*. Equivalently, we can view DFAs/NFAs as *generating* a language, by performing all possible walks from the intial state to a final state; we then define $L(M)$ as the set of strings generated by M.

A final state can be viewed as emitting the "accept" symbol, and a non-final state as emitting a "reject" symbol. A natural generalization of this is to allow states to emit arbitrary symbols. This is called a *Moore machine*. Now, whenever we perform a walk to accept an input string x, we emit a series of symbols at each state we encounter to form an output string y. Thus the resulting machine acts as a *transducer*, transforming input strings into output strings. Alternatively, we can think of associating output symbols with each arc instead of each state; this is called a *Mealy machine*. Moore and Mealy machines are interconvertible. Transducers are also called *Finite State Machines* (FSMs).

[3] In a personal communication (11/15/95), Dana Ron pointed out that their top-down algorithm actually looks at the whole tree constructed so far, and thus has roughly the same amount of information available to it as a bottom-up algorithm.

2.2 Probabilistic Finite Automata

Unfortunately, there is is a lot of confusion about what exactly is meant by a PFA. Basically a PFA is a way of specifying a probability distribution over strings in a regular language.

It is important not to confuse probabilism with non-determinism. A non-deterministic machine is one in which the next state is not fully determined by the current state and the current input character. A probabilistic machine is one which has probabilities associated with it.[4] There are at least two natural ways of associating probabilities with a finite automaton. One is to take a DFA and associate a probabilistic emission function with each state; thus $\gamma(q, a)$ is the probability that state q emits symbol a. We call this a *Deterministic Probabilistic Finite Automaton* (DPFA), since, once the symbol is emitted, the next state is fully determined (hence we can equivalently think of the symbols as being attached to the arcs).

The second way to associate probabilities with a finite automaton is to attach probabilities to the arcs of an NFA. We shall call such machines *Non-deterministic Probabilistic Finite Automaton* (NPFA), since there is a choice of next states, even once we have seen the symbol. The transition relation δ is replaced by a series of $|\Sigma|$ transition matrices $T(\cdot)$, each of size $n \times n$, where n is the number of states. There are two possible interpretations of $T(a)[i,j]$: it could denote the joint probability of going from state i to state j and emitting symbol a, which we shall call an output-NPFA, or it could denote the probability of going from state i to state j given that the input symbol was a, which we shall call an input-NPFA. The former model is the one which is most similar to Hidden Markov Models, and is the one we will concentrate on. The latter model, however, is the original definition, and stems from the tradition of viewing automata as acceptors of strings. In the next subsection, we shall show that output-NPFAs are more "general" than input-NPFAs (although input-NPFAs can be generalized to probabilistic transducers in a way which output-NPFAs cannot).

2.2.1 Input-NPFAs vs. output-NPFAs

In an input-NPFA, we define $T(a)[i,j] = \Pr(q_j|q_i, a)$. Hence the transition matrices $T(a)$ are stochastic matrices, which means each row must sum to 1: $\sum_j T(a)[i,j] = 1$ for all i, a pairs. Hence the number of degrees of freedom is $(n-1) \times n \times |\Sigma|$. In an output-NPFA, the probabilities are joint probabilities over transitions and emissions, that is, we define $T(a)[i,j] = \Pr(a, q_j|q_i)$. We require that $\sum_j \sum_a T(a)[i,j] = 1$ for all i. Hence the number of degrees of freedom is $n(n|\Sigma|-1)$. Thus we see that an output-NPFA has more degrees of freedom than an input-NPFA; for example, for a binary alphabet, an output-NPFA has $2n^2 - n$ degrees of freedom, whereas an input-NPFA has only $2n^2 - 2n$. This is one sense in which an output-NPFA is more general than an input-NPFA.

We can convert an input-NPFA into an output-NPFA if we specify the distribution $\Pr(a|q)$, i.e., the probability of generating a string. This is an extra $n(|\Sigma|-1)$ degrees of freedom. (This distribution is implicit in the output-NPFA, since $\Pr(a|q) = \sum_{q'} \Pr(a, q'|q)$.) This is another sense in which an output-NPFA is more general than an input-NPFA.

Finally, in the next subsection we will see that output-NPFAs can be used both to generate random strings, and to evaluate the probability that a random string was generated by a given output-NPFA, whereas input-NPFAs can only be used to evaluate the probability of a string generated by some other process. This is the final sense in which output-NPFAs are more general. Since we are interested in passively learning the structure of an automaton which "spontaneously" generates strings, from now on we will shall only consider the case of output-NPFAs, and drop the prefix "output".

[4] However, in view of the fact that DFAs and NFAs are equivalent in power (i.e., accept the same set of languages), we often use the phrase "deterministic finite automaton" to mean "non-probabilistic finite automaton", which covers both DFAs and NFAs.

The original definition of a PFA, due to Rabin [Rab63], was in fact as an input-NPFA, and stems from the tradition of viewing automata as acceptors of strings. In Rabin's formulation, a string is said to be accepted if the probability of its being accepted exceeds some threshold $\theta \in [0, 1)$. By using the fact that the number of possible values for θ is uncountable, he showed that the class of languages accepted by PFAs is strictly greater than the class of languages accepted by DFAs; if, however, θ is restricted to being a rational number, PFAs are no more powerful than DFAs. Furthermore, any PFA which has an *isolated cut-point* can be converted to a DFA. A cut-point θ is called isolated with respect to a PFA if there exists a $\delta > 0$ such that $|\Pr(x) - \theta| \geq \delta$ for all $x \in \Sigma^*$. The intuition is that, by performing repeated trials to reduce the error, we can be sure that the string is definitely accepted or rejected, since all probabilities are bounded away from θ, and hence we can convert the PFA to a DFA.

2.2.2 Generators vs. evaluators

As shown in [KMR+94], when talking about a discrete probability distribution, it is important to distinguish between generators and evaluators. An *evaluator* for a probability distribution D (over fixed length strings, say Σ^n) takes as input a string x and returns its probability under D. A *generator* takes as input a string of random bits, and outputs a string that is distributed according to D. They show that some kinds of distributions have efficient (polynomial sized) generators but no efficient evaluators.

It is clear that a DPFA and an output-NPFA can be used to generate a string of length m in $O(m)$ time: for a DPFA, if we are in state q, choose an emission symbol a according to $\gamma(q, a)$ and move to the state determined by $\delta(q, a)$; for an NPFA, if we are in state q, choose an arc leading out of q, emit the symbol on that arc's label and move to the next state. To generate strings in some stochastic regular language, we perform the above walk starting in the initial state until we end up in a final state (which might emit a special final symbol). To generate strings of length m, we just do a walk of m steps (optionally followed by a transition to the final state). To generate infinite length strings, we perform a continuous walk. It is not clear how to use an input-NPFA as a generator, however.

We now show how to use these models as evaluators. The basic idea is to consider all paths edge-labelled by $x = x_1 x_2 \ldots x_m$, and for each path, compute the product of the probabilities on each edge. If the underlying graph is deterministic, there will only be one path, so this is easy: we can evaluate the probability of a string of length m $O(m)$ time. If the underlying graph is non-deterministic, things are a bit more complicated, because there may be many paths labelled by x. Nevertheless, we can do this efficiently, as we now explain.

The first method uses matrix multiplication. The probability of all paths from i to j which are labelled by x is given by $T(x)[i, j]$, where $T(x) = T(x_1)T(x_2)\cdots T(x_m)$. This takes $O(mn^3)$ time to compute. We can stipulate that the path must start in the initial state (with index 0, say) and end in a final state (with indices in the set F), and set $\Pr(x) = \sum_{f \in F} T(x)[0, f]$. Alternatively, we could allow any state to be the initial state with some probability $\pi(i)$, and not have any distinguished final states, and set $\Pr(x) = \sum_{i,j} \pi(i)T(x)[i, j]$.

Another way of writing the formula is as follows.

$$\Pr(x) = \sum_{q_0, q_1, q_2, \ldots, q_m} \pi(0) \Pr(q_1 | q_0, x_1) \Pr(q_2 | q_1, x_2) \cdots \Pr(q_m | q_{m-1}, x_m).$$

A naive implementation of this would take $O(mn^m)$ time, since there are n^m possible paths of length m, and m multiplications to perform at each step. However, we can use the following dynamic programming algorithm, called the forward-backward algorithm [Rab89], to compute it in $O(mn^2)$ time. We define $\alpha_t(i)$ as the probability of a walk from the start state to state i labelled by $x_1 \ldots x_t$: $\alpha_t(i) \stackrel{\text{def}}{=} \Pr(x_1 \ldots x_t, q_t = i)$. We can compute this inductively as follows. For the base case, we set $\alpha_0(i) = \pi(i)$, and for the induction step,

$$\alpha_t(j) = \sum_i \alpha_{t-1}(i)\,\Pr(q_i \xrightarrow{x_t} q_j).$$

Then we set $\Pr(x) = \sum_i \alpha_m(i)$.

The forward-backward algorithm can be used to evaluate the probability that an output-NPFA generated a string. If we imagine a process uniformly generating strings and sending them to an input-NPFA, then we can use the forward-backward algorithm to evaluate the probability which the automaton induces on its inputs.

2.3 Hidden Markov Models and Markov Chains

There are two definitions of a Hidden Markov Model (HMM). The most common one is when each state has a probability distribution $\Pr(a|q)$ over symbols which are emitted (hence the states are "hidden"), and another (independent) distribution $\Pr(q'|q)$ over transitions (hence the underlying topology is a Markov chain). This is clearly the same as an NPFA in which the transition and emission probabilities are conditionally independent, since $\Pr(a, q'|q) = \Pr(a|q) \cdot \Pr(q'|q)$.

The other, less common definition of an HMM associates probabilities with the *arcs*, that is, it specifies the joint distribution $\Pr(a, q'|q)$. This is exactly the same as our definition of an NPFA. The advantage of this formulation (besides removing the independence assumption) is that one can have silent arcs, that is, arcs that emit λ, the empty symbol. This can be useful for "skipping over" parts of a model, or for "looping back"; see [Rab89] for some examples in the context of speech recognition.

To see that HMMs cannot be converted into a DPFA, note that the transition probabilities are independent of the symbol, and hence the next state is not fully determined by the current state and the current input; alternatively, we can think of each transition being labelled by λ, but since there may be several such arcs, this results in non-determinism.

We now consider a subclass of DPFAs, called Probabilistic Suffix Automata (PSAs), and show how they can be used to succinctly model variable order Markov chains [RST93, RST94]. In a PSA, every state is labelled by a finite length string in Σ^*. If every label is at most length L, we call it an L-PSA. The label encodes how much "history" we pay attention to. (To ensure this condition, we require that, if there is an arc $p \xrightarrow{a} q$, then the label of q should be a suffix of $s \cdot a$, where s is the label of p, and $s \cdot a$ denotes string concatenation.) If the set of states is (labelled by) all of Σ^L, we have an order L Markov chain, since the next state transition probability depends on the last L symbols. However, if some states need a shorter memory length than L, a PSA is a much more compact representation than a Markov Chain, and hence easier to learn.

2.4 Fuzzy automata

We just mention in passing that using probabilities is not the only way to extend DFAs. Another method uses fuzzy logic — see [Mic90] for a few references.

3 Learning DFAs

3.1 Definition of success

Our goal will be to find the smallest automaton which is compatible with the input. In the limit of infinite data, this will be equivalent to the target concept we are trying to learn.

Since this is a difficult goal to achieve, often we will find it acceptable if, with high probability, we can learn a model which is approximately equal to the target concept. This notion, called PAC-learning (Probably Approximately Correct), is due to Valiant, and is discussed further in [Ang92, KV94]. The formal definition is given below.

Definition 2 *Let X be the problem instance space, $\mathcal{C}$ be a concept class over X (i.e., a set of subsets of X), and $\mathcal{H}$ be a hypothesis class over X (i.e., a set of representations for (at least) every function in $\mathcal{C}$). (Example: X is Σ^*, $\mathcal{C}$ is the set of regular languages, and $\mathcal{H}$ is the set of DFAs.) We say that $\mathcal{C}$ is* PAC learnable using *$\mathcal{H}$ if there exists an algorithm L with the following property: for every target concept $c \in \mathcal{C}$, for every distribution $\mathcal{D}$ on X, and for all error parameters $0 < \epsilon < 1/2$ and confidence parameters $0 < \delta < 1/2$, if L is given random examples of c drawn according to $\mathcal{D}$, then with probability at least $1 - \delta$, L outputs a hypothesis concept $h \in \mathcal{H}$ such that $\epsilon \geq error(h) = \Pr_{x \in \mathcal{D}}[c(x) \neq h(x)]$. If L runs in time polynomial in $1/\epsilon$, $1/\delta$, n (the dimensionality of the instance space), and size(c) (the size of the concept under some fixed representation scheme), we say that $\mathcal{C}$ is* efficiently *PAC learnable.*

3.2 Complexity results

A *complete presentation* of a language L is an ordered sequence of all the strings in Σ^* labelled as either positive (being in L) or negative (not in L). A positive presentation just consists of all the strings in L. We assume L could be an infinite set. Gold [Gol76] showed that it is not possible to exactly identify L, even in the limit of infinite data, given only a positive presentation. This seems to be a paradox in view of the fact that humans seem to learn languages from only positive examples. However, we will see later that it is possible to identify in the limit with probability 1 a stochastic regular language from only positive examples.

In practice one only ever sees a finite sample. A *uniform complete sample* is the set of all labelled strings of length at most m. We will see later that there is a simple algorithm, which runs in time polynomial in the number of nodes in the input tree, and which can learn the minimal FSM (i.e., the one with the least number of states) consistent with such an input set. If the sample isn't uniform complete, and is missing even an arbitrarily small fixed fraction $0 < \epsilon < 1$ (i.e., the input contains only $(|\Sigma^n|)^\epsilon$ strings) then Angluin [Ang78] showed that the problem is NP-hard. Obviously, therefore, the case of learning the smallest DFA consistent with an arbitrary set of positive and negative examples is also NP-hard [Gol78, Ang78]. Even learning near-minimal DFAs, or PAC-learning DFAs, is hard: Pitt showed that learning a near-minimal DFA (i.e., one with n^k states, where n is the minimal number of states, for some fixed k) is NP-hard [PW93], and Kearns and Valiant [KV89] showed that PAC-learning a DFA with any reasonable hypothesis class is as hard as breaking various cryptographic protocols which are based on factoring.

However, these are only worst case results, and on average, we will see that we can do quite well, even with sparse input sets.

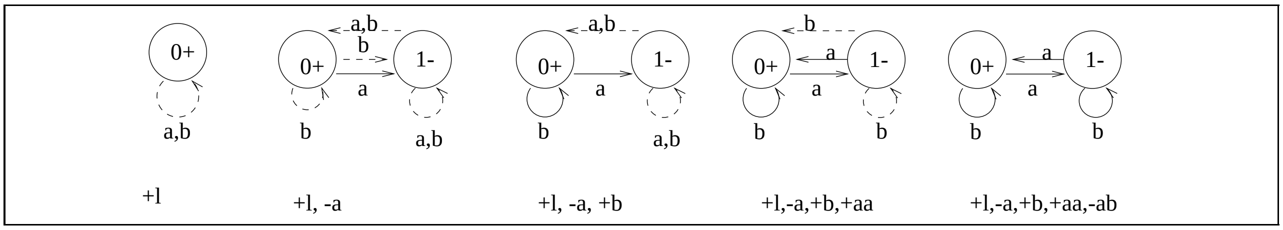

Figure 1: The best hypotheses produced by the Porat and Feldman iterative algorithm as a function of sample size. The language is the "even a's" language (i.e., the input set is $+\lambda, -a, +b, +aa, -ab, \ldots$), and the rightmost DFA is the correct minimal automaton. Dotted lines denote mutable links (which may be subsequently deleted), solid lines denote permanent links. l denotes λ, the empty string. 0 is an accept state, 1 is a reject state.

3.3 Top-down algorithms

3.3.1 Exactly learning DFAs using an ordered complete presentation

Porat and Feldman [PF91] give an online algorithm for incrementally learning DFAs, i.e., at each step, they output a DFA which is the best guess so far (in the sense of being the smallest model consistent with the data seen so far), based only on the previous guess and the current input string. Hence the storage space required is polynomial in n, the number of states in the minimal automaton. (By contrast, batch algorithms store all the examples strings, and hence could take space exponential in the length of the longest string if they are given a complete presentation.) The algorithm introduces a new state, complete with all possible arcs to and from it, if the current example cannot be fit to the model, and arcs are pruned by subsequent counterexamples. A worked example is shown in Figure 1. The algorithm takes $O(n^2)$ time and space.

Porat and Feldman require that the input be a complete, lexicographically ordered presentation of the language (i.e., λ, a, b, aa, ab, ..., where each string is marked as either + or -, and we have assumed $\Sigma = \{a, b\}$), since they prove that no algorithm which uses a fixed amount of working storage (in addition to the space required to store the current guess and example) can learn a DFA from an arbitrarily ordered presentation. The inspiration for this was the fact that connectionist (neural network) systems use only finite working space. Indeed, they show that they can implement their algorithm in a neural network. (We will briefly discuss other neural network methods later.)

3.3.2 Exactly learning FSMs from an incomplete specification

Oliveira and Edwards [OE95] deal with the more general case of when the input consists of an arbitrary set of consistent input/output pairs. This case arises in practice when synthesising FSMs from a partial behavioral specification. The specification can be given in the form of an incomplete labelled prefix tree (see Figure 2 for an example). (Hence this algorithm is a batch algorithm.) However, rather than adopting the bottom-up approach of trying to merge states in this tree, which takes time exponential in t (the number of nodes in the tree)[5], they adopt a top-down approach of walking over the specification tree, and growing the current model whenever they encounter some transition which cannot be fit into the current model. This runs in time which is exponential in n, the number of states in the final minimal automaton. (This algorithm is similar to the one by Biermann *et al.* [BK76]).

[5] Exponential time may be necessary because we may need to consider all possible subsets of nodes, to find which ones should be merged. [Pfl73] proved the general problem of minimizing an FSM from a partial specification is NP-complete.

In more detail, the algorithm works as follows. It tries to build a deterministic Moore or Mealy machine with no more than n states which is compatible with the input seen so far. If no machine is found, one more state is allowed, and we continue the search. (A lower bound on the number of states can be determined by computing the size of the largest clique in the incompatibility graph, in which the vertices are the nodes in the behavior specification, and the edges join nodes which cannot be merged because they have incompatible behavior. Unforunately, max-clique is an NP-complete problem, so this may take exponential time.)

Suppose we already have a partially constructed machine M and we are in state q. We then try to fit the next labelled transition $q \xrightarrow{x/y} q'$ from the prefix tree into M. ($q \xrightarrow{x/y} q'$ denotes an arc from q to q' which converts the input string x into the output string y.) There are two possibilities:

- There is no such transition currently in M. The algorithm considers all possible ways of adding such a transition, that is, all possible destination states. It makes a choice, and then recursively tries to satisfy the rest of the specification. If this works, the choice was correct and the algorithm terminates successfully. Otherwise, it backtracks and considers the next untried destination. It also considers the possibility that the destination might be a new state.

- There already is a corresponding transition $q \xrightarrow{x/y'} q'$. If the outputs agree ($y = y'$), we can use this transition, otherwise a previous decision must have been incorrect, so we backtrack.

They also give an implicit version of their algorithm, which considers all possible mappings from nodes in the prefix tree to states, and "filters out" those that are inconsistent with the data. They use a package called the Multi-valued Decision Diagram (MDD) method to implicitly manipulate the boolean functions which enforce the output compatibility constraint (the fact that, if there are two arcs $q \xrightarrow{x/y} q'$ and $q \xrightarrow{x/z} q'$, then y must equal z) and the transition compatibility constraint (the fact that the automaton is deterministic). They show that the performance of this algorithm is comparable to their explicit enumeration method.

3.3.3 Enumerating all compatible automata of a given size

Gaines [Gai76] presents an algorithm which produces an exhaustive enumeration of all the n-state Moore machines which are compatible with the input. If n is too small, the Moore machine may be non-deterministic. Each string is then "sent through" the machine, and the number of times each arc is transitioned is counted. In this way, the transition probabilities can be estimated so as to minimize some error measure. Hence the machine can also be probabilistic.

The algorithm returns what Gaines calls an admissible subspace of solutions, ordered by the two partial orders of simplicity (namely number of states) and goodness-of-fit. It is then up to the user to make the appropriate tradeoff. (We will study the Bayesian approach to this later.)

One interesting aspect of his algorithm is that it takes a single string as input. However, the algorithm might be told that it contains special delimiter (end of string) symbols (and thus represents a set of examples), and/or that, say, every odd symbol is an input and every even symbol an output. Naturally the algorithm runs faster with this "side information", but in principle it can infer it, by noticing, for example, that every time a certain symbol is read, there is a transition back to the initial state (hence that symbol represents the end of a string), or that every odd symbol is completely unpredictable (because it represents a random input), but that every even symbol is predictable given its predecessor.

3.4 Bottom-up algorithms: minimizing canonical automata

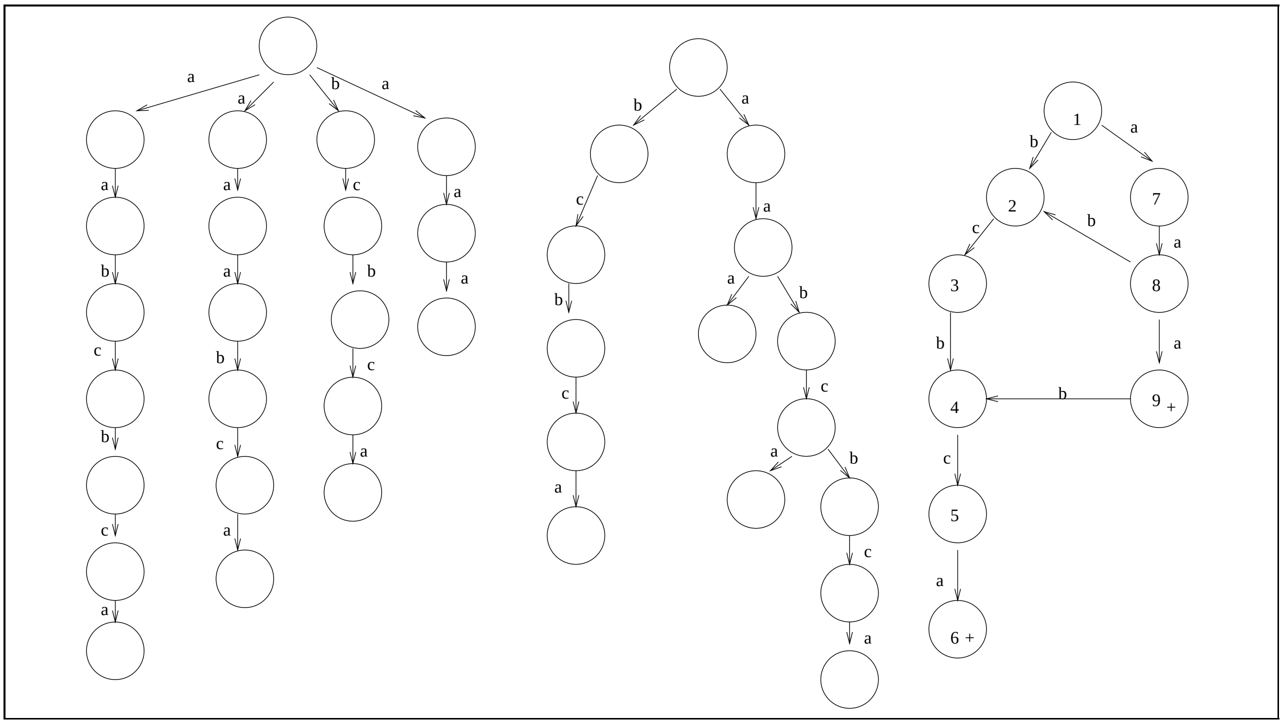

Figure 2: Canonical automata for the set of positive examples {bcbca, aaabca, aabcbca, aaa}. All states are rejecting except for the leaves, and, in the case of the rightmost automaton, the internal states labelled as +. If we want to represent an FSM, each arc can be labelled by an input/output pair; this is sometimes called a behavioral specification. Left: the maximal canonical automaton (which is non-deterministic). Middle: the prefix tree canonical automaton. (A prefix tree is also called a trie [AHU83].) Right: the DAG (Directed Acyclic Graph) canonical automaton.

We can always construct a canonical automaton M compatible with any given finite input set I, as illustrated in Figure 2. (An automaton M is called canonical if $L(M) = I$.) What we would like is to find the smallest automaton compatible with the input set. This can be achieved by minimizing the canonical automaton, by merging equivalent states (nodes in the graph).

There is a simple algorithm for minimizing a *given* DFA with n states in $O(n^2)$ time and a slightly more complex one which runs in $O(n \log n)$ time [AU72, Hop71]. This algorithm uses an equivalence relation to partition the states into equivalence classes. The relation is $q \equiv q'$ iff $T(q) = T(q')$, where $T(q) = \{x : \delta(q, x) \in F\}$ is the tail of q, that is, the set of strings accepted from state q. We will explain this method later.

If the input set consists only of positive examples, then any automaton M' derived from the canonical automaton M by merging states will be compatible with the input set, since $L(M') \supseteq L(M) = I$. For example, we could merge all the states and return the one-state automaton with $L(M') = \Sigma^*$. There are many different algorithms for this problem because they each embody a different tradeoff between model size and over-generalisation (as measured, for example, by $|L(M') - L(M)|$). When learning PFAs, we only deal with positive examples, but in that case, we are interested in learning a probability distribution, and there are well-defined notions of how good our answer is.

Unfortunately, when the input consists of labelled examples, we cannot apply these standard minimization algorithms, since the input is no longer a DFA; rather, it is an incomplete specification of the behavior of a DFA or FSM. It is incomplete because some arcs are missing. In the case of an FSM, we do not know what output they should produce, and in the case of a DFA, we do not know whether the nodes they lead to should be accepting or rejecting. Intuitively we must consider all combinations of ways of labelling these "don't know" nodes, which leads to exponential time behavior. Indeed, minimizing an incompletely specified FSM is NP-complete [Pfl73]. Nevertheless, it is an important problem in the synthesis of sequential logic circuits, so fast heuristic algorithms have been developed. There are basically two kinds, based on an explicit [HRSJ91] or implicit [KVBSV94] enumeration of the compatibles. (A compatible is a set of states equivalent in the sense that they can be merged without affecting the behavior of the machine. The number of compatibles may be exponential in the number of states.) In practice, however, it seems that top-down strategies work better than minimization strategies, at least for synthesising FSMs designed by human engineers, which may be highly structured [OE95].

A special case is if we know that the machine which generated the data has only n states, and we are given a uniform complete sample which contains all strings of length at most $2n - 1$ and their labels; then it is possible to reconstruct the original machine, as we will see next. (Of course, the set of all strings of length at most $2n - 1$ is exponentially large!)

3.4.1 Learning FSMs from a uniform complete sample: the Russian algorithm

We now describe an algorithm due to Trakhtenbrot and Barzdin' [TB73] which forms the basis of much of our subsequent discussion. We will call this the "Russian algorithm" for convenience. The input is a complete labelled prefix tree of height $2n - 1$. The algorithm has two phases. In the first, it makes a pass over the tree in breadth-first (lexicographic) order to determine the states Q; that is, it computes a mapping from nodes in the tree to states in the machine. In the second, it finds the node corresponding to each state and adds the out-going transitions from this state. The states are the equivalence classes induced by $\equiv$, which will be defined shortly. Let $[n_i]$ denote the equivalence class containing node n_i. For each state (equivalence class) $[n_i]$, we need to define one unique representative node; this can be any node in that class, but for concreteness, we will take the first node added to that class, and denote it n_i.

Input: a complete, labelled prefix tree T of height $2n - 1$

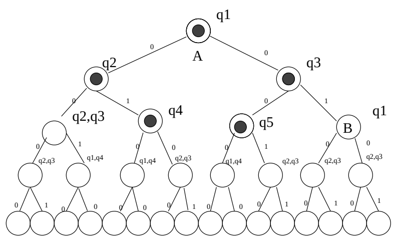

Figure 3: A uniform complete tree. This can be regarded as specifying the input/output behavior of an FSM. For example, the input 1111 gets transformed to 0101 (follow the path down the right edge of the tree). The nodes with a black circle inside them are the states in the final minimal machine. All the other nodes are labelled with the states they are indistinguishable from, because they have the same subtrees (where defined). A and B are used to identify nodes discussed in the text. From Figure 15 of [TB73].

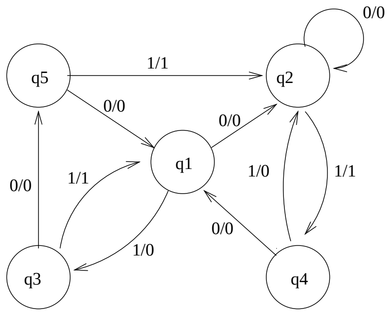

Figure 4: The minimal FSM realizing the tree in Figure 3. Taken from Figure 15 of [TB73]. The initial state is q_1.

Output: the smallest FSM M consistent with T
Algorithm:
$Q := \emptyset$
for each node n_i in T in breadth-first order
    if $\exists [n_j] \in Q$ s.t. $n_j \equiv n_i$
    then add n_i to $[n_j]$
    else Q := Q $\cup [n_i]$ /* make a new state */
for each $[n_i] \in Q$
    for each edge $n_i \xrightarrow{a} n_j$ in T
        add the transition $[n_i] \xrightarrow{a} [n_j]$ to M

Two nodes n_i, n_j are equivalent (denoted $n_i \equiv n_j$) if their tails, where defined, are equivalent; the tail of a node is just the labelled subtree below it. A node q specifies the behavior of the machine starting in state q for the strings which appear in its tail. For example, node A specifies that 1111 should be transformed to 0101, which we denote $A(1111) = 0101$. Nodes A and B are equivalent because they specify the same behavior for the strings on which they are both defined, e.g., B(11) = 01 and A(11) = 01. In this case, there are five distinct states, and the resulting minimal machine is shown in Figure 4. It is possible that in the second phase, the child n_j will belong to several equivalence classes (which is why some nodes in Figure 3 have several labels), in which case we can produce a family of equivalent deterministic FSMs.

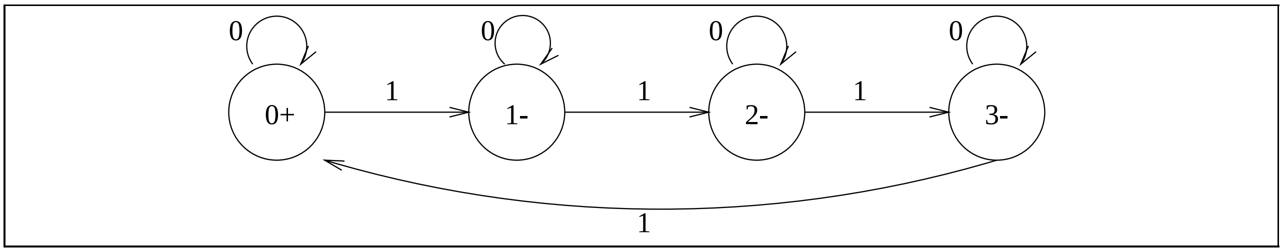

Figure 5: An example FSM with the greatest possible degrees of accessibility and distinguishability. + denotes an accept state, - a reject state. Here, the machine accepts all strings $x \in \{0,1\}^+$ such that the number of 1s in x is a multiple of 4. 0 is the start state. It is clear that we need a string of length 3 to get to state 3. Also, to distinguish state 0 from state 1, we need a string of length 3 (since state 0 maps 111 to a reject state, and state 1 maps 111 to an accept state). If we allow the empty string as an input (which does not make sense for Mealy machines), then the degree of distinguishability is $d \leq n-2 = 2$ [AU72, p.125].

3.4.2 The complexity of the Russian algorithm

When comparing two nodes for equivalence, it isn't always necessary to compare the whole of their subtrees, as we now show. We say two states q, q' are k-distinguishable if there exists a word x of length at most k, such that there is a path edge-labelled by x, starting in q, which ends in an accept state and another path edge-labelled by x, starting in q', which ends in a reject state, or vice versa. Otherwise the two states are called k-indistinguishable or k-equivalent, denoted $q \equiv_k q'$. (In the case of FSMs, we require that the two paths transform x into the same output string, rather than requiring that they both label it as either acceptable or not.) Two states are indistinguishable (equivalent) if they are k-indistinguishable for all k. In terms of the tree, two nodes are k-equivalent if the subtrees of height k below them are the same. The subtree of height k below a node is also called its k-tail or k-signature. Indistinguishability is an equivalence relation which partitions the states into indistinguishability classes.

The degree of distinguishability of a (minimal) FSM, $d(M)$, is the minimal k such that any two states are k-distinguishable. The degree of accessibility of an FSM, $a(M)$, is the minimal k such that any state is accessible from q_0 by a word of length at most k. Trakhtenbrot and Barzdin' show that, to reconstruct M, we need a uniform complete sample of size at most Σ^{d+a+1}, that is, a complete tree of height $h = d + a + 1$. (h is called the degree of reconstructibility.) This makes good intuitive sense: we need strings of length a just to access the furthest states from q_0; we then need an additional d characters to see if their children are distinguishable. They also prove the following theorem.

Theorem 1 *Consider a minimal FSM with n states, input alphabet Σ and output alphabet Ω. Then its degree of distinguishability is bounded by*

$$\log_{|\Sigma|} \log_{|\Omega|} n - 1 \leq d \leq n - 1$$

and its degree of accessibility is bounded by

$$\log_{|\Sigma|} n - 1 \leq a \leq n - 1$$

Furthermore, these bounds are the best possible.

Though we shall not prove the theorem, the following example shows that the upper bound is tight: consider an FSM which accepts strings x iff the number of ones in $x = x_1 x_2 \dots x_m$ is a multiple of n. See Figure 5 for an example. The proof that the lower bounds are the best possible is somewhat trickier, and is omitted.

We are now in a position to state the complexity of the Russian algorithm. Let us define $N(\Sigma, h)$ to be the number of nodes in a complete $|\Sigma|$-ary tree of height h. Clearly

$$N(\Sigma, h) = |\Sigma|^0 + \cdots + |\Sigma|^h = \frac{|\Sigma|^{h+1} - 1}{|\Sigma| - 1}$$

The worst case sample complexity (i.e., the number of strings in the input set) is therefore $N(\Sigma, 2n-1)$, which is approximately 2^{2n} when $|\Sigma| = 2$. The worst case time complexity can be computed as follows. Clearly phase one is the slowest phase, and it makes up to $1 + \cdots + (t-1) = O(t^2)$ equivalence comparisons, where $t = N(\Sigma, 2n-1)$ is the number of nodes in the tree. Also, each equivalence check may need to check up to $N(\Sigma, n-1) \approx t^{0.5}$ nodes, so the total time is $O(t^{2.5})$. This is approximately 2^{5n} for binary alphabets.

The worst case complexity of the Russian algorithm is very high. Can we do better on the average? In fact we can. To define the *expected* values of d and a, we have to specify what it is that we are randomising (and hence what we are averaging over). The "maximally random" case, when both the topology and the labelling of the states (as either accept or reject) are randomly chosen, is studied by Trahktenbrot and Barzdin'. More recently, Freund et al. [FKR+93] have studied the case where the topology may be adversarially chosen, and only the labelling is random; such DFAs are called *typical*.

Trahktenbrot and Barzdin' show that, in the maximally random case, "almost all" automata have degrees of distinguishability and accessibility which are close to the lower bound. More precisely, $E[d] = \log_{|\Sigma|} \log_{|\Omega|} n$ and $E[a] = C \log_{|\Sigma|} n$, where C is a constant. Lang [Lan92] found experimentally (for $\Sigma = \Omega = \{0, 1\}$) that for a large fraction of randomly generated DFAs (with several thousand states), $a \leq 2\log_2 n - 2$ and $d \leq 4$. He therefore made the approximation that a uniform complete sample must be of size $(2^{2\log_2 n - 2 + 4 + 1 + 1} - 1)/(2-1) = 16n^2 - 1$. Contrary to Angluin's result [Ang78], he found that it was possible to (approximately) learn these random DFAs quite well even if the training set only contained about 3% of the uniform complete sample.

3.4.3 Learning random DFAs from sparse samples: the Greedy Russian algorithm

We briefly present Lang's algorithm [Lan92], which is a variation of the Russian algorithm. Essentially he combines the two phases of looking for states and then computing the transitions into one single pass over the tree. When a node n_i is found to be equivalent to a previous node n_j, it means they have the same subtree, so we can make the parent of n_i point to n_j instead, and discard n_i and its children since they are now inaccessible. (Hence the tree is transformed into a non-tree-structured graph on the way down.) An upper bound on the running time is $O(tn^2)$, where t is the number of nodes in the tree and n is the number of states in the final, minimal machine. To see this, note that each state must be compared for compatibility with every other state, and each comparison may involve checking up to t nodes (walking the whole tree).

Lang proposes the following extension of this algorithm to deal with the case where the input tree is not complete. In this case, the absence of a labelling conflict between the subtrees rooted at n_i and n_j does not guarantee that the two nodes correspond to the same state in the smallest consistent machine. However, we can be greedy and merge the nodes anyway. This requires "overlaying" the subtree below n_i onto the subtree below n_j, before it is discarded. This involves making the parent of n_i point to n_j, as before, and recursively merging each of the children of n_i with the corresponding children of n_j, to remove any non-determinism. However, this process might produce a conflict, because the subgraph can contain cycles and other non-tree features. See Figure 6 for an example of such a conflict. Thus one cannot pre-check the legality of a potential

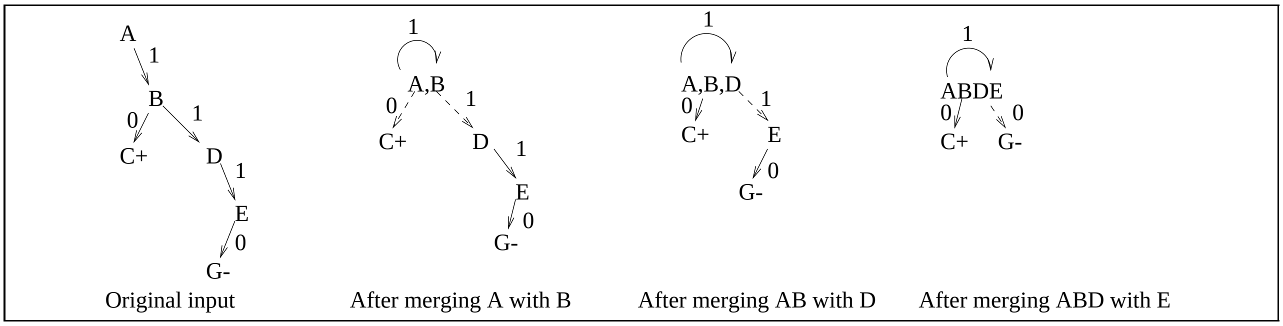

Figure 6: The prefix tree for the training set 10 $\rightarrow$ accept, 1110 $\rightarrow$ reject. + denotes accept states, - denotes reject states, other states have unknown polarity. The first step in Lang's algorithm is to try merging states A and B. Walks at A and B don't reach C and G simultaneously, so no conflict is detected. However, when A and B are combined, D and E get merged in as well, which causes C to be merged with G, which is a conflict. (Lang, 9/7/95, personal communication.)

merge by a simple walk starting at the two nodes. Lang's solution is to "optimistically perform a destructive merger of the subtree and subgraph while keeping backup copies of modified nodes so that the work can be cheaply undone if a labelling conflict is detected."

Later we will study several algorithms for learning PFAs, which are very similar to Lang's and involve merging equivalent states. However, the input only consists of positive examples, so these labelling conflict complications do not arise. In fact, we cannot learn a DFA from positive data only; it is the repeated presentation of strings (with frequency proportional to their probability) which compensates for the lack of negative examples in the PFA case.

3.4.4 Learning typical DFAs from a random walk

A "typical" automaton is one in which the states are randomly labelled as accept or reject, but the topology may be arbitrary (possibly adversarially chosen). Freund et al. [FKR+93] extend the definitions and theorems of the somewhat less random case considered by Trakhtenbrot and Barzdin' as follows.

Definition 3 *We say that* uniformly almost all *automata have property $P_{n,\delta}$ if the following holds: for all $\delta > 0$ (where δ is a confidence parameter), for all $n > 0$ (where n is the number of states), and for any underlying automaton graph G_M on n nodes, if we randomly choose $\{+, -\}$ labels for the states, then with probability at least $1 - \delta$, $P_{n,\delta}$ holds for the resulting automaton M. Such an automaton is called* typical. *If the topology is also randomly chosen, we drop the word "uniform" (since the property P might not be "spread" uniformly through the space of all automata with n states).*[6]

Theorem 2 *For uniformly almost all automata (with binary input and output alphabets), the degree of distinguishability is at most $2\log(n^2/\delta)$, where δ is a confidence parameter (this is "one logarithm worse" than the bound for the non-uniform case). Hence for $d \geq 2\log(n^2/\delta)$ uniformly almost all automata have the property that the d-signature is unique, where the d-signature of a state is its depth d subtree.*

Because the signature is so short, we can hope to construct the signature of each state just by performing a random walk on the underlying graph. In particular, Freund et al. propose that a "robot" receive a random

[6] Trahktenbrot and Barzdin' define the phrase "uniformly almost all" in an asymptotic way, namely: uniformly almost all automata have property P if $\delta \rightarrow 0$ as $n \rightarrow \infty$.

input bit at each step, and then predict the output of the current state as either + (accept) or − (reject). It is also allowed to predict ?, called a *default mistake*, after which it will observe the correct output, and then return to the start state. (They also give a more complicated algorithm in which ? does not return the robot to the start state; this has slightly poorer performance.) Since each signature is unique, the robot can recognize when it returns to a state, and hence can fill in the transition data. They prove that, for uniformly almost all n state automata, their algorithm is efficient (runs in time polynomial in n and $1/\delta$), makes no prediction mistakes, and makes an expected number of default mistakes that is at most $O((n^5/\delta^2)\log(n/\delta))$. This algorithm can be extended to learn a (restricted kind of) DPFA, by counting the number of times each transition is made.

3.4.5 Approximately learning DFAs with the k-tails algorithm

Recall that two states q, q' are k-equivalent if they are not distinguishable by any string x such that $|x| \leq k$. In other words, they have the same k-tails. Clearly

$$q \equiv_{k+1} q' \Leftrightarrow \begin{cases} q \equiv_k q' \text{ and} \\ \forall a \in \Sigma : \delta(q,a) \equiv_k \delta(q',a) \end{cases}$$

Also, it can be shown [HU79] that any partition of the states induced by $\equiv$ will lead to the minimal DFA. This suggests the following iterative strategy for minimizing a DFA, known as Moore's algorithm [AU72].

```
k := 0
compute ≡_0
while k ≤ n − 2 and ≡_k ≠ ≡_{k−1} do
        k := k + 1
        compute ≡_k in terms of ≡_{k−1}
merge the k-equivalent states
```

This can be implemented to run in $O(|\Sigma|n^2)$ time as follows [HU79]. Construct an $n \times n$ array and mark cell (i,j) (and (j,i)) as soon as q_i and q_j are found to be distinguishable. Start by marking (i,j) for all $q_i \in F, q_j \notin F$. Then, for each pair of distinct states p, q, if, for some a, their children $r = \delta(p,a)$ and $s = \delta(q,a)$ are distinguishable by some string x, then p and q are distinguishable by ax, so mark cell (p,q) and dependents. Otherwise, (p,q) is placed on a list of dependents associated with the (r,s) entry, and if (r,s) subsequently receives a mark, so will (p,q). Since each pair (p,q) is considered at most $|\Sigma|$ times (it could be on up to $|\Sigma|$ lists), the running time is $O(|\Sigma|n^2)$.

As Miclet [Mic90] points out, the k-tails algorithm of Biermann and Feldman [BF72] can be thought of as running Moore's algorithm but terminating at a particular value of k. If we stop at a low value of k, the resulting machine will be small but highly non-deterministic (and therefore not equivalent to the target DFA). If we stop at a value of k which is greater than or equal to $n-1$, then we will return the minimal machine which accepts the input set, namely the DAG canonical automaton.[7] See Figure 7 for a worked example.

3.4.6 The k-tails clustering algorithm

In the k-tails algorithm, we defined two states to be k-equivalent if their k-tails were equal. We can consider other measures of similarity between k-tails, and then use a clustering algorithm to group the k-tails (now

[7] Interestingly, Biermann and Feldman independently arrive at the result of Trakhtenbrot and Barzdin' that to uniquely reconstruct the FSM which generated the input set, we need a uniform complete sample of all strings of length at most $2n-1$.

q	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$
1	$\emptyset$	$\emptyset$	$\emptyset$	aaa	aaa
2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	cbca
3	$\emptyset$	$\emptyset$	$\emptyset$	bca	bca
4	$\emptyset$	$\emptyset$	ca	ca	ca
5	$\emptyset$	a	a	a	a
6	λ	λ	λ	λ	λ
7	$\emptyset$	$\emptyset$	aa	aa	aa
8	$\emptyset$	a	a	a	a, abca
9	λ	λ	λ	λ, bca	λ, bca

Table 2: The k-tails of each state of the DAG DFA in Figure 2 as a function of k. λ denotes the empty string.

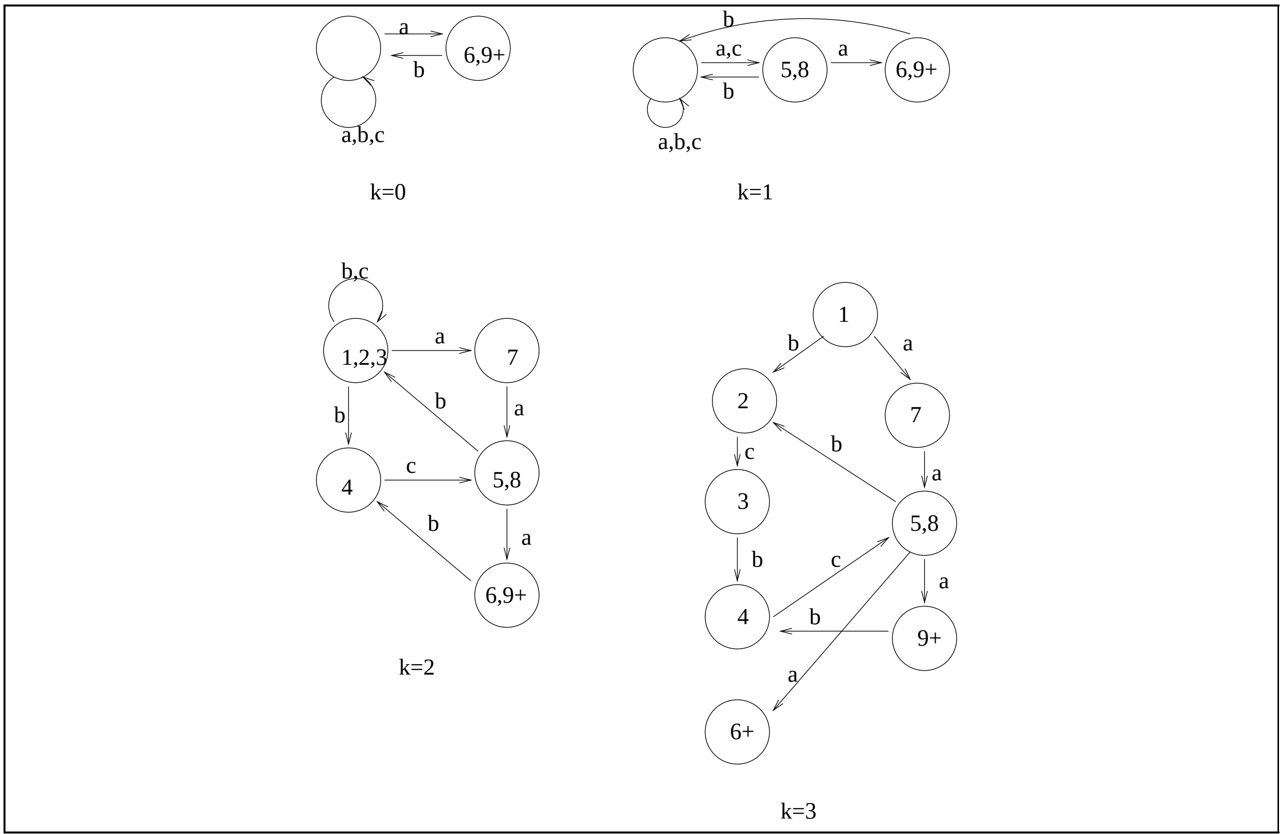

Figure 7: The DFA returned by the k-tails algorithm as a function of k, for the input set { bcbca, aaabca, aabcbca, aaa}. For $k=4$ the algorithm returns the DAG canonical automaton in Figure 2. See Table 2 for the details of the computation.

viewed as "points" in a space of dimension Σ^k). Since we can always reconstruct a DFA given its set of k-tails (for $k \geq n-2$), in some sense each state is uniquely identified by its k-tail. Hence, if two k-tails are similar (in the same cluster), their corresponding states should be merged. Miclet [Mic80] calls this the k-tails clustering algorithm. (Note that clusters are not the same as blocks in a partition, since we are not claiming that two states in the same cluster are *equivalent*, just that they have similar k-tails.)

We have to specify how the cluster will change when we add a new point to it. To save time, Miclet makes the approximation $k(q \cup q') = k(q) \cup k(q')$ (where $k(q)$ is the k-tail of q), whereas the exact relation is $k(q \cup q') \supseteq k(q) \cup k(q')$ (since when you merge states, more strings become acceptable).

We also have to specify the distance measure between two clusters. He proposes two rather *ad hoc* measures of distance between k-tails, a "global" one

$$d_1(k(q), k(q')) \stackrel{\text{def}}{=} \min\{|k(q)| - |k(q) \cap k(q')|, |k(q')| - |k(q) \cap k(q')|\}$$

which measures the number of strings they do not have in common, and a more "local" one based on the edit distance $d_s(x, x')$ between two strings

$$d_2(k(q), k(q')) \stackrel{\text{def}}{=} \max_{x \in k(q)} \sum_{x' \in k(q')} d_s(x, x')$$

which measures the worst-case total number of changes needed to convert a string x in one k-tail to every other string x' in the other k-tail.

He then uses a hierarchical (bottom-up) clustering algorithm[8], which repeatedly looks for the nearest two points and merges them, until no two points (k-tails) have any common strings. He starts with $k = t - 2$, where t is the number of nodes in the prefix tree, and decrements it at each step. This proves to be more effective than the k-tails algorithm on test cases consisting of small, hand-crafted DFAs.

3.5 Other methods

3.5.1 Actively learning DFAs with oracles

In view of the hardness results on passively learning DFAs with arbitrary input, a number of people have proposed equipping the learner with an *oracle*, which the learner can ask questions. This is called active learning, or learning with queries; for a good account of actively learning DFAs, see [KV94]. Angluin [Ang87] showed that the learner must be able to ask both membership queries (is x in the language?[9]) *and* equivalence queries (is the hypothesis M' equal to the target concept M, and if not, please give me a counterexample) in order to exactly learn DFAs in time polynomial in n and the length of the largest counter example. We can get by without the equivalence queries, and just use random examples and membership queries, if we are content to PAC-learn the DFA.

In the oracle membership model, we assumed that the target DFA returns to the reset state before answering whether x is in the language. If this is not the case — the so-called no-reset model — learning becomes much harder, because the learner starts in an unknown state and ends up in an unknown state. However,

[8] For a good review of clustering algorithms, see [Seb84, ch. 7]

[9] Asking whether a string is in the language is like performing an experiment in which the string is "fed into" the black box, and we observe whether it is accepted or rejected. In the more general case of inferring the structure of an FSM from a series of experiments, see [Con71]. (See also Exercise 3.22 in [HU79].)

Rivest and Schapire [RS89] show how the DFA can still be learnt in this case, by using *homing* sequences. Intuitively, a homing sequence is an input sequence which, when executed, may allow the learner to determine "where it is" in the machine, based on the observed output sequence. Freund et al. [FKR+93] show how to extend this method to the case of passively learning typical DFAs from a random walk, (i.e., learning random DFAs without an oracle) in the no-reset case. Rivest and Schapire need an equivalence oracle for their homing sequence method, whereas Ron and Rubinfeld [RR95] do not; however, their algorithm only works for DFAs with small cover time[10], which intuitively means that all the states in the DFA are easy to get to (Angluin's proof relies on states which are hard to reach).

3.5.2 Neural network methods

Minsky [Min67] proved that "every FSM is equivalent to, and can be simulated by, some [recurrent] neural network". However, this does not imply that it is easy to learn the structure of an unknown FSM using a neural network (NN). The main problem is that FSMs are discrete (they define operators over discrete valued alphabets, and have a discrete set of internal states), whereas NNs are continuous (they define functions over real-valued vectors, and have real-valued internal states). Consequently, most early attempts to learn DFAs with recurrent NNs (both first and second order) from positive and negative examples were not very successful: they took a long time to run, and couldn't even learn machines with as few as 5 states. More recently, Das and Mozer [DM93] have suggested clustering points in the hidden layer's state space before passing it on to other layers, to encourage the formation of discrete representations during training. Unfortunately, since they use a different test set from the other algorithms mentioned in this section, it is hard to compare empirical results, and no theoretical results exist.

4 Learning DPFAs

4.1 Definition of success

4.1.1 Goodness-of-fit to the data

As we mentioned earlier, the goal of learning PFAs is to approximate the target distribution to within some precision given only a finite training sample. The definition of acceptable precision is captured in the following definition, which comes originally from [KMR+94], and which has been used subsequently in [RST94, RST95]. It is inspired by the definition of PAC-learning a concept, given earlier.

Definition 4 *Let M be the target PFA we are trying to learn, and $\hat{M}$ be a hypothesis PFA. Let P^M and $P^{\hat{M}}$ be the two probability distributions they generate on Σ^*, respectively. We say that $\hat{M}$ is an ϵ-*good hypothesis *with respect to M, for $\epsilon \geq 0$, if*

$$D(P^M, P^{\hat{M}}) \leq \epsilon$$

where $D(P^M, P^{\hat{M}})$ is some measure of distance between these two distributions. An algorithm is said to efficiently learn a PFA $\mathcal{M}$ if, for any $\mathcal{M}, \epsilon, \delta > 0$ and random sample, it outputs an ϵ-good hypothesis with probability at least $1-\delta$, and runs in time polynomial in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$, $|\Sigma|$ and n, where n is the number of states in the target PFA.

[10] The cover time of a DFA M is defined to be the smallest integer t such that for every state of q in M, a random walk of length t starting from q visits every state in M with probability at least $1/2$.

Several measures of distance between two probability distributions (with set of support $\mathcal{X}$) have been proposed, including the χ^2 distance, the L_1 distance, the variation distance, the quadratic distance [KS90], and the Hellinger distance [BC89]. However, we shall use the popular Kullback-Leibler divergence (also known as the cross or relative entropy [CT91]), defined by

$$D_{KL}(P||Q) \stackrel{\text{def}}{=} \sum_{x\in\mathcal{X}} P(x)\log\frac{P(x)}{Q(x)}$$

We think of P as being the true, target distribution, and Q as being our best guess.

Strictly speaking, KL divergence is not a distance, since it only satisfies one of the three necessary properties of a metric[11]. Nonetheless, it upper bounds (some simple function of) the other measures, so if we achieve closeness in the KL sense, we achieve closeness with respect to the other measures, too. Let us briefly see why this is so.

The L_1 norm is defined as

$$||P-Q||_1 \stackrel{\text{def}}{=} \sum_{x\in\mathcal{X}} |P(x)-Q(x)|$$

the L_2 norm (quadratic distance) as

$$||P-Q||_2 \stackrel{\text{def}}{=} \left[\sum_{x\in\mathcal{X}} (P(x)-Q(x))^2\right]^{\frac{1}{2}}$$

and the variational distance as

$$\max_{B\subseteq\mathcal{X}}(P(B)-Q(B)) = \frac{||P-Q||_1}{2}.$$

Now, it can be shown [CT91, p.300] that $D_{KL}(P||Q) \geq \frac{1}{2\ln 2}||P-Q||_1^2$, so that if $D_{KL}(P||Q) \leq \epsilon$, then $||P-Q||_1 \leq \sqrt{\epsilon 2\ln 2}$, which is less than ϵ when $\epsilon > 2\ln 2 = 1.36$. It can be shown that the L_1 norm bounds the L_2 norm.

Note that the L_∞ norm, defined as

$$||P-Q||_\infty = \lim_{n\to\infty}\left[\sum_x (P(x)-Q(x))^n\right]^{1/n} = \max_x |P(x)-Q(x)|$$

behaves rather differently, since it does not consider the sum of the deviations, but rather the single largest deviation. Thus requiring two distributions to be close in the L_∞ sense is a much more stringent requirement than requiring them to be close in some aggregate sense. This will turn out to be the crucial difference between those classes of PFAs which are efficiently learnable and those that are not efficiently learnable, as we will see.

[11] A metric $d(x,y)$ should satisfy (1) $d(x,y)=0$ iff $x=y$. (2) $d(x,y)=d(y,x)$ (symmetry). (3) $d(x,z)\leq d(x,y)+d(y,z)$ (triangle inequality). The KL divergence only satisfies (1).

The KL divergence enjoys several other nice properties, as we now show [AW92]. Firstly, note that $D_{KL}(P||Q) = E_P \log(1/Q(\cdot)) - H(P)$, where $H(P) = \sum_{x \in \mathcal{X}} P(x)\log(1/P(x))$ is the entropy of the distribution P. Since $\log(1/P(x))$ is the length of the code word for x in an ideal (minimal length) code, $D_{KL}(P||Q)$ measures the expected additional code length (beyond the ideal code) needed to encode future data if we think the distribution is Q but actually it is P; hence choosing the Q which minimizes $D_{KL}(P||Q)$ will give us the minimal length encoding.

Now suppose P is the empirical distribution of the data $\mathcal{X}$. Finding the Q which minimizes $D_{KL}(P||Q)$ corresponds to minimizing

$$E_P \log(1/Q(\cdot)) = \frac{1}{|\mathcal{X}|}\sum_{x \in \mathcal{X}} \log\frac{1}{Q(x)} = \frac{1}{|\mathcal{X}|}\log\prod_{x \in \mathcal{X}}\frac{1}{Q(x)}$$

which corresponds to maximizing $\prod_{x \in \mathcal{X}} Q(x)$, i.e., the likelihood of the sample.

4.1.2 Model simplicity

So far we have only considered how well our model fits the data, but of course we are also interested in learning simple models, since they generalize better, that is, perform better on data which was not part of the training set. Enumerative algorithms, which generate every possible model in turn, will obviously find the simplest model, but since the number of possible models grows exponentially in the number of states, these algorithms are too inefficient to be useful. "Top down" algorithms, which only add states when forced to by the data (e.g., [RST94]), will also often find the simplest model (but not always, since they may not be able to delete states in the light of new data). So model simplicity is mainly a concern with "bottom up" algorithms, which start with a model which exactly fits the data (such as a canonical automaton) and then proceed to merge similar states.

There are two kinds of errors a bottom-up algorithm can make: type 1 errors, in which two equivalent states are not merged, and type 2 errors, in which two non-equivalent states are merged. If there are many type 1 errors, the resulting machine will be unnecessarily large; we can reduce the type 1 error rate by loosening our definition of similarity (at the risk of introducing more type 2 errors). If there are many type 2 errors, the inferred distribution will not be close to the target distribution; to reduce the type 2 error rate, we can make our definition of similarity more restrictive (at the risk of introducing more type 1 errors). Carrasco and Oncina [CO94] show that in their algorithm, the type 2 error rate vanishes in the limit of infinite data, and hence they can achieve identification in the limit with probability 1.

Stolcke and Omohundro [SO92, SO94] adopt a Bayesian approach in their bottom-up algorithm, and attempt to learn the HMM which has the greatest posterior probability, i.e., which maximises $\Pr(M|x)$. (This is called the Maximum A Posteriori (MAP) model.) Bayes' theorem tells us

$$\Pr(M|x) = \frac{\Pr(M)\Pr(x|M)}{Pr(x)}$$

so if we assign higher prior probability to simpler models, we can trade off model simplicity $\Pr(M)$ with goodness-of-fit $\Pr(x|M)$. Stolcke et al. achieve this by using a prior based on the description length of the (structure of the) model. This introduces a bias towards models with fewer states. (This bias is called an Occam factor).

4.2 Complexity results

Abe and Warmuth [AW92] showed that PFAs are trainable in time polynomial in ϵ, δ and m (the length of each input string), but *not* in $|\Sigma|$, assuming $RP \neq NP$.[12] However, this is a representation dependent result, that is, it assumes the hypothesis (as well as the target) must be a PFA. It is known that if the hypothesis class is allowed to be more general than the concept class, otherwise intractable problems sometimes become tractable. For example, assuming $RP \neq NP$, the class of 3-term DNF formulae is not efficiently PAC learnable unless we use 3-CNF to represent our hypotheses (see [KV94]).[13] However, Kearns et al. [KMR+94] prove that, under a certain assumption[14], it is not possible to efficiently PAC learn PFAs using any kind of hypothesis, when the hypothesis must be an evaluator. If the hypothesis is allowed to be a generator, they are only able to prove that learning distributions generated by polynomial sized circuits (a much larger class) is intractible. Since their proof is fairly simple, and also quite enlightening, we shall present it here. But first we must define the conditions under which it holds, namely under the Noisy Parity Assumption.

A parity function $f_a(x)$ takes $x = x_1 \dots x_n$ as an argument, and then computes the parity of the bits of x specified by its binary subscript $a = a_1 \dots a_n$, which can be thought of as a bit mask:

$$f_a(x) = \sum_{i=1}^{n} a_i x_i \pmod 2$$

The parity problem is to find the vector a given a set of examples $(x, f_a(x))$. This is easy to solve. The noisy parity problem is when the parity of the examples may be incorrect with probability $0 < \eta < \frac{1}{2}$. That is, the input set consists of pairs of the form (x, l), where $l = f_a(x)$ with probability $1 - \eta$ and $l = \neg f_a(x)$ with probability η. The Noisy Parity Assumption is the assumption that this problem cannot be efficiently solved in the PAC model when the example strings are chosen uniformly; there is good evidence for it.

Theorem 3 *Under the Noisy Parity Assumption, the class of distributions over $\{0,1\}^n$ generated by a DPFA is not efficiently learnable with an evaluator.*

Proof.
The basic idea is that for any parity function $f_a(\cdot)$, we construct a PFA whose distribution is uniform on the first n bits of x, and for which $x_{n+1} = f_a(x_1 \dots x_n)$ with probability $1 - \eta$, and the complement of this with probability η. Thus the PFA will generate noisy labelled examples of f_a; call this distribution D_a. The construction is illustrated in Figure 8. Now suppose we have a hypothesis evaluator $\hat{D}$ which approximates D_a, i.e., which has learnt the PFA. It can be shown that we can determine $f_a(x)$ for any $x \in \{0,1\}^n$ to arbitrary precision, which violates the Noisy Parity Assumption. In particular, suppose $KL(D_a||\hat{D}) \leq \epsilon(1 - H(\eta))$, where $H(\eta) = \eta \log \eta + (1 - \eta) \log(1 - \eta)$ is the binary entropy function. Then for a random $x \in \{0,1\}^n$, we can determine $f_a(x)$ with probability $1 - \epsilon$ by saying the parity bit is 0 if $\hat{D}(x0) > \hat{D}(x1)$, and 1 otherwise. ∎

There are several interesting things about this proof. First, it applies to *deterministic*, *acyclic*, levelled (layered), width two PFAs with *binary alphabets*. Clearly more general kinds of PFAs are going to be at

[12] NP is the set of languages for which we can test membership in polynomial time (but it may require exponential time to generate such members). RP is the set of languages for which we can test membership with probability at least $1 - \delta$ using a randomized algorithm which runs in time polynomial in $1/\delta$ and the length of the string.

[13] A 3-term DNF (Disjunctive Normal Form) formula is a disjunction of three conjunctions, e.g., $(p \wedge q) \vee (\neg p \wedge r \wedge s \wedge t) \vee (\neg q)$. A 3-CNF (Conjunctive Normal Form) formula is a conjunction of disjunctions, each of which contains exactly three literals, e.g., $(p \vee \neg q \vee r) \wedge (s \vee p \vee \neg p)$.

[14] In [GS94], they give an information-theoretic argument, which does not make any assumptions, that learning a hidden Markov chain is hard.

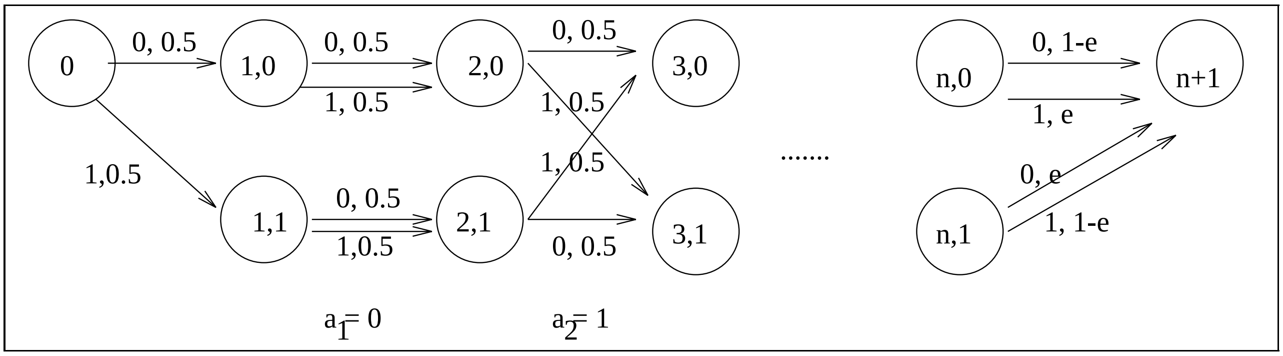

Figure 8: Constructing a noisy parity function using a PFA. The state $(i, 0)$ in the top row indicates that the first i bits of x that we have examined (as determined by a) have even parity. Similarly, the states in the bottom row store the fact that the parity so far is odd. Whenever $a_i = 1$, we switch "tracks" on encountering a 1, since this indicates that the parity has just changed. If $a_i = 0$, we don't switch tracks, since we are ignoring bit x_i. At the end, we emit the correct parity symbol with probability $1-\eta$ (in the figure, e denotes η).

least as hard to learn. Secondly, 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. To see this, consider two states p, q and their corresponding distributions P, Q. If p and q are on different levels, the distance between P and Q will be large simply because they generate strings of different lengths. So suppose p and q are on the same level l. For every string s of length $n-l$, if s has even parity, then starting from the even parity state the string $s0$ is generated with probability $2^{-(n-l)}(1-\eta)$ and the string $s1$ is generated with probability $2^{-(n-l)}\eta$. The opposite is true starting from an odd state. Even though each of these differences is small (especially for small l), their sum is $2(1-2\eta)$, and assuming η is bounded away from $\frac{1}{2}$, this is large.

$$\begin{aligned}
||P-Q||_1 &= \sum_{s\in\Sigma^{n-l}} |P(s0)-Q(s0)| + |(P(s1)-Q(s1)| \\
&= 2^{n-l}2^{-(n-l)}((1-2\eta)+(1-2\eta)) \\
&= 2(1-2\eta)
\end{aligned}$$

By imposing an extra condition on the class of Acyclic, Deterministic PFAs (APFAs), Ron, Singer and Tishby [RST95] prove that it is possible to efficiently learn this class using an algorithm we will study later. (They also conjecture that it is possible to learn *cyclic* DPFAs (with the distinguishability property) using their algorithm, but were unable to prove this case.[15]) The extra condition they impose is that the APFA be μ-distinguishable, defined as follows.

Definition 5 *We say that two states q_1, q_2 are* μ-distinguishable *if there exists a string s for which $|P^M_{q_1}(s) - P^M_{q_2}(s)| \geq \mu$. We say that a PFA M is μ-distinguishable if every pair of states in M is μ-distinguishable.*

This can be thought of as the probabilistic extension of k-distinguishability. Note that the distributions between any pair of states in the noisy parity PFA are far apart in the L_1 sense, but not in the L_∞ sense, and hence are not μ-distinguishable.

Finally, we mention a result on the complexity of learning PFAs with oracles. Tzeng [Tze89] discusses how to learn an input-PFA when the input consists of a set of strings and the corresponding probability distributions

[15] Dana Ron, personal communication, 8/15/95.

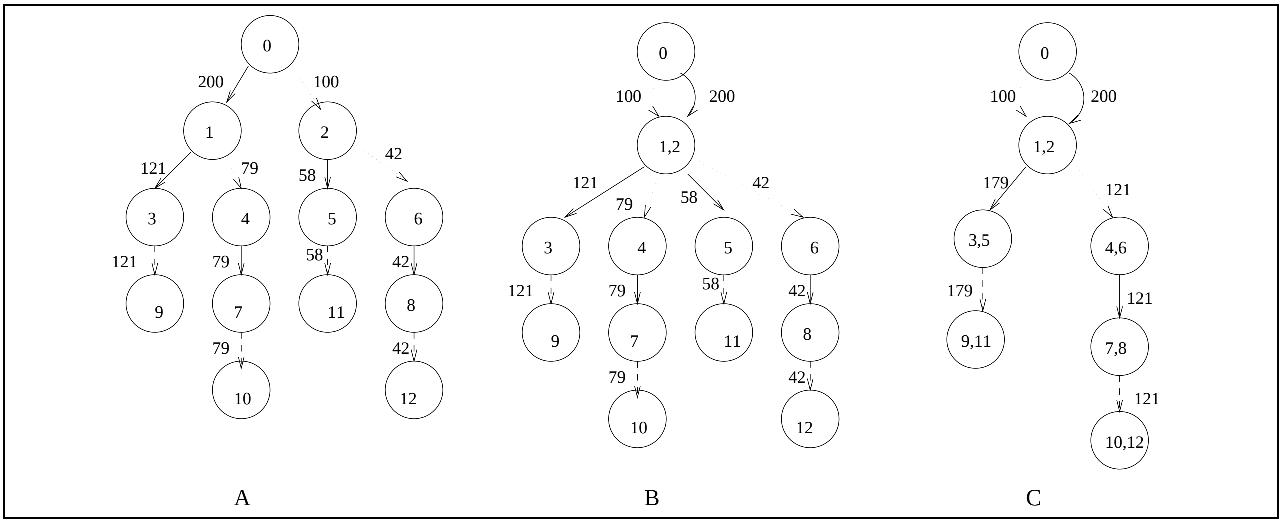

Figure 9: An illustration of merging states. A is the original canonical automaton. B is the result of merging nodes 1 and 2, and corresponds to a non-deterministic automaton. C is the result of merging all the children of 1 and 2. Solid edges are transitions labelled by 0, dashed egdes (which always lead into leaves) are labelled by the final symbol, and dotted edges are labelled by 1. Adapted from Figure 1 of [RST95].

they induce on the states, i.e., the probability that the automaton ends up in each state after accepting the string. He shows this problem is NP-complete (see also [GS94]). However, if the learner is given access to an oracle which can specify the probability distribution induced by any string, then the input-PFA can be learnt in polynomial time. He assumes throughout that the probabilities are rational.

4.3 The Spanish algorithm

The algorithm due to Carrasco and Oncina [CO94], which we call the "Spanish" algorithm for brevity, is basically a stochastic version of the "Greedy Russian" algorithm we discussed earlier. The difference is that the input is now a multiset of positive strings instead of a set of positive and negative strings. Also, the definition of equivalence between two states is changed to take probabilities into account: two states q, q' are defined to be similar if they assign similar probabilities to each of the branches in the first level of their subtree, and, recursively, if their corresponding children are similar. (We will define what it means for two probabilities to be similar shortly.) The recursive part of the definition effectively implies that all the corresponding branches in the subtrees below q, q' must be similar in probability. When q and q' are merged, the automaton may become non-deterministic, but since their children are similar, they will subsequently be merged, removing the non-determinism (see Figure 9).

The definition of similarity which they use is motivated by the following bound, due to Hoeffding, on the probability that a Bernoulli random variable with expected value p deviates from its empirical mean given by f successful trials out of n:

$$\left|p - \frac{f}{n}\right| < \sqrt{\frac{1}{2n}\log\frac{2}{\epsilon}} \text{ with probability larger than } (1-\epsilon).$$

They reject the equivalence of two nodes only if the two estimated probabilities differ by an amount which is greater than the sum of their confidence bounds. The empirical probabilities which are being compared are defined as follows. Let m_u denote the number of strings arriving at u and $m_u(\sigma)$ the number of strings leaving u via an arc labelled σ. Then the probabilities on two nodes u and v are similar if for all $\sigma \in \Sigma$, $|\frac{m_u(\sigma)}{m_u} - \frac{m_v(\sigma)}{m_v}| \leq \sqrt{\frac{1}{2}\log\frac{2}{\epsilon}}(\frac{1}{\sqrt{m_u}} + \frac{1}{\sqrt{m_v}})$. The advantage of this criterion is that it is sensitive to the sample size: nodes lower down the tree will have lower counts, so the criterion for similarity is relaxed to allow for bigger fluctuations.

They show that it is possible to choose a value for ϵ such that, in the limit of infinite data, the probability of type 2 errors (merging non-equivalent nodes) vanishes. It is also possible to reduce the type 1 error rate (rejection of compatibility between two equivalent nodes) provided one has a larger sample to compensate for the increased frequency of type 2 errors. This algorithm can therefore learn arbitrary DPFAs with probability 1 in the limit, from positive examples only.

4.4 The APFA algorithm

Ron, Singer and Tishby [RST95] present an algorithm which is very similar to the Spanish algorithm, in that it merges states based on a probabilistic similarity criterion. The difference is that they can characterize precisely how well the algorithm will perform on finite data sets. In particular, they show that if the sample size is large enough, they can return an ϵ-good hypothesis with probability at least $1-\delta$ for any ϵ and δ. Furthermore, the algorithm is efficient.

Of course, in view of the hardness results we discussed above, they must impose certain constraints to achieve these goals. Namely, their algorithm can only learn acyclic DPFAs (APFAs) which are μ-distinguishable (defined above). They speculate that the acyclicity condition is inessential, but needed it to make the proof go through. In fact their algorithm works on *levelled* acyclic APFAs, in which each node only makes a transition to the next level, except for transitions labelled by the final symbol (which denotes the termination of each string), which may "hop over" to the final node in the lowest level. However, every APFA having n states and depth D can be converted into an equivalent levelled APFA with at most $n(D-1)$ states, so this is not a major restriction.

The algorithm takes as input a confidence parameter δ, an error tolerance ϵ, a bound on the number of states n, and a distinguishability parameter μ. (The latter two assumptions can be removed by searching for an upper bound on n and a lower bound on μ.) It then proceeds to merge similar states within each level (thus keeping the graph acyclic), provided their counts exceed some minimum value m_0 (which is given in the proof). This prevents nodes whose counts are too low to be reliable from being merged.

The definition of similarity is similar to the Spanish case: Two nodes u and v are similar if for *every* string s, $|m_v(s)/m_v - m_u(s)/m_u| \leq \mu/2$, where $m_v(s)$ is the number of strings leaving v which have s as a prefix, and m_v is the total number of strings leaving v (i.e., its count). This can be computed by performing a depth-first search over the tails (subtrees) of u and v. The reason this does not take exponential time is that we don't need to examine every string of length equal to the longest distinguishing string: they can prove that, if the sample is big enough, there will be a string which distinguishes every pair of non-equivalent states of sufficient weight with high probability (because the machine is μ-distinguishable), and as soon as such a witness is found, we can quit searching the subtree.

At the end, for each level i in turn, all nodes which have too low a count are merged into a special node called small(i). The emission probabilities are estimated by the following smoothing formula:

$$\Pr(\sigma|q) = (m_u(\sigma)/m_u)(1 - (|\Sigma|+1)\gamma_{min}) + \gamma_{min}$$

where γ_{min} is a constant specified in the analysis of the algorithm. Smoothing is necessary to avoid saying that something has zero probability (and hence is impossible) just because it hasn't been seen in the data set.

4.5 The PSA algorithm

We have already defined Probabilistic Suffix Automata as a kind of DPFA, which can be viewed as a variable memory Markov chain. As a reminder, every state is labelled by a finite length string in Σ^*. If every label is at most length L, we call it an L-PSA. The label encodes how much "history" we pay attention to. (To ensure this condition, we require that, if there is an arc $p \xrightarrow{a} q$, then the label of q should be a suffix of $s \cdot a$, where s is the label of p, and $s \cdot a$ denotes string concatenation.) If the set of states is (labelled by) all of Σ^L, we have an order L Markov chain, since the next state transition probability depends on the last L symbols.

Ron, Singer and Tishby [RST94] show how to efficiently PAC-learn a PSA using a Probabilistic Suffix Tree (PST) as the hypothesis representation.[16] We shall just give a brief sketch of their algorithm.[17] Start with the empty tree and add a new node v with label s when given a string s only if, for some symbol σ, the empirical probability of seeing σ following s differs susbtantially from the empirical probability of seeing σ following suffix(s), the string labelling the parent of v. This is a top-down approach, since it incrementally grows the tree. A bottom-up approach would consist of building the PST for the whole dataset, and then trimming nodes which don't meet the above criterion.

Ron et al. say that PSAs are good for capturing the long-range stationary, statistical properties of the source, and APFAs are better for capturing short sequence statistics. They have used APFAs to segment cursive handwriting into characters, and PSAs to correct corrupted strings of characters, and combined the two in a complete handwriting recognition system.

4.5.1 When the input is a single string

Another interesting aspect of [RST94] algorithm is that it can take as input a single long string of length m. This is converted into the standard multiset by sliding an overlapping window of width ℓ, to generate $m - \ell + 1$ strings. ℓ is then an upper bound on the order of the Markov chain which can be learnt.

In the single string case, we can only infer ergodic models, since transient transitions are not exercised often enough to get reliable statistics. (An ergodic model is one in which you every state can be reached from every other with non-zero probability. The underlying directed graph must therefore be strongly connected.[18])

We now consider the problem of how long the input string must be in order to reliably learn a PSA M. Intuitively, we must observe each state whose stationary probability is non-negligible enough times so that we can identify the state as significant, and so that we can compute (approximately) the probability of emitting each symbol from that state. So the string must be long enough to ensure the convergence to the stationary distribution.

The stationary distribution $\Pi_M(\cdot)$ is the unique distribution satisfying

$$\Pi_M(q) = \sum_{q'} \Pi_M(q') \Pr(q|q').$$

[16] A suffix tree for a set of strings S is the prefix tree of the set of suffixes of the strings in S (see [Gus95]). A probabilistic suffix tree is a suffix tree with probabilities attached to each branch. This is similar to a DPFA.

[17] For an algorithm for learning a "normal" Markov chain in the limit with probability 1, see [Rud85].

[18] In [RST94] they say that it is also necessary that the machine be aperiodic, i.e., that the greatest common divisor of the lengths of the cycles in the underlying graph is 1.

Let R_M be the state transition matrix, and $\tilde{R_M}$ the time reversal of R_M, that is, $R_M(q, q')$ is the probability of going from q to q', and $\tilde{R_M}(q, q')$ is the probability of going from q' to q. Define the multiplicative reversibilization U_M of M by $U_M \stackrel{\text{def}}{=} R_M \tilde{R_M}$. Denote the second largest eigenvalue of U_M by $\lambda_2(U_M)$. They then allow the length of the string (and hence the running time of the algorithm) to be polynomial in ℓ, n, $|\Sigma|$, $\frac{1}{\epsilon}$, $\frac{1}{\delta}$ and $\frac{1}{1-\lambda_2(U_m)}$, where ℓ is the pre-specified maximum order of the chain, and n is the maximum number of states (if this is unknown, we can search for an upper bound). This means that, to learn Markov chains which only slowly converge to their stationary distribution, you need more data, and hence more time.

4.6 The Crutchfield algorithm

Crutchfield and Young [CY89] proposed an algorithm very similar to the two-pass Russian algorithm, but which has different input and output. Namely, the input is one long string, which is divided into all the overlapping substrings of length ℓ, and the output is a DPFA. (Hence the canonical automaton is a not necessarily complete tree of height ℓ.) The algorithm constructs a DFA M in a similar way to the Russian algorithm, the difference being that two states are deemed equivalent if all strings of length *exactly* k in their tails are the same. This means that we can only perform the equivalence test on nodes down to level $\ell - k$ (counting the root at level 0). They call process "topological reconstruction". They then attach probabilities to the arcs of this DFA by examining the counts on the nodes in the tree in a manner we now describe.

For each state $[n_i]$ in M, pick a representative node n_i from this equivalence class and examine all its arcs in the tree. If we are currently examining transition $n_i \xrightarrow{a} n_j$ in the tree, we add the transition $[n_i] \xrightarrow[c]{a} [n_j]$ to the automaton, where $c = c(n_i)/c(n_j)$. Unfortunately this is not sound: when we are learning DFAs, it does not matter which node we choose to "represent" state $[n_i]$, since they are all equivalent, but when are trying to learn PFAs, the representative we choose affects the resulting probabilities. They suggest picking as a representative the node highest up in the tree, where the counts are larger, and hence the statistics more reliable. We will discuss alternatives later, in the section on learning NPFAs.

5 Learning NPFAs (HMMs)

5.1 Using the Baum-Welch algorithm

A popular approach to learning the topology of an HMM is to construct a fully connected graph (a clique) on n nodes (where n is an upper bound on the number of states in the model you are trying to learn) and then let the Baum-Welch training procedure assign zero weight to arcs which are not necessary. This approach has two drawbacks. Firstly, n will usually be unknown. This is not a major problem, since we can incrementally try $n = 1$, $n = 2$, $\ldots$, until the results (as determined by, say, a cross-validation test) are deemed acceptable. (Some other methods of computing the right size of model have been proposed, but cross-validation seems to work the best in practice [KMNR95].) The more serious problem is that a fully connected graph on n nodes has $O(n^2)$ arcs, which is a large number of free parameters; not only are many of these potentially redundant, but models with a large number of free parameters need more training data to avoid overfitting.

5.2 Iterative state duplication

We now discuss an ad hoc method which uses the Baum-Welch procedure as part of its "inner loop". In [FAK94], the authors are interested in using HMMs for finding common patterns in proteins called motifs.

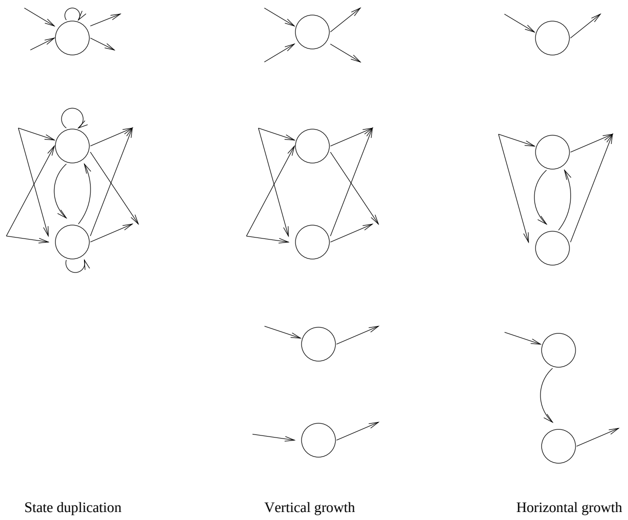

Figure 10: An illustration of the state duplication method. On the left we illustrate how all the connections to the state are copied to the new state. In the next two columns we illustrate how a particular state may be connected up, what the new topology will look like after duplication, and how it may end up looking after pruning. From Fig. 16 of [FAK94].

Their topology learning algorithm is as follows. They start out with a fully connected graph on n nodes. Then they train it using Baum-Welch; prune transitions with negligible probability (to prevent the model becoming too large); find the node (call it q) with the greatest number of in and out arcs (choose randomly if there are ties); make a copy of q; and repeat until sufficient accuracy is obtained. The intuition is that the most densely connected node may in fact represent two different states, so we make a duplicate copy of it, and let subsequent training and pruning eliminate any unnecessary new arcs (see Figure 10). Since the Baum-Welch algorithm gets stuck in local optima, they repeat the whole process from different random starting points.

A similar approach, called the "Successive State Splitting" Algorithm, is described in [TS94] in the context of left-right speech recognition models.

5.3 Model surgery

Another approach is to construct a topology by hand, and then "fine tune" it after it has been trained. Consider, for example, the approach of "model surgery" used in HMM models of protein sequences [KBM$^+$94]: they start out with a "backbone" of n states, with transitions to skip over states, and transitions to side states with self-loops, which allow the insertion of extra symbols. If, after training, it is found that the skipping transitions are used very frequently, they sdelete the part of the chain which is skipped over. Similarly, if the insert states are used very frequently, they add more states into the backbone. In this paper we are interested in fully automatic means of inferring the correct topology. (It would be interesting to consider the possibility of using the handcrafted topology as a prior to an automatic Bayesian method.)

5.4 Using a Bayesian state-merging method

The most well-principled approach to learning HMMs is by Stolcke and Omohundro [SO92, SO94]. As in many of the methods for learning PFAs, they repeatedly merge states in the prefix tree (i.e., the maximum likelihood HMM), but they no longer recursively merge the children of two merged nodes, and hence the resulting machine becomes non-deterministic. This means we cannot compute the similarity of two nodes based on their subtrees, since there may be many branches below each node with the same label. Instead, they consider all pairs of states, and then compute the posterior probability of the model in which this pair is merged; the pair which maximises this posterior is then picked as the pair to be actually merged. (Hence this is a greedy or best-first algorithm, which can get stuck in local optima.) They continue to merge until the posterior stops increasing. (A large part of their lucid paper is concerned with how to compute these probabilities efficiently, which we will not go into here.) They show experimently that their algorithm outperforms the approach of learning the weights on a fully-connected graph using the standard Baum-Welch algorithm. They have also incorporated it into a speech recognition system, although, as noted above, an APFA performs as well on this same dataset and can be learnt much more quickly.

As mentioned above, they compute a prior probability for the model structure which produces a bias towards simpler models. They also use a prior for the transition and emission probabilities, which is a good way to deal with small training sets. Since these probability distributions are multinomials, they use a Dirichlet prior, which is the conjugate prior for a multinomial. This has the effect of adding "virtual samples" to the data set, which is similar to the somewhat more ad hoc approach to data smoothing of simply adding a constant to all the empirical counts, of which we saw an example above. (Similar approaches are used in other applications of HMMs, e.g., [BHK+93].)

5.5 A new algorithm for learning NPFAs

Intuitively, it seems that it ought to be possible to extend the idea of merging states whose signatures (subtrees) are similar to the case of learning non-deterministic PFAs. We decided to do this by using the two-pass approach used in the Russian and Crutchfield algorithms, where we first identify states that are similar, and then merge them.

We say that two states q, q' are similar if their k-tails are δ-similar, which means they contain the same set of strings, and for each string x in the tails, $|\Pr(x|q) - \Pr(x|q')| \leq \delta$. (This is similar to μ-indistinguishability, except that we are restricting attention to the strings in the subtrees.) Of course, as we go down the tree, the counts get smaller, and hence the statistics less reliable. We must therefore either ignore nodes which have too low a count (as in the APFA algorithm), or have a sample-size sensitive measure of similarity (as in the Spanish algorithm). This problem is especially acute since we do all the similarity computations before merging any nodes (which increases the counts). The most important difference of this definition from previous ones is that we *do not require the children of two similar nodes to be similar*. Hence, q and q' may be deemed similar, yet q transitions to state r on letter a and q' transitions to state $s \neq r$ on letter a. The resulting machine may therefore be non-deterministic. (In the APFA algorithm, they do not explicitely require that the children of two similar nodes be similar, but they recursively merge the children of two similar nodes, thus removing any non-determinism.)

δ-similarity is not an equivalence relation; rather, it produces clusters of similar points, which correspond to the same state. We need to be able to perform two operations on our clusters (states): in the first pass, decide if a node in the tree belongs in a cluster (i.e., if it is similar enough to the other nodes in that cluster), and in the second pass, find all the arcs leaving this state. The membership operation is essentially comparing a node with a moving target, since the number of nodes in a cluster is continually growing as we walk down the tree. The arc operation is also tricky: When computing the arcs from a state, should we consider all the nodes in that cluster, and if so, how do we combine their counts? We are currently investigating these issues.

6 Acknowledgements

Part of this work was carried out while the author was a summer student at Los Alamos National Laboratory, Theoretical Biology and Biophysics (T-10) group, and part was supported by DOE grant DE-FG03-90ER60999. The author would like to thank Dana Ron (MIT) and Catherine Macken (Los Alamos) for their careful reading of an earlier version of this manuscript, and Jim Crutchfield and Jim Hanson at the Santa Fe Institute for stimulating discussions.

References

[AHU83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[Ang78] D. Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350, 1978.

[Ang87] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

[Ang92] D. Angluin. Computational learning theory: Survey and selected bibliography. In *ACM Symposium on the Theory of Computing*, pages 351–369, 1992.

[AS83] D. Angluin and C. H. Smith. Inductive inference: Theory and methods. *Computing Surveys*, 15(3):237–269, 1983.

[AU72] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling. Vol. I.* Prentice Hall, 1972.

[AW92] N. Abe and M. K. Warmuth. On the computational complexity of approximating ditributions by probabilistic automata. *Machine Learning*, 9:205–260, 1992.

[BC89] A. R. Barron and T. M. Cover. Miniumum complexity density estimation. *IEEE Trans. Info. Theory*, 1989.

[BCHM94] P. Baldi, Y. Chauvin, T. Hunkapiller, and M. A. McClure. Hidden markov models of biological primary sequence information. *Proc. of the National Academy of Science, USA*, 91:1059–1063, 1994.

[BF72] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. on Computers*, pages 592–597, 1972.

[BHK+93] M. P. Brown, R. Hughey, A. Krogh, I. S. Mian, K. Sjölander, and D. Haussler. Using dirichlet mixtures priors to derive hidden Markov models for protein families. In *International Conf. on Intelligent Systems for Molecular Biology*, pages 47–55, 1993.

[BK76] A. W. Biermann and R. Krishnaswamy. Constructing programs from example computations. *IEEE Trans. on Software Engineering*, SE-2:141–153, 1976.

[CO94] R. C. Carassco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *2nd Intl. Colloq. on Grammatical Inference and Applications*, pages 139–152, 1994.

[Con71] J. H. Conway. *Regular Algebra and Finite Machines.* Chapman and Hall, 1971.

[Cru91] J. P. Crutchfield. Reconstructing language hierarchies. In H. Atmanspacher and H. Scheingraber, editors, *Information Dynamics*, pages 45–60. Plenum Press, 1991.

[CT91] T. M. Cover and J. A. Thomas. *Elements of Information Theory.* John Wiley, 1991.

[CY89] J. P. Crutchfield and K. Young. Inferring statistical complexity. *Physical Review Letters*, 63(2):105–108, 1989.

[DM93] S. Das and M. Mozer. A unified gradient-descent/ clustering algorithm for finite state machine induction. *Advances in Neural Information Processing Systems*, 6, 1993.

[FAK94] Y. Fujiwara, M. Asogawa, and A. Konagaya. Stochastic motif extraction using hidden markov model. In *International Conf. on Intelligent Systems for Molecular Biology*, 1994.

[FKM+95] Y. Freund, M. Kearns, Y. Mansour, D. Ron, R. Rubinfeld, and R. E. Schapire. Efficient algorithms for learning to play repeated games against computationally bounded adversaries. In *IEEE Symposium on the Foundations of Computer Science*, 1995.

[FKR+93] Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. E. Schapire, and L. Sellie. Efficient learning of typical finite automata from random walks. In *ACM Symposium on the Theory of Computing*, pages 315–324, 1993.

[Fu82] K. S. Fu. *Syntactic Pattern Recognition and Applications.* Prentice Hall, 1982.

[Gai76] B. R. Gaines. Behaviour/structure transformations under uncertainty. *Intnl. J. of Man-Machine Studies*, 8:337–365, 1976.

[Gol76] E. M. Gold. Language identification in the limit. *Information and Control*, 10:447–474, 1976.

[Gol78] E. M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978.

[GS94] D. Gillman and M. Sipser. Inference and minimization of hidden markov chains. In *Proc. of the Workshop on Computational Learning Theory*, pages 147–158, 1994.

[Gus95] Dan Gusfield. *Algorithms on Strings: A Dual View from Computer Science and Computational Molecular Biology.* 1995. In preparation.

[HGC94] D. Heckerman, D. Geiger, and M. Chickering. Learning bayesian networks: the combination of knowledge and statistical data. Technical Report MSR-TR-09-09, Microsoft Research, 1994.

[HKP91] J. Hertz, A. Krogh, and R. G. Palmer. *An Introduction to the Theory of Neural Comptuation.* Addison-Wesley, 1991.

[Hop71] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computation.* Academic Press, 1971.

[HRSJ91] G. Hachtel, J.-K. Rho, F. Somenzi, and R. Jacoby. Exact and heuristic algorithms for the miminimization of incompletely specified state machines. In *Proc. European Design Automation Conference*, 1991. UCB Engin TK7868 .E93.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.

[JLM92] F. Jelinek, J. D. Lafferty, and R. L. Mercer. Basic methods of probabilistic context-free grammars. *Computational Linguistics*, To appear, 1992.

[KBM+94] A. Krogh, M. Brown, I. S. Mian, K. Sjölander, and D. Haussler. Hidden markov models in computational biology: Applications to protein modelling. *J. of Molecular Biology*, 235:1501–1531, 1994.

[KMH93] A. Krogh, I. S. Mian, and D. Haussler. A hidden markov model that finds genes in *E. Coli* DNA. Technical Report USCS-CRL-93-33, U. C. Santa Cruz, Dept. Comp. Sci., 1993. Available from `ftp.cse.ucsc.edu`, directory `pub/dna`.

[KMNR95] M. Kearns, Y. Mansour, A. Y. Ng, and D. Ron. An experimental and theoretical comparison of model selection methods. In *Proc. of the Workshop on Computational Learning Theory*, 1995.

[KMR+94] M. Kearns, Y. Mansour, D. Ron, R. Rubinfeld, R. E. Schapire, and L. Sellie. On the learnability of discrete distributions. In *ACM Symposium on the Theory of Computing*, pages 273–282, 1994.

[KS90] M. Kearns and R. Schapire. Efficient distribution-free learning of probabilistic concepts. In *IEEE Symposium on the Foundations of Computer Science*, 1990.

[KV89] M. Kearns and L. G. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. In *ACM Symposium on the Theory of Computing*, pages 433–444, 1989.

[KV94] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.

[KVBSV94] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. In *Proc. Design Automation Conference*, 1994.

[Lan92] K. J. Lang. Random DFAs can be approximately learned from sparse uniform examples. In *Proc. of the Workshop on Computational Learning Theory*, 1992. Volume 5.

[Li90] W. Li. A relation between complexity and entropy for markov chains and regular languages. Technical Report 90-025, Santa Fe Institute, 1990. Submitted to *J. Physics A*.

[MB91] M. C. Mozer and J. Bachrach. SLUG: A connectionist architecture for inferring the structure of finite-state environments. *Machine Learning*, 7:139–160, 1991.

[Mic80] L. Miclet. Regular inference with a tail-clustering method. *IEEE Trans. on Systems, Man, and Cybernetics*, SMC-10(11):737–743, 1980.

[Mic90] L. Miclet. Grammatical inference. In H. Bunke and A. Sanfeliu, editors, *Syntactic and Structural Pattern Recognition. Theory and Applications*, chapter 9. World Scientific, 1990.

[Min67] M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice Hall, 1967.

[OE95] A. Oliveira and S. Edwards. Inference of state machines from examples of behavior. Technical report, Dept. of Electrical Engineering, U. C. Berkeley, 1995. Available from `http://www.eecs.berkeley.edu/~sedwards`

[Paz71] A. Paz. *Introduction to Probabilistic Automata.* Academic Press, 1971.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1988.

[PF91] S. Porat and J. A. Feldman. Learning automata from ordered examples. *Machine Learning*, 7:109–138, 1991.

[Pfl73] C. E. Pfleeger. State reduction in incompletely specified finite state machines. *IEEE Trans. Computers*, C-22:1099–1102, 1973.

[PW93] L. Pitt and M. K. Warmuth. The minimum consistent dfa cannot be approximated within any polynomial. *J. of the ACM*, 40(1):95–142, 1993.

[Rab63] M. O. Rabin. Probabilistic automata. *Information and Control*, 6(3):230–245, 1963.

[Rab89] L. R. Rabiner. A tutorial in hidden markov models and selected applications in speech recognition. *Proc. of the IEEE*, 77(2):257–286, 1989.

[RR95] D. Ron and R. Rubinfeld. On the exact learnability of automata with small cover time. In *Proc. of the Workshop on Computational Learning Theory*, 1995.

[RS87] R. L. Rivest and R. E. Schapire. Diversity based inference of finite automata. In *IEEE Symposium on the Foundations of Computer Science*, pages 78–87, 1987.

[RS89] R. L. Rivest and R. E. Schapire. Inference of finite automata using homing sequences. In *ACM Symposium on the Theory of Computing*, pages 411–420, 1989.

[RST93] D. Ron, Y. Singer, and N. Tishby. The power of amnesia. In *Advances in Neural Information Processing Systems*, 1993. Volume 6.

[RST94] D. Ron, Y. Singer, and N. Tishby. Learning probabilistic automata with variable memory length. In *Proc. of the Workshop on Computational Learning Theory*, 1994.

[RST95] D. Ron, Y. Singer, and N. Tishby. On the learnability and usage of acyclic probabilistic finite automata. In *Proc. of the Workshop on Computational Learning Theory*, 1995.

[Rud85] S. Rudich. Inferring the structure of a markov chain from its output. In *IEEE Symposium on the Foundations of Computer Science*, 1985.

[Seb84] G. A. F. Seber. *Multivariate Observations*. John Wiley, 1984.

[SO92] A. Stolcke and S. M. Omohundro. Hidden markov model induction by bayesian model merging. In *Advances in Neural Information Processing Systems*, 1992. Volume 5.

[SO94] A. Stolcke and S. M. Omohundro. Best-first model merging for hidden markov model induction. Technical Report TR-94-003, International Computer Science Institute, 1994. Available from `http://www.icsi.berkeley.edu`.

[TB73] B. A. Trakhtenbrot and Ya. M. Barzdin'. *Finite Automata: Behavior and Synthesis*. North-Holland, 1973.

[TS94] J. Takami and S. Sagayama. Automatic generation of hidden markov networks by a successive state splitting algorithm. *Systems and Computers in Japan*, 25(12), 1994. Translated from Japanese.

[Tze89] W-G. Tzeng. The equivalence and learning of probabilistic automata. In *IEEE Symposium on the Foundations of Computer Science*, pages 268–273, 1989.

[You91] K. Young. *The Grammar and Statistical Mechanics of Complex Physical Systems*. PhD thesis, Physics Dept., U.C. Santa Cruz, 1991.