JMLR: Workshop and Conference Proceedings 21:1–15, 2012 The 11th International Conference on GI

Bootstrapping and Learning PDFA in Data Streams

B. Balle BBALLE@LSI.UPC.EDU
J. Castro CASTRO@LSI.UPC.EDU
R. Gavaldà GAVALDA@LSI.UPC.EDU
LARCA Research Group, LSI Department, Universitat Politècnica de Catalunya (Barcelona)

Editor: Jeffrey Heinz, Colin de la Higuera, and Tim Oates

Abstract

Markovian models with hidden state are widely-used formalisms for modeling sequential phenomena. Learnability of these models has been well studied when the sample is given in batch mode, and algorithms with PAC-like learning guarantees exist for specific classes of models such as Probabilistic Deterministic Finite Automata (PDFA). Here we focus on PDFA and give an algorithm for infering models in this class under the stringent *data stream* scenario: unlike existing methods, our algorithm works incrementally and in one pass, uses memory sublinear in the stream length, and processes input items in amortized constant time. We provide rigorous PAC-like bounds for all of the above, as well as an evaluation on synthetic data showing that the algorithm performs well in practice. Our algorithm makes a key usage of several old and new sketching techniques. In particular, we develop a new sketch for implementing bootstrapping in a streaming setting which may be of independent interest. In experiments we have observed that this sketch yields important reductions in the examples required for performing some crucial statistical tests in our algorithm.

Keywords: Data Streams, PDFA, PAC Learning, Sketching, Bootstrapping

1. Introduction

Data streams are a widely accepted computational model for algorithmic problems that have to deal with vast amounts of data in real-time. Over the last ten years, the model has gained popularity among the Data Mining community, both as a source of challenging algorithmic problems and as a framework into which several emerging applications can be cast (Aggarwal, 2007; Gama, 2010). From these efforts, a rich suite of tools for data stream mining has emerged, solving difficult problems related to application domains like network traffic analysis, social web mining, and industrial monitoring.

Most algorithms in the streaming model fall into one of the following two classes: a class containing primitive building blocks, like change detectors and sketching algorithms for computing statistical moments and frequent items; and a class containing full-featured data mining algorithms, like frequent itemsets miners, decision tree learners, and clustering algorithms. A generally valid rule is that primitives from the former class can be combined for building algorithms in the latter class.

In this paper we present two data streaming algorithms, one in each of the above categories. First we develop a new sketching primitive for implementing bootstrapped estimators in the data stream model. The *bootstrap* is a well-known method in statistics for testing

hypothesis in non-parametric settings. Second, we present an algorithm for PAC learning probabilistic deterministic finite automata (PDFA) from a stream of strings. This new algorithm uses the bootstrapping sketch as a primitive for statistical testing. The so-called *state-merging* approach was introduced in the Grammatical Inference community for learning finite automata, discrete Markovian models with hidden states, and other state-based representations.

Our interest in this problem comes from the field known as Process Mining, which is devoted to inferring (probabilistic) state-transition models like grammars, Petri nets, finite state machines, etc. from process traces. With the advent of the Web and other high-speed stream processing environments, process mining methods have to deal with ever-increasing data sets, where the target to be modeled possibly changes or drifts over time. While the data streams computational model is a natural framework for this applications, adapting current process mining algorithms to work in this demanding model is a challenging problem. We propose a solution to this problem based on an algorithm for learning PDFA that meets the memory and processing time requirements imposed by the stringent streaming setting.

Several state-merging algorithms for learning PDFA have been proposed in the literature. Some of them are based on heuristics, while others come with theoretical guarantees, either in the limit, or in the PAC sense (Carrasco and Oncina, 1994; Ron et al., 1998; Clark and Thollard, 2004; Palmer and Goldberg, 2007; Guttman et al., 2005; Castro and Gavaldà, 2008). However, all of them are batch oriented and require the full sample to be stored in memory. In addition, most of them perform serveral passes over the sample. In contrast, our algorithm keeps in memory only the relevant parts of the observed sample and its memory usage grows sublinearly with the number of strings it processes.

Another important feature of our algorithm is its ability to perform statistical tests to determine similarity between states in an *adaptive* manner. This means that as soon as enough examples are available a decision is made. This behavior is essentially different from the algorithms of (Clark and Thollard, 2004; Palmer and Goldberg, 2007) which share similar PAC guarantees with our algorithm. If memory issues were ignored, both of these algorithms could be easily adapted to the streaming setting because they work by asking for a sample of a certain size upfront which is then used for learning the target. Thus, these algorithms always work with the worst-case sample size, while our algorithm is able to adapt to the complexity of the target and learn easy targets using less examples than predicted by the worst case analysis. Our algorithm resembles that of Castro and Gavaldà (2008) in this particular aspect, though ther is still a significant difference. Their algorithm is adaptive in the sense that it takes a fixed sample and tries to make the best of it. In contrast, having access to an unbounded stream of examples, adaptiveness in our algorithm comes from its ability to make accurate decisions as soon as possible.

State-merging algorithms usually split different states with relatively few examples, while merging with high confidence is in general extremely costly and always operates under *worst-case* conditions. To deal with this problem, our algorithm uses bootstrapped estimates for testing statistical similarity. Tests in all previous PAC learning algorithms for PDFA are derived from Hoeffding bounds or VC theory. Though useful, in general this bounds are distribution-independent, which means they work even under very adverse distributions which one may no encouter in reality. As a consequence, sometimes the sample bounds obtained from these theories are rather pessimistic. In contrast, bootstrapped estimates

are usually distribution-dependent and thus can perform better in many circumstances, while still being regarded as very accurate (Hall, 1997). Here we present a test based on bootstrapped confidence intervals with formal guarantees that has the same rate of convergence as usual VC bounds. Furthermore, in the experimental section we show that a different test also based on bootstrapping can perform better than VC-based tests in practice. Though we have no finite sample analysis for this practical test, our results guarantee that it is not worse than tests based on uniform convergence bounds. The use of the bootstrapped testing scheme comes at the price of an increase in the total memory requirements of the algorithm. This increase takes the form of a multiplicative constant that can be adjusted by the user to trade-off learning speed versus memory consumption. In a streaming setting this parameter should be tuned to fit all the data structures in the main memory of the machine running our algorithm.

The structure of the paper is as follows. Section 2 gives background and notation. Section 3 presents the technical results upon which our tests are based. Section 4 gives a detailed explanation of our state-merging algorithm and its analysis. In Section 5 we present some experiments with an implementation of our algorithm. Section 6 concludes with some future work. All proofs are omitted due to space constraints and will appear elsewhere.

2. Background

As customary, we use the notation $\widetilde{O}(f)$ as a variant of $O(f)$ that ignores logarithmic factors. Unless otherwise stated we assume the unit-cost computation model, where (barring model abuses) e.g. an integer count can be stored in unit memory and operations on it take unit time. If necessary statements can be translated to the logarithmic model, where e.g. a counter with value t uses memory $O(\log t)$, or this factor is hidden within the $\widetilde{O}(\cdot)$ notation.

2.1. Learning Distributions in the PAC Framework

Several measures of divergence between probability distributions are considered. Let Σ be a finite alphabet and let D_1 and D_2 be distributions over $\Sigma^\star$. The total variation distance is $\mathrm{L}_1(D_1, D_2) = \sum_{x\in\Sigma^\star} |D_1(x) - D_2(x)|$. Another distance between distributions over strings useful in the learning setting is the supremum over prefixes distance, or prefix-L_∞ distance: $\mathrm{L}_\infty^{\mathrm{p}}(D_1, D_2) = \max_{x\in\Sigma^\star} |D_1(x\Sigma^\star) - D_2(x\Sigma^\star)|$, where $D(x\Sigma^\star)$ denotes the probability under D of having x as a prefix.

Now we introduce the PAC model for learning distributions. Let $\mathcal{D}$ be a class of distributions over some fixed set X. Assume $\mathcal{D}$ is equipped with some measure of *complexity* assigning a positive number $|D|$ to any $D \in \mathcal{D}$. We say that an algorithm $\mathtt{L}$ *PAC learns* a class of distributions $\mathcal{D}$ using $S(\cdot)$ examples and time $T(\cdot)$ if, for all $0 < \varepsilon, \delta < 1$ and $D \in \mathcal{D}$, with probability at least $1 - \delta$, the algorithm reads $S(1/\varepsilon, 1/\delta, |D|)$ examples drawn i.i.d. from D and after $T(1/\varepsilon, 1/\delta, |D|)$ steps outputs a hypothesis $\widehat{D}$ such that $\mathrm{L}_1(D, \widehat{D}) \leq \varepsilon$. The probability is over the sample used by $\mathtt{L}$ and any internal randomization. As usual, PAC learners are considered efficient if the functions $S(\cdot)$ and $T(\cdot)$ are polynomial in all of their parameters.

2.2. PDFA and State-Merging Algorithms

A PDFA T is a tuple $\langle Q, \Sigma, \tau, \gamma, \xi, q_0 \rangle$ where Q is a finite set of states, Σ is an arbitrary finite alphabet, $\tau : Q \times \Sigma \longrightarrow Q$ is the transition function, $\gamma : Q \times (\Sigma \cup \{\xi\}) \longrightarrow [0, 1]$ defines the probability of emitting each symbol from each state – where we must have $\gamma(q, \sigma) = 0$ when $\sigma \in \Sigma$ and $\tau(q, \sigma)$ is not defined, – ξ is a special symbol not in Σ reserved to mark the end of a string, and $q_0 \in Q$ is the initial state.

It is required that $\sum_{\sigma \in \Sigma \cup \{\xi\}} \gamma(q, \sigma) = 1$ for every state q. Transition function τ is extended to $Q \times \Sigma^\star$ in the usual way. Also, the probability of generating a given string $x\xi$ from state q can be calculated recursively as follows: if x is the empty word λ the probability is $\gamma(q, \xi)$, otherwise x is a string $\sigma_0\sigma_1 \ldots \sigma_k$ with $k \geq 0$ and $\gamma(q, \sigma_0\sigma_1 \ldots \sigma_k\xi) = \gamma(q, \sigma_0)\gamma(\tau(q, \sigma_0), \sigma_1 \ldots \sigma_k\xi)$. For each state q a probability distribution D_q on $\Sigma^\star$ can be defined: for each x, probability $D_q(x)$ is $\gamma(q, x\xi)$. The probability of generating a prefix x from a state q is $\gamma(q, x) = \sum_y D_q(xy) = D_q(x\Sigma^\star)$. The distribution defined by T is the one corresponding to its initial state, D_{q_0}. Most commonly, we will identify a PDFA and the distribution it defines. The following parameter is used to measure the complexity of learning a particular PDFA.

Definition 1 *We say distributions D_1 and D_2 are* μ-distinguishable *when $\mu \leq \mathrm{L}_\infty^{\mathrm{p}}(D_1, D_2)$. A PDFA T is μ-distinguishable when for each pair of states q_1 and q_2 their corresponding distributions D_{q_1} and D_{q_2} are μ-distinguishable. The* distinguishability *of a PDFA is defined as the supremum over all μ for which the PDFA is μ-distinguishable.*

State-merging algorithms form an important class of strategies of choice for the problem of inferring a regular language from samples. Basically, they try to discover the target automaton graph by successively applying tests in order to discover new states and merge them to previously existing ones according to some similarity criteria. In addition to empirical evidence showing a good performance, state-merging algorithms also have theoretical guarantees of learning in the limit the class of regular languages (see de la Higuera (2010)).

Clark and Thollard (2004) adapted the state-merging strategy to the setting of learning distributions generated by PDFA and showed PAC-learning results parametrized by the distinguishability of the target distribution. The distinguishability parameter can sometimes be exponentially small in the number of states in a PDFA. However, there exists strong evidence suggesting that polynomiality in the number of states alone may not be achievable.

2.3. Data Streams

The *data stream* computation model has established itself in the last fifteen years for the design and analysis of algorithms on high-speed sequential data (Aggarwal, 2007). It is characterized by the following assumptions:

1. The input is a potentially infinite sequence of items $x_1, x_2, \ldots, x_t, \ldots$ from some (large) universe X

2. Item x_t is available only at the tth time step and the algorithm has only that chance to process it, probably by incorporating it to some summary or sketch; that is, only one pass over the data is allowed

3. Items arrive at high-speed, so the processing time for item must be very low – ideally, constant time, but most likely, logarithmic in t and $|X|$

4. The amount of memory used by algorithms (the size of the sketches alluded above) must be sublinear in the data seen so far at every moment; ideally, at time t memory must be polylogarithmic in t – for many computations, this is impossible and memory of the form t^c for constant $c < 1$ may be required (logarithmic dependence on $|X|$ is also desirable)

5. Approximate, probabilistic answers are often acceptable

A large fraction of the data stream literature discusses algorithms working under worst-case assumptions on the input stream, e.g. compute the required (approximate) answer at all times t for every possible values of $x_1, \ldots, x_t$ (Lin and Zhang, 2008; Muthukrishnan, 2005). For example, several sketches exist to compute an ε-approximation to the number of distinct items in memory $O(\log(t|X|)/\varepsilon)$. In machine learning and data mining, this is often not the problem of interest: one is interested in modeling the current "state of the world" at all times, so the current items matter much more than those from the far past (Bifet, 2010; Gama, 2010). An approach is to assume that each item x_t is generated by some underlying distribution D_t over X, that varies over time, and the task is to track the distributions D_t (or its relevant information) from the observed items. Of course, this is only possible if these distributions do not change too wildly, e.g. if they remain unchanged for fairly long periods of time ("distribution shifts", "abrupt change"), or if they change only very slightly from t to $t+1$ ("distribution drift"). A common simplifying assumption (which, though questionable, we adopt here) is that successive items are generated independently, i.e. that x_t depends only on D_t and not on the outcomes x_{t-1}, x_{t-2}, etc.

In our case, the universe X will be the infinite set $\Sigma^\star$ of all string over a finite alphabet Σ. Intuitively, the role of $\log|X|$ will be replaced by a quantity such as the expected length of strings under the current distribution.

3. Stream Statistics for Similarity Testing

Suppose we have two streams $(x_t)_{t>0}$ and $(x'_t)_{t>0}$ of strings over $\Sigma^\star$ generated by distributions D and D' respectively. Assume that either $D = D'$ or $\mathrm{L}^{\mathrm{p}}_\infty(D, D') \geq \mu$ for some fixed μ. We want to design a statistical test that after reading strings from each stream returns an answer from $\{\textsc{distinct}, \textsc{similar}, \textsc{unknown}\}$ which is accurate with high probability – UNKNOWN meaning that it cannot confidently conclude any of the other two answers. Ideally we would like the whole process to run in time linear in the number of strings processed and to use sublinear memory.

The test presented here is essentially asymmetric, using different statistics for testing similarity and dissimilarity between distributions. The dissimilarity test is based on VC bounds for uniform convergence of relative frequencies to probabilities. In contrast, our similarity test is based on adapting the bootstrapping method from statistics to the data streams scenario. Both tests use information collected from the streams in the form of sketches.

3.1. Finding Frequent Prefixes

Our algorithm will make an intensive use of the Space-Saving algorithm by Metwally et al. (2005). In particular, given a stream of strings we produce a new stream that contains, for every string x in the original stream, all the prefixes of x. This new stream is fed to a Space-Saving sketch with an adequate number of counters in order to keep information about *frequent prefixes* in the stream of strings. This information will be used to estimate the prefix-L_∞ between two distributions over strings.

We begin by recalling the basic properties of the Space-Saving sketch introduced in Metwally et al. (2005). Given a number of counters K, the Space-Saving sketch $\texttt{SpSv}(K)$ is a data structure that uses memory $O(K)$ at all times and has two basic operations. The first one receives an element and adds it to the sketch in time $O(1)$. The second is a retrieval operation that given some $\varepsilon \geq 1/K$ takes time $O(1/\varepsilon)$ and returns a set of at most K pairs of the form $(x, \hat{f}_x)$ with $0 \leq \hat{f}_x - f_x \leq 1/K$ – where f_x is the relative frequency of x among all the elements added to the sketch so far – that is guaranteed to contain every x whose $f_x \geq \varepsilon$. Using these operations an algorithm can mantain a summary of the most frequent elements seen in a stream together with an approximation to their current relative frequency.

In order to be able to retrieve *frequent prefixes* in a stream of strings, some modifications have to be made to this sketch. First, note that given a distribution D over strings in $\Sigma^\star$, the probabilities of all prefixes do not necessarily add up to one because prefixes are not independent of each other. In fact, it can be shown that $\sum_x D(x\Sigma^\star) = L + 1$, where $L = \sum_x |x| D(x)$ is the expected length of D (Clark and Thollard, 2004). Therefore, a Space-Saving sketch with $(L+1)K$ counters can be used to retrieve prefixes with relative frequencies larger than some $\varepsilon \geq 1/K$ and approximating these frequencies with error at most $1/K$. When computing these relative frequencies, the absolute frequency needs to be divided by the number of strings added to the sketch so far (instead of the number of prefixes).

We encapsulate this behavior into a *Prefix-Space-Saving* sketch $\texttt{PrefSpSv}(K, L)$ with two parameters: an integer K that measures the desired accuracy on relative frequency approximation for prefixes, and an upper bound L on the expected length of strings in the stream. As input, the sketch receives a stream of strings and uses memory $O(LK)$ to keep information about frequent prefixes in the stream. A string x is processed in time $O(|x|)$, and a set of prefixes with relative frequency larger than $\varepsilon \geq 1/K$ can be retrieved in time $O(L/\varepsilon)$. In practice a good estimate for L can be easily obtained from an initial fraction of the stream. One such sketch can be used to keep information about the frequent prefixes in a stream of strings, and the information in two Prefix-Space-Saving sketches corresponding to streams generated by different distributions can be used to approximate their $\mathrm{L}^{\mathrm{p}}_\infty$ distance.

We begin by analyzing the error introduced by the sketch on the empirical $\mathrm{L}^{\mathrm{p}}_\infty$ distance between two distributions. Fix $\nu > 0$. Given a sequence $S = (x_1, \ldots, x_m)$ of strings from $\Sigma^\star$, for each prefix $x \in \Sigma^\star$ we denote by $S_\nu[x\Sigma^\star]$ the absolute frequency returned for prefix x by a Prefix-Space-Saving sketch $\texttt{PrefSpSv}(\lceil 2/\nu \rceil)$ that received S as input; that is, $S_\nu[x\Sigma^\star] = m\hat{f}_x$ if the pair $(x, \hat{f}_x)$ was returned by a retieval query with $\varepsilon = \nu/2$, and $S_\nu[x\Sigma^\star] = 0$ otherwise. Furthermore, $S(x\Sigma^\star)$ denotes the relative frequency of the prefix x

in S: $S_\nu(x\Sigma^\star) = S_\nu[x\Sigma^\star]/m$. The following lemma is a corollary of Theorem 3 in (Metwally et al., 2005).

Lemma 2 *For any $\nu > 0$ and any two sequences S and S', $|\mathrm{L}^{\mathrm{p}}_\infty(S, S') - \mathrm{L}^{\mathrm{p}}_\infty(S_\nu, S'_\nu)| \leq \nu$.*

3.2. Testing Dissimilarity with VC Bounds

Using the `PrefSpSv` sketch from previous section, we define a statistic $\hat{\mu}$ for confidently testing distribution dissimilarity. We will use this statistic to give a lower bound for $\mathrm{L}^{\mathrm{p}}_\infty(D, D')$ which holds with high probability. This yields the following accurate dissimilarity test: return DISTINCT when the lower bound is larger that 0, and return UNKNOWN otherwise.

Statistic $\hat{\mu}$ is an approximation to $\mathrm{L}^{\mathrm{p}}_\infty(D, D')$ computed as follows. Setup a sketch `PrefSpSv`$(\lceil 2/\nu \rceil)$ and let S_ν denote the contents of the sketch after processing m elements from the stream $(x_t)_{t>0}$. Similarly, define another `PrefSpSv`$(\lceil 2/\nu \rceil)$ sketch for the stream $(x'_t)_{t>0}$ and let S'_ν denote its contents after processing m' elements. The statistic $\hat{\mu}$ is the distance between the contents of both sketches: $\hat{\mu} = \mathrm{L}^{\mathrm{p}}_\infty(S_\nu, S'_\nu)$.

Note that the value of $\hat{\mu}$ depends on the quantities m and m' of strings processed. Furthermore, though $\hat{\mu}$ is deterministically computed from the streams, $\hat{\mu}$ can also be regarded as a random variable on the probability space of all possible streams generated by D and D'. Adopting this perspective, next proposition gives a deviation bound for $\hat{\mu}$ that holds with high probability.

Proposition 3 *Assume $\mathrm{L}^{\mathrm{p}}_\infty(D, D') = \mu^*$. Then, with probability at least $1 - \delta$, $\mu^* \geq \hat{\mu} - \nu - \sqrt{(8/M)\ln(16(m+m')/\delta)}$, where $M = mm'(\sqrt{m} + \sqrt{m'})^{-2}$.*

The result follows from an application of classical Vapnik–Chervonenkis bounds on the rate of uniform convergence of empirical probabilities to their expectations (Bousquet et al., 2004).

3.3. Testing Similarity with Bootstrapped Confidence Intervals

Now we derive a statistic for testing the similarity of two distributions over strings in a streaming setting. The test assumes that if $D \neq D'$, then necessarily $\mathrm{L}^{\mathrm{p}}_\infty(D, D') \geq \mu$ for some *known* parameter μ. Working under this hypothesis, the goal of the test is to certify (if possible) that $\mathrm{L}^{\mathrm{p}}_\infty(D, D') < \mu$. We do that by giving an upper bound on $\mathrm{L}^{\mathrm{p}}_\infty(D, D')$ that holds with high probability. If this upper bound is less that μ, the test can conclude with high confidence that $D = D'$.

The idea behind our test is to construct a bootstrapped confidence interval for the true distance $\mathrm{L}^{\mathrm{p}}_\infty(D, D')$. Basically, the *bootstrapping* methodology says that given a sample from some distribution, one can obtain several estimates of a statistic of interest by re-sampling from the original sample. Using these estimates one can, for example, compute a confidence interval for that statistic. Though re-sampling a sample given in stream form using sub-linear memory is in general impossible, we introduce a bootstrapping-like method that can be used whenever the statistic of interest can be computed from sketched data.

Given a positive integer r, our test will compute an upper bound for $\mathrm{L}^{\mathrm{p}}_\infty(D, D')$ using r^2 statistics computed as follows. Before reading the first element on stream $(x_t)_{t>0}$, build r identical sketches `PrefSpSv`$(\lceil 2/\nu \rceil)$ labeled by $[r] = \{1, \ldots, r\}$. Then, for each string x_t

in the stream, draw r indices $i_1, \ldots, i_r \in [r]$ uniformly at random with replacement, and for $1 \leq j \leq r$ insert x_t into the i_jth sketch (with possible repetitions). For $i \in [r]$ we let $B_{i,\nu}(x\Sigma^\star) = B_{i,\nu}[x\Sigma^\star]/m_i$ denote the the relative frequency assigned to prefix $x \in \Sigma^\star$ by the ith sketch, where m_i is the number of items that have been inserted in that particular sketch after processing m elements from the stream. Similarly, we define another set of sketches $B'_{i,\nu}$ with $i \in [r]$ for stream (x'_t).

With the sketches defined above, our algorithm can compute $\hat{\mu}_{i,j} = \mathrm{L}^{\mathrm{p}}_\infty(B_{i,\nu}, B'_{j,\nu})$ for $i, j \in [r]$; these are the r^2 statistics used to bound $\mathrm{L}^{\mathrm{p}}_\infty(D, D')$. The bound will be obtained as a corollary of the following concentration inequality on the probability that many of the $\hat{\mu}_{i,j}$ are much smaller than $\mathrm{L}^{\mathrm{p}}_\infty(D, D')$.

Fix some $0 < \alpha < 1$ and $\mu > 0$. Write $\mu^* = \mathrm{L}^{\mathrm{p}}_\infty(D, D')$. Assume m and m' elements have been read from streams (x_t) and (x'_t) respectively. For $i, j \in [r]$ let $Z_{i,j}$ be an indicator random variable of the event "$\mathrm{L}^{\mathrm{p}}_\infty(B_{i,\nu}, B'_{j,\nu}) \leq (1-\alpha)\mu$". Let $Z = \sum_{i,j\in[r]} Z_{i,j}$.

Theorem 4 *Suppose $\mu \leq \mu^*$. Then, for any $0 < \eta < 1$, $\alpha > 8\nu/\mu$, and m, m' large enough,*

$$\mathbb{P}[Z \geq \eta r^2] \leq \left(4 + 400\frac{(m+m')^2}{\eta^2 r}\right) \exp\left(-2M \min\left\{c_1(\alpha\mu - 8\nu)^2, c_2\frac{(\alpha\mu - 8\nu)^4}{(16\nu)^2}\right\}\right) \ ,$$

where $M = mm'(\sqrt{m}+\sqrt{m'})^{-2}$, $c_0 = 384$, $c_1 = 2/(1+\sqrt{c_0})$, and $c_2 = (1-c_1)^2/(c_1+\sqrt{c_1})^2$.

The main difficulty in the proof of this theorem is the strong dependence between random variables $Z_{i,j}$, which precludes straightforward application of usual concentration inequalities for sums of independent random variables. In particular, there are two types of dependencies that need to be taken into account: first, if $B_{i,\nu}$ is a bad bootstrap sample because it contains too many copies of some examples, then probably another $B_{j,\nu}$ is bad because it has too few copies of these same examples; and second, if $\hat{\mu}_{i,j}$ is a far-off estimate of μ^*, then it is likely that other estimates $\hat{\mu}_{i,j'}$ are also bad estimates of μ^*. Our proof tackles the latter kind of dependency via a decoupling argument. To deal with the former dependencies we use the Efron–Stein inequality to bound the variance of a general function of independent random variables.

From this concentration inequality, upper bounds for μ^* in terms of the statistics $\hat{\mu}_{i,j}$ can be derived. Let us write $\hat{\mu}_1 \leq \ldots \leq \hat{\mu}_{r^2}$ for the ascending sequence of statistics $\{\hat{\mu}_{i,j}\}_{i,j\in[r]}$. Each of these statistics yields a bound for μ^* with different accuracy. Putting these bounds together one gets the following result.

Corollary 5 *With probability at least $1-\delta$, if m, m' are large enough then it holds that*

$$\mu^* \leq \min_{1\leq k\leq r^2}\left\{\hat{\mu}_k + 8\nu + \max\left\{\sqrt{\frac{1}{2c_1 M}\ln\frac{K_k}{\delta}}, \sqrt[4]{\frac{(16\nu)^2}{2c_2 M}\ln\frac{K_k}{\delta}}\right\}\right\} \ ,$$

where $K_k = 4r^2 + 400(m+m')^2 r^5/k^2$.

4. Adaptive State-Merging with Streaming Data

In this section we present our algorithm for learning distributions over strings from a stream. The algorithm learns a PDFA by adaptatively performing statistical tests in order to discover new states in the automata and merge similar states. The procedure requires as input a guess on the number of states in the target and its distinguishability parameter. We will show that when this guesses are accurate, then the algorithm is in fact a PAC-learner for PDFA in the streaming setting.

Here we assume the distribution generating the stream is stationary. The algorithm will read successive strings over Σ from a data stream and, after some time, output a PDFA. We will show that if the stream comes from a PDFA with at most n states and distinguishability larger than μ, then the output will be accurate with high probability.

We begin with an informal description of the algorithm, which is complemented by the pseudocode in Algorithm 1. The algorithm follows a structure similar to other state-merging algorithms, though here tests to determine similarity between states are performed adaptively as examples arrive. Furthermore, Prefix-Space-Saving sketches are used in order to keep in memory only the relevant parts of the observed sample needed for testing purposes.

Algorithm 1: `ASMS` procedure

Input: Parameters $n, \mu, \Sigma, \varepsilon, \delta, L, r$
Data: A stream of strings $x_1, x_2, \ldots \in \Sigma^\star$
Output: A hypothesis H

initialize H with safe q_λ;
foreach $\sigma \in \Sigma$ **do**
  add a candidate q_σ to H;
  $t^0_\sigma \leftarrow 0$, $t^s_\sigma \leftarrow 128$, $t^u_\sigma \leftarrow (64n|\Sigma|/\varepsilon)\ln(2/\delta')$, $i_\sigma \leftarrow 1$;
end
foreach *string* x_t *in the stream* **do**
  foreach *decomposition* $x_t = wz$, *with* $w, z \in \Sigma^\star$ **do**
    if q_w *is defined* **then**
      add z to $\hat{S}_w$;
      if q_w *is a candidate* **and** $|\hat{S}_w| \geq t^s_w$ **then** call `STest`(q_w, δ_{i_w});
    end
  end
  foreach *candidate* q_w **do**
    if $t^u_w \geq t$ **then**
      if $|S_w|/(t - t^0_w) < (3\varepsilon)/(8n|\Sigma|)$ **then** declare q_w insignificant;
      else $t^u_w \leftarrow t^u_w + (64n|\Sigma|/\varepsilon)\ln 2$;
    end
  end
  if H *has more than* n *safes* **or** *there are no candidates left* **then return** H;
end

Algorithm 2: `STest` procedure

Input: A candidate q_w

foreach *safe* $q_{w'}$ *not marked as distinct from* q_w **do**
  $\hat{\mu}_\ell \leftarrow \mathcal{L}(\hat{S}_w, \hat{S}_{w'}, \delta_{i_w}/2)$;
  $\hat{\mu}_u \leftarrow \mathcal{U}(\hat{S}_w, \hat{S}_{w'}, \delta_{i_w}/2)$;
  if $\hat{\mu}_\ell > 0$ **then** mark $q_{w'}$ as distinct from q_w;
  else if $\hat{\mu}_u < \mu$ **then** merge q_w to $q_{w'}$ and **return**;
  else $t^s_w \leftarrow 2 \cdot t^s_w$, $i_w \leftarrow i_w + 1$;
end
if q_w *is marked distinct from all safes* **then**
  promote q_w to safe;
  foreach $\sigma \in \Sigma$ **do**
    add a candidate $q_{w\sigma}$ to H;
    $t^0_{w\sigma} \leftarrow t$, $t^s_{w\sigma} \leftarrow 128$, $t^u_{w\sigma} \leftarrow t + (64n|\Sigma|/\varepsilon)\ln(2/\delta')$, $i_{w\sigma} \leftarrow 1$;
  end
end

Our algorithm requires some parameters as input: the usual accuracy ε and confidence δ, a finite alphabet Σ, a number of states n, a distinguishability parameter μ, an upper bound L on the expected length of strings generated from any state, and the size r of the bootstrapping sketch. The algorithm, which is called $\texttt{ASMS}(n, \mu, \Sigma, \varepsilon, \delta, L, r)$, reads data from a stream of strings over Σ. At all times it keeps a hypothesis represented by a directed graph where each arc is labeled by a symbol in Σ. The nodes in the graph (also called *states*) are divided into three kinds: *safes*, *candidates*, and *insignificants*, with a distinguished *initial* safe node denoted by q_λ. The arcs are labeled in such a way that, for each $\sigma \in \Sigma$, there is at most one arc labeled by σ leaving each node. Candidate and insignificant nodes have no arc leaving them. To each string $w \in \Sigma^\star$ we may be able to associate a state by starting at q_λ and successively traversing the arcs labeled by the symbols forming w in order. If all transitions are defined, the last node reached is denoted by q_w, otherwise q_w is undefined – note that by this procedure different strings w and w' may yield $q_w = q_{w'}$.

For each state q_w, `ASMS` keeps a multiset S_w of strings. These multisets grow with the number of strings processed by the algorithm and are used to keep statistical information about the distribution D_{q_w}. In fact, since the algorithm only needs information from frequent prefixes in the multiset, it does not need to keep the full multiset in memory. Instead, it uses a set of sketches to keep the relevant information for each state. We use $\hat{S}_w$ to denote the information contained in these sketches associated with state q_w. This set of sketches contains a $\texttt{PrefSpSv}(\lceil 64/\mu \rceil)$ sketch to which the algorithm inserts the suffix of every observed string that reaches q_w. Furthermore, $\hat{S}_w$ contains another r sketches of the form $\texttt{PrefSpSv}(\lceil 64/\mu \rceil)$; these are filled using the bootstrapping scheme described in Section 3.3. We use $|\hat{S}_w|$ to denote the number of strings inserted into the sketches associated with state q_w.

Execution starts from a graph consisting of a single safe node q_λ and several candidates q_σ, one for each $\sigma \in \Sigma$. Each element x_t in the stream is then processed in turn: for each

prefix w of $x_t = wz$ that leads to a node q_w in the graph, the corresponding suffix z is added to the state's sketch. During this process, similarity and insignificance tests are performed on candidate nodes in the graph following a certain schedule; the former are triggered by the sketch's size reaching a certain threshold, while the latter occur at fixed intervals after the node's creation. In particular, t_w^0 denotes the time state q_w was created, t_w^s is a threshold on the size $|\hat{S}_w|$ that will trigger the next round of similarity tests, and t_w^u is the time the next insignificance test will occur. Parameter i_w keeps track of the number of similarity tests performed for state q_w. These numbers are used to adjust confidences in tests by using the convergent series defined by $\delta_i = 6\delta'/\pi^2 i^2$, where $\delta' = \delta/2|\Sigma|n(n+1)$.

Insignificance tests are used to check whether the probability that a string traverses the arc reaching a particular candidate is below a certain threshold; it is known that these transitions can be safely ignored when learning a PDFA. Similarity tests use statistical information from a candidate's sketch to determine whether it equals some already existing safe or it is different from all safes in the graph. Pseudocode for the similarity test used in `ASMS` is given in Algorithm 2. It uses two functions $\mathcal{L}$ and $\mathcal{U}$ that given the sketches associated with two states compute lower and upper bounds to the true distance between the distributions on the states that hold with a given confidence. These functions can be easily derived from Proposition 3 and Corollary 5. We assume that when the size assumptions in Corollary 5 are not satisfied, $\mathcal{U}$ returns 1.

A candidate node will exist until it is promoted to safe, merged to another safe, or declared insignificant. When a candidate is merged to a safe, the sketches associated with that candidate are discarded. The algorithm will end whenever there are no candidates left, or when the number of safe states surpasses the given parameter n.

4.1. Analysis

Now we proceed to analyze the `ASMS` algorithm. Our first result is about memory and number of examples used by the algorithm. Note the result applies to any stream generated i.i.d. from a probability distribution over strings with expected length L, not necessarily a PDFA.

Theorem 6 *The following hold for any call to* $\texttt{ASMS}(n, \mu, \Sigma, \varepsilon, \delta, L, r)$*:*

1. *The algorithm uses memory* $\widetilde{O}(n|\Sigma|Lr/\mu)$
2. *The expected number of elements read from the stream is at most* $\widetilde{O}(n^2|\Sigma|^2/\varepsilon\mu^2)$
3. *Each item in the stream is processed in* $\widetilde{O}(rL)$ *expected amortized time*

We want to remark here that item (3) above is a direct consequence of the scheduling policy used by `ASMS` in order to perform similarity tests adaptatively. The relevant point is that the ratio between executed tests and processed examples is $O(\log t/t) = o(1)$. In fact, by performing tests more often while keeping the tests/examples ratio to $o(1)$, one could obtain an algorithm that converges slightly faster, but has a larger (though still constant) amortized processing time per item.

Our next theorem is a PAC-learning result. It says that if the stream is generated by a PDFA and the parameters supplied to `ASMS` are accurate, then the resulting hypothesis

will have small error with high probability when transition probabilities are estimated with enough accuracy. Procedures to perform this estimation have been analyzed in detail in the literature. Furthermore, the adaptation to the streaming setting is straightforward. We use an analysis from (Palmer, 2008) in order to prove our theorem.

Theorem 7 *Given a stream generated from a PDFA T with n states, distinguishability μ, and expected length from all states L, let H denote the hypothesis returned by a call to* $\texttt{ASMS}(n', \mu', \Sigma, \varepsilon, \delta, r, L')$ *with $n' \geq n$, $\mu' \leq \mu$, and $L' \geq L$. If $\tilde{H}$ denotes a PDFA obtained from H by estimating its transition probabilities with $\widetilde{O}(n^4|\Sigma|^4/\varepsilon^3)$ examples, then with probability at least $1-\delta$ we have $\mathrm{L}_1(T, \tilde{H}) \leq \varepsilon$.*

The proof of Theorem 7 is similar to those learning proofs in Clark and Thollard (2004); Palmer and Goldberg (2007); Castro and Gavaldà (2008). Therefore, we only discuss in detail those lemmas involved in the proof which are significantly different from the batch setting. In particular, we focus on the effect of the sketch on the estimations used in the test, and on the adaptive test scheduling policy. The rest of the proof is quite standard: first show that the algorithm recovers a transition graph isomorphic to a subgraph of the target containing all relevant states and transitions, and then bound the overall error in terms of the error in transition probabilities. We note that by using a slightly different notion of unsignificant state and applying a smoothing opertion after learning a PDFA, our algorithm could also learn PDFA under the more strict KL divergence.

The next two lemmas establish the correctness of the structure recovered: with high probability, merges and promotions are correct, and no non-insignificant candidate nodes are marked as insignificant.

Lemma 8 *With probability at least $1 - n^2|\Sigma|\delta'$, all transitions between safe states are correct.*

Following Palmer and Goldberg (2007), we say a state in a PDFA is *insignificant* if a random string passes through that state with probability less than $\varepsilon/2n|\Sigma|$; the same applies to transitions. It can be proved that a subgraph from a PDFA that contains all its non-insignificant states and transitions fails to accept a set of strings accepted by the original PDFA of total probability at most $\varepsilon/4$.

Lemma 9 *With probability at least $1 - n|\Sigma|\delta'$ no significant candidate will be marked insignificant and all insignificant candidates with probability less than $\varepsilon/4n|\Sigma|$ will be marked insignificant during its first insignificance test.*

Though the algorithm would be equally correct if only a single insignificance test was performed for each candidate state, the scheme followed here ensures the algorithm will terminate even when the distribution generating the stream changes during the execution and some candidate that was significant w.r.t. the previous target is insignificant w.r.t. to the new one.

With the results proved so far we can see that, with probability at least $1 - \delta/2$, the set of strings in the support of T not accepted by H have probability at most $\varepsilon/4$ w.r.t. D_T. Together with the guarantees on the probability estimations of $\tilde{H}$ provided by Palmer (2008), we can see that with probability at least $1 - \delta$ we have $\mathrm{L}_1(T, \tilde{H}) \leq \varepsilon$.

Structure inference and probabilities estimation are presented here as two different phases of the learning process for clarity and ease of exposition. However, probabilities could be incrementaly estimated during the structure inference phase by counting the number of times each arc is used by the examples we observe in the stream, provided a final probability estimation phase is run to ensure that probabilites estimated for the last added transitions are also correct.

5. Experiments

This section describes experiments performed with a proof-of-concept implementation of our algorithm. The main goal of our experiments was to showcase the speed and memory profile of our algorithm, and the benefits of using a bootstrap test versus a VC-based test.

Data used for the experiments was generated using a PDFA over the Reber grammar, a widely used benchmark in the Grammatical Inference community for regular language learning algorithms (de la Higuera, 2010). Table 1 summarizes the basic parameters of this target. We set $\varepsilon = 0.1$ and $\delta = 0.05$ in our experiments experiments.

Our experiment compares the performance of `ASMS` using a test based on VC bounds against the same algorithm using a test based on bootstrapping. We run `ASMS` with the true parameters for the Reber grammar with an input stream generated by this PDFA. In the algorithm using a bootstrapped test we set $r = 10$.

In Figure 1 we plot the number of safes and candidates in the hypothesis against number of examples processed for both executions. We observe that the test based on bootstrapping identified all six safes and performed all necessary merges about half the examples required by the test based on VC bounds.

Table 2 shows processing time and memory used by both executions, where we can see that, as expected, the algorithm using the bootstrap sketch requires more memory and processing time. We also note that without using the boostrapped test, `ASMS` is extremely fast and has a very low memory profile.

We note that the number of examples consumed by the algorithm until convergence is at least one order of magnitude larger than reported sample sizes required for correct graph identification in batch state-merging algorithms (Carrasco and Oncina, 1994; Castro and Gavaldà, 2008). However, one has to keep in mind that our algorithm can only make one pass over the sample and is not allowed to store it. Thus, every time a state is promoted from candidate to safe, $|\Sigma|$ candidates attached to the new safe are created, each having an empty set of sketches. Among these new candidates, the non-insignificant need to have their sketches populated with a large enough sample before a merging or promoting decision can be made; this population will happen at a rate proportional to the probability of traversing the particular edge that reaches each particular candidate. Overall, it is clear that the restrictions imposed by the streaming setting must introduce a non-negligible overhead in the minimum number of examples required for correct graph identification in comparison with the batch setting. Though it seems hard to precisely quantify this overhead, we believe that our algorithm may be working within a reasonable factor of this lower bound. Furthermore, we would like to note that in this particular example our algorithm used one order of magnitude less examples than $n^2|\Sigma|^2/\varepsilon\mu^2 = 225000$, the asymptotic upper bound given in Theorem 6.

Table 1: Parameters of the Reber Grammar

$\|\Sigma\|$	n	μ	L
5	6	0.2	7.6

Table 2: Experimental results with `ASMS`

	Examples	Memory (MiB)	Time (s)
VC	57617	6.1	3.2
BS	23844	53.7	30.8

6. Future Work

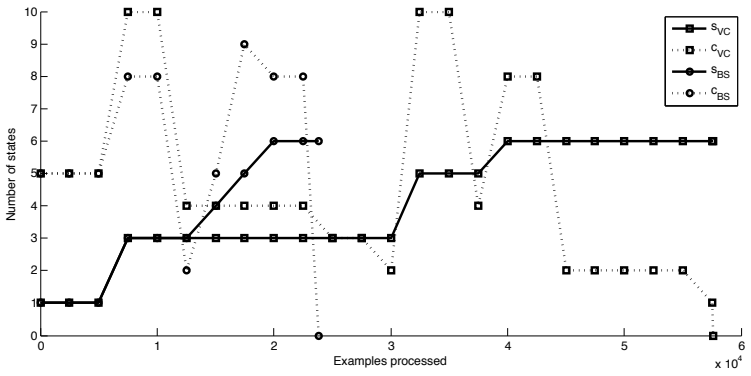
As future work, we would like to parallelize our implementation and perform large-scale experiments with real data. This will be very important in order to exploit the full benefits of our approach.

Our algorithm can be extended to incorporate a change detection module. This would allow the algorithm to adapt to changes in the target distribution and modify the relevant parts of its current hypothesis. This can be complemented with an efficient search strategy to determine the true number of states and distinguishability of the target PDFA generating the stream, which may also change over time. Altogether, these extensions yield a complete solution to the problem of learning a distribution over strings that changes over time in the data streams framework.

Acknowledgments

The authors would like to thank the reviewers for many useful comments. This work is partially supported by MICINN projects TIN2011-27479-C04-03 (BASMATI) and TIN-2007-66523 (FORMALISM), by SGR2009-1428 (LARCA), and by EU PASCAL2 Network of Excellence (FP7-ICT-216886). B. Balle is supported by an FPU fellowship (AP2008-02064) from the Spanish Ministry of Education

Figure 1: Evolution of the number of states



References

C. Aggarwal, editor. *Data Streams – Models and Algorithms*. Springer, 2007.

A. Bifet. *Adaptive Stream Mining: Pattern Learning and Mining from Evolving Data Streams*. IOS Press - Frontiers of Artificial Intelligence Series and Applications, 2010.

O. Bousquet, S. Boucheron, and G. Lugosi. Introduction to statistical learning theory. *Advanced Lectures on Machine Learning*, 2004.

R. C. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *International Colloquium on Grammatical Inference (ICGI)*, 1994.

J. Castro and R. Gavaldà. Towards feasible PAC-learning of probabilistic deterministic finite automata. In *International colloquium on Grammatical Inference (ICGI)*, 2008.

A. Clark and F. Thollard. PAC-learnability of probabilistic deterministic finite state automata. *Journal of Machine Learning Research*, 2004.

C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

J. Gama. *Knowledge Discovery from Data Streams*. Taylor and Francis, 2010.

O. Guttman, S. V. N. Vishwanathan, and R. C. Williamson. Learnability of probabilistic automata via oracles. In *Conference on Algorithmic Learning Theory (ALT)*, 2005.

P. Hall. *The Bootstrap and Edgeworth expansion*. Springer, 1997.

X. Lin and Y. Zhang. Aggregate computation over data streams. In *Asian-Pacific Web Conference (APWeb)*, 2008.

A. Metwally, D. Agrawal, and A. Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International Conference on Database Theory (ICDT)*, 2005.

S. Muthukrishnan. Data streams: algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 2005.

N. Palmer and P. W. Goldberg. PAC-learnability of probabilistic deterministic finite state automata in terms of variation distance. *Theor. Comput. Sci.*, 2007.

N. J. Palmer. *Pattern Classification via Unsupervised Learners*. PhD thesis, University of Warwick, 2008.

D. Ron, Y. Singer, and N. Tishby. On the learnability and usage of acyclic probabilistic finite automata. *Journal of Computing Systems Science*, 1998.