PDFA Distillation with Error Bound Guarantees

Baumgartner, Robert; Verwer, Sicco

DOI
10.1007/978-3-031-71112-1_4

Publication date
2024

Document Version
Final published version

Published in
Implementation and Application of Automata

PDFA Distillation with Error Bound Guarantees

Robert Baumgartner(✉) and Sicco Verwer

Technical University of Delft, Delft, Netherlands
{r.baumgartner-1,s.e.verwer}@tudelft.nl

Abstract. Active learning algorithms to infer probabilistic finite automata (PFA) have gained interest recently, due to their ability to provide surrogate models for some types of neural networks. However, recent approaches either cannot guarantee determinism, which makes the automaton harder to understand and compute, or they rely on techniques that bound errors on individual transitions. In this work we propose a derivative of the recent $L^{\#}$ algorithm to learn deterministic PFA (PDFA) from systems returning a distribution over a set of tokens given an input string. Along with determinism, we can give error bounds on probabilities assigned to whole strings with an easy to understand approach. We show formal correctness of our algorithm and test it on neural networks trained to model three datasets from computer- and network-systems respectively. We show that the algorithm can learn the network's behaviour closely, and provide an example application of how the model can be used to interpret the network. We note that our approach is in theory applicable in general to learn deterministic weighted finite automata. We provide the source code of our algorithm and relevant scripts on our public repository.

Keywords: Active Automata Learning · PDFA distillation · Explainable AI

1 Introduction

Active learning of automata has had its advent in the work of Dana Angluin, who introduced the L* algorithm [1] to learn deterministic finite automata (DFA) from an unknown target system. Multiple derivatives of the L* have come out since then, optimizing one or more parts of the algorithm (for more details, see e.g. [7]). Starting with the work of Weiss et al. [20] these algorithms have gained interest again by distilling DFA from a neural network (NN) trained to recognize an unknown target language. Follow up work constitutes e.g. of the work of Mayr and Yovine [11], who introduce a derivative of the L* they call bounded L* and show its properties to be PAC-bounded. Muškardin et al. [14] investigate the effect of the counterexample search strategy on the resulting automata.

While training neural networks to recognize unknown (regular) languages provides an interesting theoretical basis for deep learning, these types of networks

S. Z. Fazekas (Ed.): CIAA 2024, LNCS 15015, pp. 51–65, 2024.
https://doi.org/10.1007/978-3-031-71112-1_4

are rarely found outside academic environments. More interesting networks are language models: Given a set of possible tokens Σ, the network returns conditional probabilities of the form $P(a|x)$, where $a \in \Sigma$ and x is a string in Σ^*, the set of all possible strings over Σ. In words, $P(a|x)$ models the probability of a token a to occur after having seen the substring x. Weiss et al. [19] proposed an adaptation of the L* algorithm to learn PDFA from such networks. A similar approach is taken by Eyraud and Ayache [4], which use a spectral approach from [2] to extract weighted finite automata (WFA) from the real-valued matrix. Okudono et al. [15] focus their work on the counterexample search. While powerful, a drawback of the spectral approach is that the resulting automata are not deterministic. A slightly new approach is taken in [13] and [12]. Unlike the previous approaches these algorithms use an observation tree and minimize it via merging states. To this end they quantize the distributions of the states and define congruence over strings, which leads them to different notions of similarity of states.

In our work we build on the recent $L^\#$ algorithm (Vandraager et al., [18]), an algorithm designed to learn Mealy-automata, similar to L*. Unlike L* it builds a tree of observations and identifies states by searching for distinguishing criteria. We build an observation tree modeling probability distributions and introduce a simple notion of state similarity to generalize the model. Unlike $L^\#$ we do not have to distinguish a state from the rest, but can use the numerical output of PDFAs and choose states that minimize induced errors, thus relaxing the strictness of our merge-requirements. Our similarity measure guarantees that, given a parameter $\mu > 0$ as input, for each string x that has been seen by the algorithm that $|P(x) - \pi(x)| \leq \mu$, where $P(x)$ is the assigned probability by the network, and $\pi(x)$ is the probability of x assigned by our inferred PDFA. We further support this notion of proximity between the network's output and the PDFA's output via our proposed equivalence test, giving us PDFAs that mimic the network with intuitive requirements. We show the capability of our algorithm on three datasets, namely the CTU-13 dataset, the BGL dataset, and the HDFS dataset. We train a recurrent neural network (RNN) on each of these respectively, and then distill PDFA varying μ to investigate the effect on the distilled models, and show an example application of the surrogate model. For reproducibility and verification purposes we provide the source code of our method on our public repository[1].

2 Background

2.1 PDFA

A PDFA is an automaton defined over a tuple $\mathcal{A} = \{q_0, Q, \Sigma, \tau, \pi\}$. In this tuple, Q denotes a set of states, and q_0 is a special initial state. The alphabet Σ is a finite set of tokens that the PDFA can accept as input. We denote a as an individual token in Σ, and x is an arbitrary string from the set of all possible

[1] https://github.com/tudelft-cda-lab/FlexFringe.

strings over Σ, denoted by Σ^*. We write in short $|x|$ for the length of string x, and λ is the special empty string with length $|\lambda| = 0$.

An input string $x = a_0a_1...a_n$ traverses $\mathcal{A}$ via the transition function $\tau : Q \times \Sigma \to Q$ recursively: $\tau(q, \lambda) = q$ and $\tau(q, ax) = \tau(\tau(q, a), x)$.... Note that we introduced shorthand notation for $\tau(q, x)$ as the recursive traversal through $\mathcal{A}$ with a string x. A state q' is reachable from state q iff $\exists x : q' = \tau(q, x)$. We write shorthand $Q_{\tau(q)}$ to denote the set of all states that are reachable from state q, and we denote by $X_{\tau(q)}$ the set of strings needed to reach them starting in state q. Traversing the automaton with a string x results in a sequence of states $q^0q^1...q^{n-1}$ being visited. We call this sequence of states the trace of x and write $T(x)$.

Lastly, the mapping $\pi : Q \times \Sigma \to [0, 1]$, $\pi : Q \to [0, 1]$ maps state-symbol pairs and states to probabilities. A PDFA requires $\forall q \in Q$: $\sum_{a \in \Sigma} \pi(q, a) + \pi(q) = 1$. We call $\pi(q)$ the stopping probability of state q, modeling the probability of a string to reach state q and end there. Given an input string $x = a_0a_1...a_{m-1}$ and its associated trace of states $T(x) = q^0q^1...q^{m-1}q^m$ we can compute the probability of string x to occur via $\pi(x) = \prod_{i=0}^{m-1} \pi(q^i, a_i) \cdot \pi(q^m)$. Figure 1 depicts an observation tree and its minimal PDFA.

2.2 Observation Tree, Closed PDFA, and State Merging

We say that a PDFA is an observation tree $\mathcal{OT}$ if for each state $q \in \mathcal{OT}$ there exists a unique string x_q^A s.t. $q = \tau(x_q^A)$, and q is only reachable from q_0 by this string. We call x_q^A the access string of q, and $x_{q_0}^A = \lambda$. Contrary to the observation tree, we call a PDFA $\mathcal{A}$ consisting of state set Q closed iff $\forall q \in Q$, $\forall a \in \Sigma$: $\tau(q, a) \in Q$.

In this work we build observation trees and minimize them via state merging techniques [7]. Building deterministic models furthermore requires a determinization process whenever two states are merged, subsequently merging state pairs such that $\tau(q, a)$ is uniquely identified $\forall q \in Q, \forall a \in \Sigma$. State merging induces a mapping in between the observation tree and the PDFA $\mathcal{A}$. In Fig. 1 we show an observation tree along with a related closed PDFA obtained via 1. Merge q_1 into q_0, which merges q_2 into q_3 as part of the determinization procedure. 2. Merge q_4 and q_5 into q_3, in arbitrary order.

3 Learning Algorithm

Our algorithm leans heavily on the works of Vandraager et al. [18]. We define a teacher, which is an abstraction of the system-under-learning (SUL), and a learner, an abstraction of the learning algorithm. The learner can ask the teacher two question: Firstly a membership query $\mathcal{MQ}(xa)$ for string xa, to which the teacher replies with a conditional probability $P(a|x), a \in \Sigma_\xi$. Here, we enhanced the set Σ with a unique stopping symbol ξ indicating the end of a sequence[2].

[2] The presence of such a symbol is a reasonable assumption. E.g. in natural language processing this is commonly referred to as the <EOS> (end-of-sentence) symbol.

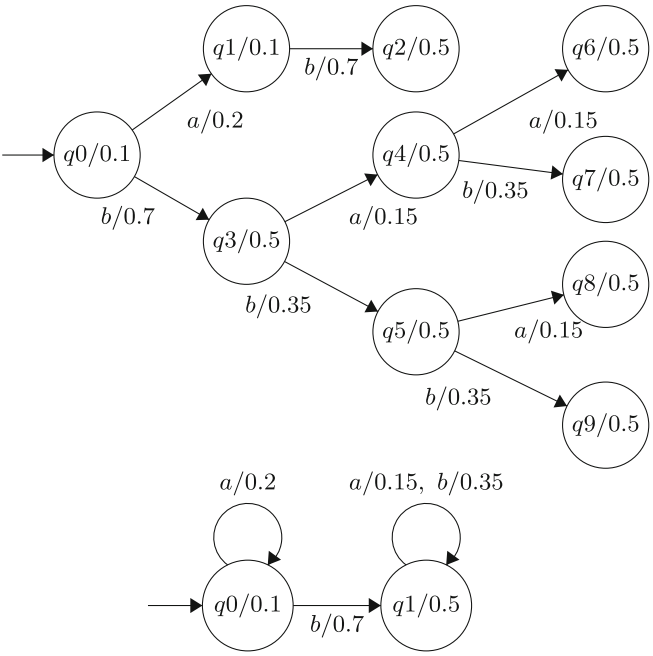


Fig. 1. An observation tree (above) and a corresponding (closed) PDFA.

Secondly, the learner can ask an equivalence query $\mathcal{EQ}(\mathcal{T}, \mathcal{H})$ between the target SUL $\mathcal{T}$ and the current hypothesis $\mathcal{H}$. The teacher will respond with either a 'yes', meaning the systems are behaving similar enough, or a 'no' along with a counterexample string x_{cex} which violates the error bound.

The learner has two main routines: In the first routine it tries to find a hypothesis candidate for target $\mathcal{T}$. Whenever it finds a valid hypothesis candidate $\mathcal{H}$ the second routine starts, either validating $\mathcal{H}$ and terminating or processing the counterexample returned by the teacher. The learner repeats these two routines until it has found a hypothesis that can pass $\mathcal{EQ}(\mathcal{T}, \mathcal{H})$. We describe the two routines in the following.[3]

3.1 Finding a Hypothesis

In order to find a hypothesis we employ the red-blue-framework [9], which distinguishes a core of red states, which are considered states of the final automaton, blue states and white states. Blue states are states q' who have a transition $q' = \tau(q, a)$ s.t. q' is not red but q is. States that are neither red nor blue are

[3] Because we describe our algorithm at all stages, it was not always clear on how we should refer to the individual states and nodes of the automaton. For simplicity we decided to stick to the word 'state' meaning both state and node from here on, as they are mostly interchangeable for us.

Algorithm 1. Extend fringe

Input: State q, alphabet Σ
  for all $a \in \Sigma$ **do**
    Create state q' satisfying $q' = \tau(q, a)$
    $InitializeState(q')$ ▷ Algorithm 2
    $\mathcal{F}_n \leftarrow \mathcal{F}_n \cup \{q'\}$
  end for

Algorithm 2. Initialize state

Input: State q, access to target $\mathcal{T}$ via teacher
  Save $P(x_q^A)$ in state
  $\pi(q) \leftarrow \mathcal{MQ}(x_q^A \xi)$
  for all $a \in \Sigma$ **do**
    $\pi(q, a) \leftarrow \mathcal{MQ}(x_q^A a)$
  end for

white. Initially, the observation tree $\mathcal{OT}$ consists only of one red state q_0. The learner initializes q_0 via Algorithm 2 and creates a set of blue states around q_0 via Algorithm 1. Then the learner attempts to minimize $\mathcal{OT}$ via state merging.

We call a pair of a red state q_r and a blue state q_b consistent under $\mu \in [0, 1)$ iff it holds that

$$d(q_r, q_b) = \left| \frac{\pi(x_{q_b}^A)}{\pi(q_b)} \cdot \pi(q_r) - P(x_{q_b}^A) \right| \leq \mu. \tag{1}$$

Note that we introduced a distance between q_r and q_b here, written $d(q_r, q_b)$ in short. The term $\frac{\pi(x_{q_b}^A)}{\pi(q_b)} \cdot \pi(q_r)$ represents $\pi(x_{q_b}^A)$ if the merge were to happen. We call q_r and q_b mergeable iff $\forall q_b' \in Q_{\tau(q_b)}$ and their respective set of strings $X_{\tau(q_b)}$ it holds that either $\tau(q_r, x')$, $x' \in X_{\tau(q_b)}$ is not defined, or $q_r' = \tau(q_r, x')$ and $d(q_r', q_b') \leq \mu$. In words: When we want to merge two states the inequality (1) needs to hold for all states that are merged through determinization as well. However, with these merge requirements we are able to guarantee that $|P(x) - \pi(x)| \leq \mu$ for each string x for which $\tau(x)$ is defined on the observation tree.

In order to find a hypothesis the learner continues growing the hypothesis until it has found a set of red nodes that form a closed PDFA. In every step the learner fixes $\mathcal{C}$, the current set of red states, and $\mathcal{F}$, the current set of blue states, and compares each of the blue states with each of the red states. If a blue state is consistent with a red state, the blue state will be merged into the red state. If a blue state q_b can be merged with multiple red states, it will be merged into the red state q_r satisfying $q_r = argmin_{q_r' \in \mathcal{C}}\{d(q_r', q_b)\}$. A blue state that cannot be merged with any red state will turn red at the end of the iteration, and will be extended via Algorithm 1. If during an iteration all blue states were able to merge into a red state, then the learner has found a closed PDFA. The entire subroutine is depicted in Algorithm 3.

Algorithm 3. Find hypothesis

Input: Unmerged observation tree with root state q_0, alphabet Σ, threshold μ
Output: Hypothesis $\mathcal{H}$

```
Initialize H with root state q_0
while Hypothesis not found do
    Fix red states in set C, blue states in set F
    R ← ∅
    for all Blue states q_b ∈ F do
        s_min ← 1, m ← NULL                          ▷ m is placeholder for best merge
        for all Red states q_r ∈ C do
            if Consistent(q_r, q_b) and Score(q_r, q_b) < s_min then     ▷ See Section 3.1
                s_min ← Score(q_r, q_b)
                m ← (q_r, q_b)
            end if
        end for
        if m not NULL then
            Merge q_b into q_r
        else                                          ▷ No merge found for q_b
            R ← R ∪ {q_b}
        end if
    end for
    if R is not empty then
        for all q_b ∈ R do
            Mark q_b red
            ExtendFringe(q_b)                         ▷ Algorithm 1
        end for
    else
        Return H
    end if
end while
```

3.2 Counterexample Search and Processing

Once the learner found a valid hypothesis, it asks the teacher for an equivalence query $\mathcal{EQ}(\mathcal{T}, \mathcal{H})$. The query can have two possible outcomes: Either the teacher deems $\mathcal{T}$ and $\mathcal{H}$ sufficiently close, or it rejects $\mathcal{H}$ and provides a counterexample string x_{cex} for which the following requirement does not hold. In accordance with our merging procedure we require closeness for $\mathcal{T}$, modeling $P(x)$, and $\mathcal{H}$, modeling $\pi(x)$, via $|P(x) - \pi(x)| \leq \mu$.

Finding a counterexample can mean one of two possible cases: In the first case the string x_{cex} has been seen before. In this case a merge between a red state q_r and a blue state q_b has been performed, and after the merge one or more states q' have been added s.t. q' was reachable by q at the time q' was created. In this case the merge of q_r and q_b has been wrong. In the second scenario x_{cex} has not been seen yet. Thus $\mathcal{H}$ does not have sufficient information, yet.

We opted for a simple strategy that can deal with both scenarios: Our learner remembers an inverse mapping from $\mathcal{H}$ to $\mathcal{OT}$. To deal with counterexamples we

first reset $\mathcal{H}$ to $\mathcal{OT}$, and then parse x_{cex} via $\tau(x_{cex})$. Whenever transitions are not defined on τ the learner creates the missing states, and initializes any new state q' through subroutines 1 and 2. Once the counterexample is processed the algorithm returns to finding a valid hypothesis. We show the counterexample processing subroutine in Algorithm 4, and the main loop of our algorithm in Algorithm 5.

Algorithm 4. Process Counterexample

Input: Observation tree starting in root node q_0, counterexample string x_{cex}
  $q' \leftarrow q_0$
  $m \leftarrow |x_{cex}| - 1$
  for i in $0...m$ **do** ▷ $x_{cex} = a^0 a^1 ... a^m$
    if $\tau(q', a^i)$ not defined **then**
      $ExtendFringe(q')$
    end if
    $q' \leftarrow \tau(q', a^i)$
  end for

Algorithm 5. Main routine

Input: Access to target $\mathcal{T}$ via teacher, alphabet Σ, error bound μ
Output: Hypothesis $\mathcal{H}$
  $Initialize(q_0)$
  $ExtendFringe(q_0)$ ▷ Algorithm 1
  while Hypothesis not found **do**
    $\mathcal{H} \leftarrow FindHypothesis(q_0, \Sigma, \mu)$
    Perform $\mathcal{EQ}(\mathcal{T}, \mathcal{H})$
    if Counterexample x_{cex} returned by $\mathcal{EQ}(\mathcal{T}, \mathcal{H})$ **then**
      Reset $\mathcal{H}$ ▷ q_0 holds observation tree again
      $ProcessCounterexample(q_0, x_{cex})$
    else
      Return $\mathcal{H}$
    end if
  end while

3.3 Practical Considerations

Equivalence Oracle. While practical in theory, in reality it is impossible to check $\mathcal{H}$ for each possible string x. Multiple search strategies exist to find counterexamples [14]. Here we opted for a simple solution, generating random strings over Σ^* assuming a uniform distribution over one and the maximum string length, and a uniform distribution over Σ for each token a of x. If a maximum number of strings has been suggested without finding a counterexample the oracle deems $\mathcal{H}$ and $\mathcal{T}$ equivalent.

Early Stopping. Depending on the complexity of the underlying problem the hypothesis $\mathcal{H}$ can grow to a very large model. In these cases it is desirable to have early stopping criteria. We set a limit n_{max} on the number of red states.

The first time the number of red states reaches n_{max} we *force-merge* the set of remaining blue states into the set of current red states: For each of the blue states q_b, find a red state q_r that minimizes $q_r = argmin_{q'_r \in C} d(q'_r, q_b)$ and merge q_b into q_r.

4 Correctness

Lemma 1. *Every iteration of FindHypothesis (Algorithm 3) over the current set of blue states, comparing them with the set of red states, either results in a complete basis $\mathcal{B}$, or it identifies a new red state, growing $\mathcal{H}$ by at least one new state.*

The proof of this Lemma is by design of the algorithm. It is important to note that every new red state q_r accepts at least its access string $x^A_{q_r}$ s.t. $P(x^A_{q_r}) = \pi(x^A_{q_r})$. Therefore, each identified red state increases the number of strings satisfying $|P(x) - \pi(x)| \leq \mu$ by at least one and therefore the quality of the result.

Lemma 2. *Every identified red state in the observation tree q directly corresponds to a state q' in target $\mathcal{T}$ s.t. $\pi(q, a) = \pi(q', a)$, $\forall a \in \Sigma$ and $\pi(q) = \pi(q')$.*

Proof. We prove by contradiction: Assume the learner identified a red state q' that is not part of $\mathcal{T}$. We further assume that no states of $\mathcal{OT}$ have been merged yet, i.e. $\mathcal{H}$ is an observation tree still. Then, $\exists a \in \Sigma_\xi$: $P(x^A_{q'}a) \neq \pi(x^A_{q'}a)$. By design however on $\mathcal{OT}$ models probabilities precisely, therefore this event cannot happen. The case for $\mathcal{H}$ with performed merges follows from the fact that our merge routine holds error bounds $\forall \mu \in [0, 1)$.

We have to note that assuming the target $\mathcal{T}$ to be a PDFA provides value by helping us in showing correctness of our algorithm. Many real-world applications however have underlying systems of higher complexity. In those cases the number of states to model the target system cannot easily be bound by a fixed number of states, but we know that the probability assigned to strings decreases monotonically in length: $P(x) \geq P(xa)$. Therefore it does make sense to define an upper bound on the length of the strings we want $\mathcal{T}$ to model, and then have the algorithm cover those cases. We now focus on the *ProcessCounterexample* routine.

Lemma 3. *Assuming the teacher rejected $\mathcal{H}$ along with counterexample string x_{cex}, there are only two possible reasons:*

1. *x_{cex} has been seen before by the learner, in which case the learner merged two inconsistent states.*
2. *x_{cex} or a substring of it have not yet been seen by the learner. In this case there exist states in $\mathcal{T}$ that are not yet part of $\mathcal{H}$ and its observation tree.*

As explained in Sect. 3.1, our merge routine ensures that $\forall x \in \Sigma^*$ s.t. $\tau(x)$ is well defined on the unmerged observation tree $\mathcal{OT}$ it is $|P(x) - \pi(x)| \leq \mu$ at the time of a merge. Therefore, case 1 of Lemma 3 can only happen on states the learner added to $\mathcal{H}$ after a merge has happened. Resetting $\mathcal{H}$ to the observation tree will solve this problem, since the merge check will ensure that this merge will not happen again. In case 2 of Lemma 3 resetting to the observation tree and adding x_{cex} will create the unknown states necessary to accept x_{cex}. In this case we either add new states to the automaton, or we remove one or more wrongly performed merges on $\mathcal{H}$. Either case is going to increase the number of accepted traces x by the automaton by a minimum of one, namely x_{cex} will be accepted from there on.

Theorem 1. *Assuming $\mathcal{T}$ is a PDFA with n states, and the membership queries $\mathcal{MQ}(x)$ be noise free. Then the algorithm will terminate after a finite number of iterations and output a hypothesis $\mathcal{H}$ with $n' \leq n$ states s.t. $\forall x \in \Sigma^* : |P(x) - \pi(x)| \leq \mu$.*

Proof. The proof follows from the previous lemmas. The idea is that in each turn, one of the two main routines will increase the number of identified states by at least one. Since the target has a finite number of states the algorithm must terminate.

5 Experiments and Results

To test our algorithm we applied it to three datasets, namely the CTU-13 dataset, the HDFS dataset, and the BGL dataset. We extracted sliding windows and trained a language model of the form $P(a|\sigma), a \in \Sigma_\xi$ for each of them. We then distilled PDFA from the networks.

To test how well the distilled models approximate the underlying neural networks we did the following: We set the threshold μ to the values of $1e^{-3}$, $1e^{-5}$, $1e^{-7}$, and $1e^{-10}$ respectively. We report for each tested value of μ the values $min_x(|P(x)-\pi(x)|)$, $max_x(|P(x)-\pi(x)|)$, $MSE = \sqrt{\sum_x (P(x) - \pi(x))^2}$, as well as the number of states. In all these instances, x is over the corresponding test-set of the respective dataset. Our test sets were all limited to a size of $20,000$, due to the slow prediction of the neural networks.

Additionally, we made sure that the neural networks learned the underlying datasets correctly in the following manner: All selected datasets consist of normal and anomalous data. We consider an extracted sliding window malign if it contains one or more malign tokens, else it is benign. After training the models, we can assign labels: Given a sequence of length $|x| = m$ and $x = a^0a^1...a^{m-1}$, we obtain predicted probabilities $P_1 = P(a^0|\lambda)$, $P_2(a^1|a^0)$, ..., $P_m = P(a^{m-1}|a^0...a^{m-2})$ from the network. We assign a score of $1 - min_{P_i, i \in \{0...m\}}$ to each sequence, over which we compute an receiver operating characteristic (ROC) curve and report the area-under-curve (AUC). The closer the AUC to a value of 1, the better the classification works. Because the

inferred PDFAs behave similar to the underlying neural network in their input-output-behavior, we can use the same method to assign anomaly scores as we did with the networks.

5.1 Datasets

CTU-13. The CTU-13 dataset [5] consists of captured network traffic grouped together in Netflow format [8] called flows, and each flow is labeled as either background or malicious. The dataset comes in 13 individual scenarios. We picked scenario 10 and used the algorithm of Pellegrino et al. [17] to encode the netflows into an alphabet of size $|\Sigma| = 92$ with 10 even percentiles. To create sequences we sorted via time stamp and grouped by connection, and then extracting sliding windows of size 10 over the connections. Connections with fewer flows resulted in a single window.

We considered a sequence as malign if it contained a malicious data packet, else benign. We counted both the number of benign n_{benign} and malign sequences n_{malign}, and computed $f = \frac{n_{malign}}{n_{benign}}$. Malign sequences were automatically assigned to the test-set. Benign sequences were randomly assigned to the test-set with a uniform probability of f, and else to the train-set. This way, the number of benign and malign sequences in the test-set was roughly even, and the train-set consisted only of benign sequences, making sure the network learned only the normal behavior of the network. Because the train-set was too large for our machine, we randomly sampled $400,000$ sequences.

HDFS. Another dataset that we tested our algorithm on is the HDFS dataset [21], which represents logs generated from the Hadoop File System run on Amazon's Elastic Compute cloud. Because the system logs come semi-structured, they first have to be tokenized. We used the already preprocessed dataset as made available by [3].

BGL. The last dataset we tested on was the BGL dataset [16]. The dataset consists of log-data collected on the Blue Gene/L supercomputer as of 2006, which has been labelled by alerts or normal activity. To obtain tokens we applied the DRAIN algorithm [6] implemented by Zhu et al. [22], giving us an alphabet of size $|\Sigma| = 321$. We then grouped the templates by node and rolled a sliding window of size 10 the obtained tokens, and then split the resulting sequences into train- and a test-set. We split the dataset into train- and test-set the same way we did on the CTU-13 dataset. Due to the large size of the test-set, we further randomly sampled $100,000$ sequences from the test-set for evaluation.

5.2 Results and Discussion

Approximation Results. The initial results of the experiments can be taken from Table 1. We omitted the entries for $\mu = 1e^{-5}$ on the HDFS dataset, because

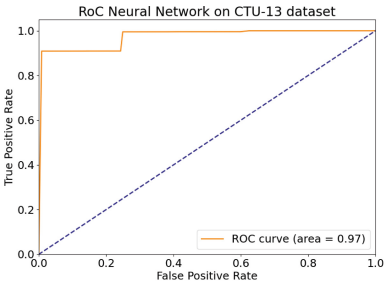
Table 1. Results of the experiments. We show the approximation errors on the respective test-sets, as well as the resulting number of states that the PDFA has.

Dataset	Metric	$\mu = 1e^{-3}$	$\mu = 1e^{-5}$	$\mu = 1e^{-7}$	$\mu = 1e^{-10}$
CTU-13	$min_x(\|P(x) - \pi(x)\|)$	$1.27e\{-78\}$	$1.27e\{-78\}$	$1.27e\{-78\}$	$1.27e\{-78\}$
	$max_x(\|P(x) - \pi(x)\|)$	$4.08e\{-4\}$	$9.69e\{-5\}$	$7.20e\{-5\}$	$1.40e\{-4\}$
	MSE	$1.71e\{-10\}$	$4.56e\{-12\}$	$2.43e\{-12\}$	$5.27e\{-12\}$
	# States	14	121	1493	3644
HDFS	$min_x(\|P(x) - \pi(x)\|)$	$5.16e\{-256\}$	–	$5.16e\{-256\}$	$5.16e\{-256\}$
	$max_x(\|P(x) - \pi(x)\|)$	$1.15e\{-6\}$	–	$1.28e\{-7\}$	$7.44e\{-8\}$
	MSE	$2.00e\{-13\}$	–	$1.55e\{-15\}$	$7.65e\{-16\}$
	# States	1	1	14	798
BGL	$min_x(\|P(x) - \pi(x)\|)$	$6.70e\{-177\}$	$6.59e\{-190\}$	$3.62e\{-193\}$	$3.91e\{-186\}$
	$max_x(\|P(x) - \pi(x)\|)$	$3.02e\{-3\}$	$3.02e\{-3\}$	$6.47e\{-4\}$	$3.02e\{-3\}$
	MSE	$8.90e\{-8\}$	$8.90e\{-8\}$	$7.09e\{-9\}$	$8.90\{e-08\}$
	# States	16	165	918	3500

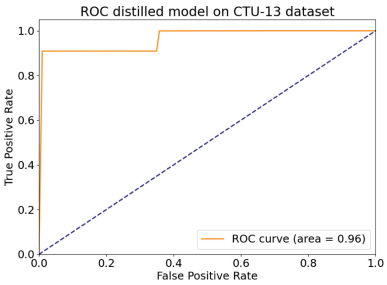
the distilled model is the same as for $\mu = 1e^{-3}$. Because of the increased complexity of the BGL dataset we had to use our early-stop criterion at 3500 states, as the machine became too large for our hardware with 16GB RAM memory.

Obviously, the HDFS dataset was the easiest to learn. We can see from the results that the model could be represented with a single state and self loops even for a value of $\mu = 1e^{-5}$, and only 14 states for $\mu = 1e^{-7}$. Compared with the HDFS dataset the CTU-13 and the BGL datasets learned larger models. This is likely due to their larger alphabet sizes, which increases the number of possible search paths exponentially.

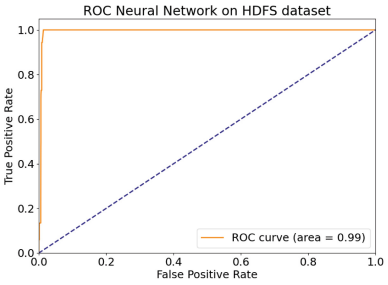
Decreasing the μ parameter greatly increased the number of states of the distilled models, and in the end many more states have to be added for less gain. Interestingly, on the CTU-13 and BGL datasets the performance in terms of maximum error and MSE decreased slightly in the model with the largest number of states compared with a smaller one. We again attribute this to the fact that larger models are harder to check. An example: In the example of the BGL dataset, we identified the sequence consisting of template 314 repeated ten times. The probabilities are predicted precise until after seven applications of τ the distilled model suddenly underestimates the real probability by several orders of magnitude, caused by a wrong merge. Due to the size of the model this wrong merge is much harder to detect by the counterexample search and remains in the final model. Due to the fact that this string appears a lot in the test-set the MSE of the model becomes larger again. The model with 918 states does not make this wrong merge and predicts the same sequence better. This is a problem common to all active learning algorithms, but can be improved with better search strategies. For a more elaborate discussion of this we refer the reader to the work of Marzouk and de la Higuera [10].



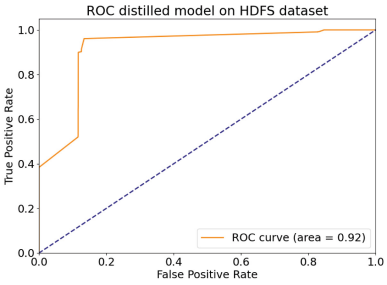
(a) CTU-13 dataset: Network.



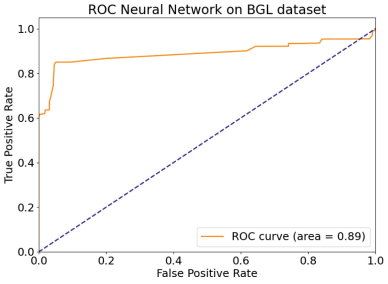
(b) CTU-13 dataset: Model with 14 states.



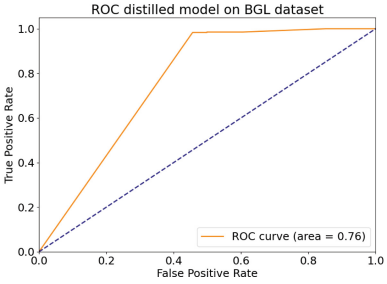
(c) HDFS dataset: Network.



(d) HDFS dataset: Model with 14 states.



(e) BGL dataset: Network.



(f) BGL dataset: Model with 16 states.

Fig. 2. ROC curves obtained on the datasets. On the left are the ROC curves of the neural network, contrasted with a respective distilled PDFA to the right.

Anomaly Detection Results. We show some results for the anomaly detection, a proxy for how well the networks approximated the dataset, as well as an example for a distilled in Fig. 2. We can see that the networks performed very strong on the CTU-13 dataset and the HDFS dataset, with an AUC of 0.97 and 0.99 respectively. The BGL dataset proved to be more challenging with an AUC of 0.89 for the network. This has to do with the underlying dataset, which is

harder to learn than the other two, as well as with how we chose to extract tokens from it. However, the network is still capable of detecting anomalies significantly above random guesses, showing that it learned the data it was given.

Not very surprising the distilled models did not match the underlying network's performances perfectly: The model with 14 states on the CTU-13 dataset obtained an AUC of 0.96, just very slightly below the neural network. On the HDFS dataset however the model with 14 states has an AUC of just 0.92, 0.07 below the neural network, and the distilled model with 15 states on the BGL dataset has an AUC of just 0.76, 0.13 below the neural network. Interestingly, on most of the datasets the smaller models were the better ones, and in most cases the detection performance decreased with larger models. We attribute this to the following: Larger models do have more paths to cover. Therefore, ensuring consistency with the error bound becomes much harder with smaller μ. An exception to this was posed by the HDFS dataset, where the model with 1 state had an AUC of 0.4 only, indicating that an error of $1e^{-5}$ is not sufficient for any anomaly detection on this problem.

Use Case Example. In this subsection we want to demonstrate how the models can be used. We take the distilled model with 14 states we extracted from the on the HDFS data trained network. Figure 3 shows the model. For simplicity we omitted all transitions with a probability less than $\pi(q, a) \leq 1e^{-4}$, marked all transitions with probabilities larger than 0.01 blue, and all with probabilities larger than 0.1 green. The following is a malign string from our test-set: *17, 26, 26, 28, 16*. In our model we can see that the first three transitions were highly

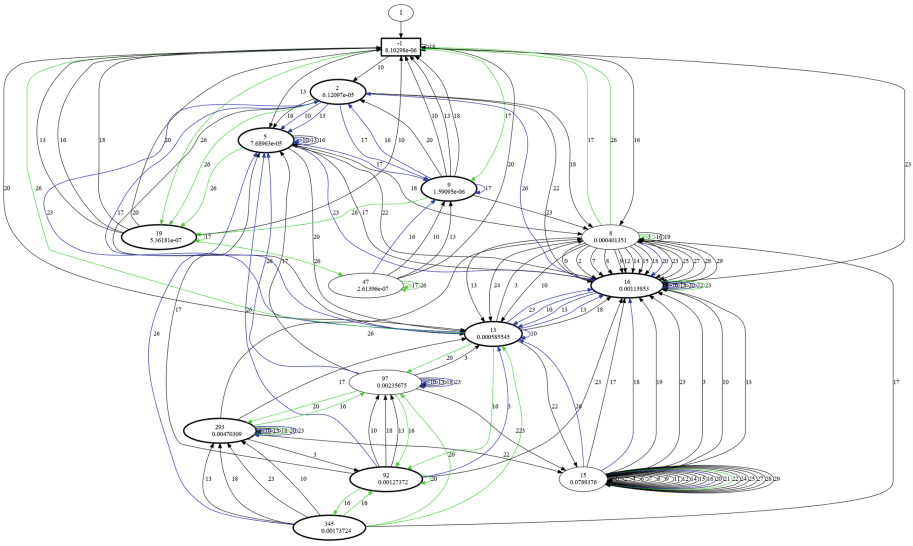


Fig. 3. Visual representation of the 14-state PDFA extracted from the NN trained on the HDFS dataset.

probable. However, the transition after the second *26* is very low according to our model. In fact the substring *17, 26, 26, 28* does not appear in our training set at all. We can thus use our distilled model to get a clean visual representation of our network, helping us to predict low probability strings.

6 Conclusion and Future Work

In this work we introduced a new algorithm that actively learns PDFAs and demonstrated its capabilities learning simple models that can still achieve good detection results in anomaly detection examples. We also showed a simple use case example of how the distilled PDFA can be used as a global explanation for an underlying neural network. An advantage of our method is that bounds are guaranteed to hold on every tested string. Other methods use merge criteria such as the variation distance, which is a much less intuitive metric when parsing whole strings. Compared with spectral methods and in accordance with some other works we are also able to infer deterministic models.

Just as in the case of the original $L^{\#}$ algorithm, the run-time of our algorithm is a drawback. Even smaller machines can lead to a large observation tree, leading to many asked queries. We could pinpoint the bulk of the run-time to these queries. Tackling this could for example include pruning the tree when predicted probabilities become too small to violate the error bounds. Another possible optimization of the algorithm is obviously a better counterexample search. We also consider the further investigation of applications such as the anomaly detection an interesting research direction.

Acknowledgments. This work is supported by NWO TTW VIDI project 17541 - Learning state machines from infrequent software traces (LIMIT).

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
2. Balle, B., Carreras, X., Luque, F.M., Quattoni, A.: Spectral learning of weighted automata: a forward-backward perspective. Mach. Learn. 33–63 (2014)
3. Du, M., Li, F., Zheng, G., Srikumar, V.: Deeplog: anomaly detection and diagnosis from system logs through deep learning. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, pp. 1285–1298. Association for Computing Machinery, New York (2017)
4. Eyraud, R., Ayache, S.: Distillation of weighted automata from recurrent neural networks using a spectral approach. Mach. Learn. (2021)
5. García, S., Grill, M., Stiborek, J., Zunino, A.: An empirical comparison of botnet detection methods. Comput. Secur. **45**, 100–123 (2014)
6. He, P., Zhu, J., Zheng, Z., Lyu, M.R.: Drain: an online log parsing approach with fixed depth tree. In: 2017 IEEE International Conference on Web Services (ICWS), pp. 33–40 (2017)
7. de la Higuera, C.: Grammatical Inference: Learning Automata and Grammars. Cambridge University Press, Cambridge (2010)

8. Hofstede, R., et al.: Flow monitoring explained: from packet capture to data analysis with NetFlow and IPFIX. IEEE Commun. Surv. Tutor. **16**(4), 2037–2064 (2014)
9. Lang, K.J., Pearlmutter, B.A., Price, R.A.: Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In: Honavar, V., Slutzki, G. (eds.) ICGI 1998. LNCS, vol. 1433, pp. 1–12. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054059
10. Marzouk, R., de la Higuera, C.: Distance and equivalence between finite state machines and recurrent neural networks: computational results. CoRR (2020). https://arxiv.org/abs/2004.00478
11. Mayr, F., Yovine, S.: Regular inference on artificial neural networks. In: Holzinger, A., Kieseberg, P., Tjoa, A.M., Weippl, E. (eds.) CD-MAKE 2018. LNCS, vol. 11015, pp. 350–369. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99740-7_25
12. Mayr, F., Yovine, S., Carrasco, M., Pan, F., Vilensky, F.: A congruence-based approach to active automata learning from neural language models. In: Coste, F., Ouardi, F., Rabusseau, G. (eds.) Proceedings of 16th edition of the International Conference on Grammatical Inference. Proceedings of Machine Learning Research, vol. 217, pp. 250–264. PMLR (2023)
13. Mayr, F., Yovine, S., Pan, F., Basset, N., Dang, T.: Towards efficient active learning of PDFA (2022)
14. Muškardin, E., Aichernig, B.K., Pill, I., Tappler, M.: Learning finite state models from recurrent neural networks. In: ter Beek, M.H., Monahan, R. (eds.) Integrated Formal Methods, pp. 229–248. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-07727-2_13
15. Okudono, T., Waga, M., Sekiyama, T., Hasuo, I.: Weighted automata extraction from recurrent neural networks via regression on state spaces (2019)
16. Oliner, A., Stearley, J.: What supercomputers say: a study of five system logs. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007), pp. 575–584 (2007)
17. Pellegrino, G., Lin, Q., Hammerschmidt, C., Verwer, S.: Learning behavioral fingerprints from netflows using timed automata. In: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), pp. 308–316 (2017)
18. Vaandrager, F., Garhewal, B., Rot, J., Wißmann, T.: A new approach for active automata learning based on apartness. In: Fisman, D., Rosu, G. (eds.) TACAS 2022. LNCS, vol. 13243, pp. 223–243. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-99524-9_12
19. Weiss, G., Goldberg, Y., Yahav, E.: Learning deterministic weighted automata with queries and counterexamples. In: Wallach, H., Larochelle, H., Beygelzimer, A., d' Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems, vol. 32. Curran Associates, Inc. (2019)
20. Weiss, G., Goldberg, Y., Yahav, E.: Extracting automata from recurrent neural networks using queries and counterexamples. Mach. Learn. (2022)
21. Xu, W., Huang, L., Fox, A., Patterson, D., Jordan, M.I.: Detecting large-scale system problems by mining console logs. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP 2009, pp. 117–132. Association for Computing Machinery, New York (2009)
22. Zhu, J., et al.: Tools and benchmarks for automated log parsing. In: Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2019, pp. 121–130. IEEE Press (2019)