The Next Symbol Prediction Problem: PAC-learning and its relation to Language Models

Satwik Bhattamishra[1] **Phil Blunsom**[1,2] **Varun Kanade**[1]

[1]University of Oxford [2]Cohere

{satwik.bmishra, phil.blunsom, varun.kanade}@cs.ox.ac.uk

Abstract

The *next symbol prediction* (NSP) problem has been widely used to empirically evaluate the performance of neural sequence models on formal language tasks. We formalize the setting so as to make it amenable to PAC-learning analysis. In the NSP setting, a learning algorithm receives valid sequences (positive examples) from the underlying language, along with rich labels indicating for every prefix, whether the prefix is in the language and what symbols could appear subsequently that lead to accepting string. In the conventional classification setting where learning occurs with only positive and negative examples, the problem of learning regular languages or even subclasses represented by acyclic DFAs is known to be computationally hard based on cryptographic assumptions. In contrast, our main result shows that regular languages are efficiently PAC-learnable in the next symbol prediction setting. Further, we provide a more efficient learning algorithm for the case where the target DFA is known to be acyclic. Given the rich labels required in the NSP setting, one may wonder whether this setting is applicable to non-artificial tasks. We explain how language models can act as a source of such labeled data, and consequently, our algorithm can be applied to fit a finite-state model (DFA) that learns the (truncated) support of the language model.

1 Introduction

The ability of large language models (LLMs) to generate well-formed text has improved significantly over the past decade. However, they remain largely uninterpretable, and the sentences they could generate in practice remain to some degree unpredictable. An interesting and arguably fundamental question about language models is the following: *Given a neural network-based language model, can we construct an interpretable formal system such as a finite automaton that (approximately) accepts the same strings that are in the support of the model and rejects them otherwise?* In this work, we formally introduce and examine this problem.

Problem. We study the problem of learnability of languages in the *Next Symbol Prediction* (NSP) setting. In this setting, the learning algorithm is granted access to an example oracle, capable of supplying positively labeled strings. For every prefix of a given string, the oracle presents the set of permissible continuations and the label associated with that prefix. During evaluation, a prediction on a test example is considered correct if the output hypothesis can predict the set of valid continuations as well as the labels for *every* prefix correctly.

Why study this problem? There are two primary motivations behind understanding the complexity of learning in the NSP setting.

(a) Learning support of LMs. First, developing efficient learning algorithms for the NSP setting can be applied to learning the support of the language models. The support of the model contains

Mathematics of Modern Machine Learning Workshop at NeurIPS 2023.

strings that are assigned nonzero probability by a model. Although softmax-based LMs assign a nonzero probability to all strings, texts are sampled from language models by employing methods such as top-k [Fan et al., 2018, Radford et al., 2019] or top-p sampling [Holtzman et al., 2019] where not all strings have a nonzero probability of being generated. To learn the set of strings that can be generated by language models, efficient algorithms for the NSP setting can be applied which could be useful for understanding the robustness and verification of the language model. Past works [Hewitt et al., 2020, Yellin and Weiss, 2021, Yao et al., 2021] have also evaluated language models by testing them in the same way as is done in the NSP setting. Such evaluations can be made more robust by applying learning algorithms to verify the correctness of the models instead of sampling examples to evaluate the prediction of the models.

(b) Empirical analysis of models. There has been significant interest in characterizing the class of formal languages that sequence models such as Transformers and LSTMs can learn in practice. Past works on understanding neural sequence models by conducting experiments in the NSP framework date back to a couple of decades ago [Gers and Schmidhuber, 2001, Rodriguez, 2001] and are more recently used in numerous works (see Suzgun et al. [2019b,a], Ebrahimi et al. [2020], Bhattamishra et al. [2020] and references within). On the other hand, several other works evaluate the capabilities of models in the standard classification setting [Delétang et al., 2022, Liu et al., 2022, Weiss et al., 2018]. Prior works seem to have adopted the NSP setting because the evaluation of models seems to be much stricter than classification in the sense that models have to make correct predictions for every prefix and hence, the accuracy of Null classifier or random guessing is near zero. While several works have adopted the NSP setting for experiments, the learnability of languages in the setting and its relation to learnability in the classification setting is not clear. Before drawing conclusions about the kind of formal languages neural networks can learn by conducting such experiments, it is essential to formally understand the learnability of languages in this setting.

Our contributions. We show that regular languages, i.e. languages that can be represented using DFAs, are efficiently PAC-learnable in the NSP setting. This is particularly interesting because the problem of finding the minimum consistent DFA or *properly* learning DFAs from just positive and negative examples is known to be NP-hard [Gold, 1978, Angluin, 1978, Pitt and Warmuth, 1993]. Moreover, the problem of *improperly* learning (output hypothesis need not be a DFA) DFAs or even the subclass of DFAs without cycles is known to be computationally intractable under cryptographic assumptions [Kearns and Valiant, 1994]. In this work, we give a $\tilde{\mathcal{O}}(n^3L^3)$-time algorithm of learning DFAs, where n is the number of states in the DFA, and L is the length of the longest string; furthermore, we give a more efficient $\tilde{\mathcal{O}}(n^2L)$ algorithm in the case where the target DFA has no cycles. Our algorithms are based on state-merging algorithms [Oncina and Garcia, 1992] and leverage the fact that certain labels in the NSP setting can help us identify distinguishing strings to efficiently find the minimum consistent DFA. Our results suggest that even if the NSP setting has a stricter evaluation protocol, due to the rich set of labels provided to the learning algorithm, the problem of learning in the NSP setting is easier than the standard PAC-learning setting.

2 Problem

Notation. A deterministic finite automaton (DFA) is characterized by a finite set of states Q, a finite alphabet Σ (or the vocabulary), a transition function $\delta : Q \times \Sigma \to Q$, a start state $q_0 \in Q$, and a set of accepting states $F_A \subseteq Q$. Let $x_{:n}$ denote the prefix of x up to length n. We denote the empty string with λ. We use $\mathrm{Pref}(x) = \{x_{:k} \mid 0 \leq k \leq |x|\}$ to denote a function that takes a string as input and returns all its prefixes including the empty string λ as output. Given a set of strings S, the function $\mathrm{Pref}(S)$ returns the set of all prefixes of all strings in S. For any state q, we use $F(q)$ to denote a function that outputs 1 if $q \in F_A$ and 0 otherwise. We use q_{dead} to denote a dead state which is a reject state where all symbols have a transition to the same dead state. Let $\varphi(q, \sigma)$ denote a function which outputs 1 if $\delta(q, \sigma) = q_{\mathrm{dead}}$ and 0 otherwise.

Next Symbol Prediction task. In the *next symbol prediction* (NSP) setting, the example oracle produces only positive examples but for every prefix in an input string, it also provides the set of valid/legal symbols for the next step as well as the label $c(x_{:n})$ of the prefix. More concretely, for any string $x = s_0, s_1, s_2, \ldots, s_n$, the example oracle provides $|\Sigma| + 1$ labels $\in \{0, 1\}$ for each prefix $x_{:n} = s_0, \ldots, s_k$ where $0 \leq k \leq n$ and s_0 is the empty string λ. For any prefix, there is a label corresponding to each symbol in the alphabet Σ. For any prefix $x_{:n}$, a label 1 corresponding to a symbol $\sigma \in \Sigma$ indicates that there exists a suffix s such that the string $x_{:n} \cdot \sigma \cdot s$ has a positive label

or it belongs to the language induced by the concept c. Similarly, the label 0 for a symbol σ indicates that there are no suffixes (including the empty string) such that $x_{:n} \cdot \sigma \cdot s$ has a positive label. Hence, for an input string of length l, the labels can be seen as a vector of the form $\{0,1\}^{(|\Sigma|+1)(l+1)}$. Note that, although the example oracle provides only positive examples, information about the negative examples can be obtained from the labels of the prefixes and the labels indicating the valid set of symbols for the next step.

For any input x, a predictor h has to predict the next set of valid symbols and the label for each prefix – $(|\Sigma|+1)(l+1)$ labels. The error of h for any input is 1 if it predicts any of the $(|\Sigma|+1)(l+1)$ incorrectly and the error is 0 otherwise. Let $h(x)$ denote the $(|\Sigma|+1)(l+1)$ length output vector, then the error for a given input is defined as $\mathrm{err}(h(x), c(x)) = \left\|h(x) - c(x)\right\|_\infty$ and the error of h with respect to target c and the distribution D is $\mathcal{L}_{\mathrm{NSP}}(h; c, D) = \mathbb{E}_D[\mathrm{err}(h(x), c(x))]$.

Boolean Acyclic DFAs. The concept class of Boolean Acyclic DFAs (B-DFA) contains DFAs which represent functions of the form $f : \{0,1\}^* \to \{0,1\}$ where for all $x \notin \{0,1\}^L$, the function $f(x) = 0$. Hence, the class B-DFA can be seen as a class of functions comprising Boolean functions which are of the form $\{0,1\}^L \to \{0,1\}$.

We use DFA_n and B-DFA_n to denote the class of DFAs and Boolean Acyclic DFAs with at most n states respectively.

3 Results

Definition 3.1. – PAC-learning in NSP setting. A concept class C is PAC-learnable in the next symbol prediction setting using hypothesis class H if there exists a learning algorithm L which satisfies the following: for every $c \in C$, for every distribution D over X, for every $0 < \epsilon < 1/2$, for every $0 < \delta < 1/2$, if L is given access to $\mathrm{EX}(c, D)$, and inputs ϵ and δ, then L outputs h such that with probability $1 - \delta$, the error $\mathcal{L}_{\mathrm{NSP}}(h) \leq \epsilon$. Further, the runtime of L and the number of calls to the example oracle $\mathrm{EX}(c, D)$ must be bounded by a polynomial in size(c), $\frac{1}{\epsilon}$, and $\frac{1}{\delta}$.

A well-known approach to designing learning algorithms is the ERM (Empirical Risk Minimization) framework, where finding a hypothesis consistent with the labeled data from a constrained class of functions suffices to guarantee generalization on unseen examples. In particular, if the class of hypotheses, H is finite, a straightforward application of Occam's principle guarantees that if the h has a low error on $\tilde{O}(\log |H|/\epsilon)$ data, then the hypothesis has at most ϵ error with high probability. Hence, the main difficulty in designing learning algorithms is to develop methods that find automata with a relatively small number of states (a measure of complexity) consistent with the data. In fact, our algorithms find the *minimum consistent* DFA in polynomial time in the NSP setting – something known to be NP-hard when only using positive and negative examples.

Prefix tree DFA. For both of our results, the algorithm begins by creating a prefix tree DFA which is a trie-like structure where each prefix of the sample set ($\mathrm{Pref}(S)$) corresponds to a state in the DFA. For each prefix $s \in \mathrm{Pref}(S)$ and $\sigma \in \Sigma$ such that $s \cdot \sigma \in \mathrm{Pref}(S)$, a transition between the states corresponding to s and $s \cdot \sigma$ exist in the DFA. Based on the labels provided in the NSP setting, the states are marked as accept or reject and transitions to dead state are added. Further details about the prefix tree DFA are provided in Appendix B.

Theorem 3.1. *The concept class of DFAs is efficiently PAC-learnable in the Next Symbol Prediction setting.*

The proof is provided in Appendix C. Given a set of examples and their corresponding labels, the algorithm begins by constructing a prefix tree DFA (Algorithm 1) where each state corresponds to a prefix in the sample set ($\mathrm{Pref}(S)$). The states are partitioned into three sets – permanent states, candidate states, and unlabelled states. Initially, the set of permanent states only has the initial state corresponding to the empty string (λ) and the states corresponding to $\sigma \in \Sigma$ which are neighboring the initial state are in the set of candidate states.

The algorithm iteratively tries to merge the candidate states with each of the permanent states. The merge between a candidate state and a permanent state succeeds if and only if certain conditions regarding the consistency of the resulting DFA can be guaranteed. If a candidate state cannot be merged with any permanent state then it is promoted to a permanent state. In both cases, some unlabelled states may also be promoted to candidate states. The state-merging process is designed in

such a way that, for every pair of permanent states, a distinguishing string in the minimum consistent DFA is guaranteed to exist which ensures that the resulting DFA at the end is minimum. The algorithm terminates when no candidate or unlabelled states remain and returns the minimum DFA consistent with the sample set.

Given m examples, the time complexity of the algorithm is $\mathcal{O}(m^3L^3)$. Due to Lemma B.1, the minimum consistent learner PAC-learns the class DFA_n in $\mathcal{O}((n\log n)^3L^3)$.

While regular languages are computationally hard to learn with positive and negative examples, they are efficiently PAC-learnable with access to a membership query oracle [Angluin, 1987]. Our result implies that in the case of the NSP setting, efficient (PAC) learning of regular languages is possible even without membership queries. Whether membership queries can be used to derive more efficient learning algorithms is an open problem.

Theorem 3.2. *The concept class of Boolean Acyclic DFAs is efficiently PAC-learnable in the Next Symbol Prediction setting.*

The proof is provided in Appendix D. Similar to the previous case, the algorithm begins by constructing a prefix tree DFA using the samples and labels provided by the oracle. The algorithm then iteratively tries to merge states corresponding to prefixes of different lengths. The merge between any two states succeeds if the corresponding prefixes are indistinguishable based on the sample set. For any two states that are not merged till the end, they are guaranteed to have a distinguishing string based on the labels provided in the sample set. Unlike the case of the general class of DFAs, we can leverage certain properties known about the structure of the DFA to minimize the DFA more efficiently. Since the algorithm iterates over prefixes of different lengths and the merge is attempted among prefixes at a particular length, the time complexity of the algorithm is $\mathcal{O}(m^2L)$. Given that the log of the size of the hypothesis set is $\mathcal{O}(n\log n)$ (Lemma B.1), this implies that the state-merging algorithm PAC-learns the class of Boolean Acyclic DFAs in $\mathcal{O}((n\log n)^2L)$ time. Note that, we focus on the case where $\Sigma = \{0, 1\}$ given the hardness results associated with it in the standard classification setting but the algorithm can be extended for any finite sized Σ with minor changes.

Reduction to learning support of LMs. Given a PAC-learning algorithm for the NSP setting, a language model (LM) can act as an example oracle and target function to generate labeled examples. An LM is a function of the form $\Sigma^* \to \Delta^{|\Sigma|-1}$ where the simplex $\Delta^{|\Sigma|-1}$ is set of all possible probability distributions over Σ. Suppose, the generation of sequences is done using top-k or top-p sampling. The set of strings that can be generated by such top-k or top-p sampling has been referred to as the ϵ-truncated support [Hewitt et al., 2020, Yellin and Weiss, 2021]. For any input string, the symbols in the top k ranking (or in top-p probability mass) are valid continuations and are assigned label 1, and other symbols are assigned label 0. The symbols which correspond to the end of sequence (such as space for words or '.' for sentences) can be considered as the equivalent of the label of the entire sequence. For formal (or programming) languages, the end of sequence symbol will be more well-defined. Suppose a PAC-learning algorithm A exists for learning a concept class C in the NSP setting (such as Algorithm 2) and assume that the support of the language is in class C as well. Then, the LM can act as the example oracle where the examples can be labeled as described above. Note that, while our algorithm to learn DFAs in the NSP is primarily discussed with only positive examples, with minor changes, it works with both positive and negative examples (along with other labels) as well. Given sufficient examples from some distribution over Σ^* labeled by an LM, the algorithm will output a hypothesis h such that with high probability, h can accurately predict whether the string belongs to the truncated support of the LM with error at most ϵ.

4 Conclusion

We formally introduced the problem of learnability in the next symbol prediction setting and discussed its relation to learning the support of LMs and empirical analysis of sequence models. We showed that certain fundamental concept classes that are not PAC-learnable in the conventional classification setting are efficiently learnable in the NSP setting. Note that, while the time and sample complexity of the algorithm for learning DFAs is polynomial, it could still get quite large when applying them to practical problems related to language models. Understanding the effectiveness and drawbacks of the described algorithm by applying it to language models for text or even formal languages is a natural direction for future work. An interesting open problem is whether the use of membership queries can help in any way to improve efficiency in the next symbol prediction setting.

References

Dana Angluin. On the complexity of minimum inference of regular sets. *Information and control*, 39 (3):337–350, 1978.

Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the Ability and Limitations of Transformers to Recognize Formal Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7096–7116, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.576. URL https://aclanthology.org/2020.emnlp-main.576.

Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Marcus Hutter, Shane Legg, and Pedro A Ortega. Neural networks and the chomsky hierarchy. *arXiv preprint arXiv:2207.02098*, 2022.

Javid Ebrahimi, Dhruv Gelda, and Wei Zhang. How can self-attention networks recognize dyck-n languages? *arXiv preprint arXiv:2010.04303*, 2020.

Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. *arXiv preprint arXiv:1805.04833*, 2018.

Felix A Gers and E Schmidhuber. Lstm recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

E Mark Gold. Complexity of automaton identification from given data. *Information and control*, 37 (3):302–320, 1978.

John Hewitt, Michael Hahn, Surya Ganguli, Percy Liang, and Christopher D Manning. Rnns can generate bounded hierarchical languages with optimal memory. *arXiv preprint arXiv:2010.07515*, 2020.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

Michael Kearns and Leslie Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *Journal of the ACM (JACM)*, 41(1):67–95, 1994.

Bingbin Liu, Jordan T Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata. *arXiv preprint arXiv:2210.10749*, 2022.

José Oncina and Pedro Garcia. Identifying regular languages in polynomial time. In *Advances in structural and syntactic pattern recognition*, pages 99–108. World Scientific, 1992.

Leonard Pitt and Manfred K Warmuth. The minimum consistent dfa problem cannot be approximated within any polynomial. *Journal of the ACM (JACM)*, 40(1):95–142, 1993.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Paul Rodriguez. Simple recurrent networks learn context-free and context-sensitive languages by counting. *Neural computation*, 13(9):2093–2118, 2001.

Mirac Suzgun, Yonatan Belinkov, Stuart Shieber, and Sebastian Gehrmann. LSTM networks can perform dynamic counting. In *Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges*, pages 44–54, Florence, August 2019a. Association for Computational Linguistics. doi: 10.18653/v1/W19-3905. URL https://www.aclweb.org/anthology/W19-3905.

Mirac Suzgun, Yonatan Belinkov, and Stuart M. Shieber. On evaluating the generalization of LSTM models in formal languages. In *Proceedings of the Society for Computation in Linguistics (SCiL) 2019*, pages 277–286, 2019b. doi: 10.7275/s02b-4d91. URL https://www.aclweb.org/anthology/W19-0128.

Gail Weiss, Yoav Goldberg, and Eran Yahav. On the practical computational power of finite precision RNNs for language recognition. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 740–745, Melbourne, Australia, July 2018. Association for Computational Linguistics. doi: 10.18653/v1/P18-2117. URL https://www.aclweb.org/anthology/P18-2117.

Shunyu Yao, Binghui Peng, Christos Papadimitriou, and Karthik Narasimhan. Self-attention networks can process bounded hierarchical languages. *arXiv preprint arXiv:2105.11115*, 2021.

Daniel M Yellin and Gail Weiss. Synthesizing context-free grammars from recurrent neural networks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 351–369. Springer, 2021.

A Definitions

Definition A.1. A Deterministic Finite Automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states,
- Σ is a finite alphabet,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_\lambda \in Q$ is the start state, and
- $F \subseteq Q$ is the set of accepting states.

For any string $x \in \Sigma^*$, the DFA starts with the start state q_λ and transitions to other states based on the function δ and the symbols in the string. If the transitions end on a final or accept state, then the string is accepted or else it is rejected. For any particular DFA, the set of strings in Σ^* that lead to the accept state is also referred to as the language of the DFA.

Minimum consistent learner is an algorithm which, given a hypothesis class $\mathcal{H}$ and a set of training examples S, selects the hypothesis $h \in \mathcal{H}$ that is consistent with the examples in S and minimizes the complexity of the hypothesis. Formally, for a target function c, a hypothesis h is consistent with S if and only if $\forall x \in S, h(x) = c(x)$, meaning it accurately predicts the label of every example in the training set S. In the context of the NSP setting where $h(x) \in \{0,1\}^{(|\Sigma|+1)(l+1)}$, a hypothesis h is consistent with samples in S if and only if $\left\|h(x) - c(x)\right\|_\infty = 0$ for all examples in S. In our work, we focus on proper PAC learning where the target concept class and hypothesis class are the same. In other words, the learner for the class DFA_n or $\mathrm{B\text{-}DFA}_n$ finds the minimum consistent hypothesis within the same class of functions respectively.

Note that, even in the NSP setting, finding the minimum consistent hypothesis implies the PAC-learnability of a concept class. Given a set S of examples and labels, let L be a minimum consistent learner which produces a hypothesis h in polynomial time such that the error on the sample set $\hat{\mathcal{L}}_{\mathrm{NSP}}(h, S) = 0$. Let h be a 'bad' hypothesis – $\mathcal{L}_{\mathrm{NSP}}(h) > \epsilon$. Let A_h be the event where a minimum consistent learner converges to h where $\mathcal{L}_{\mathrm{NSP}}(h) > \epsilon$ and $\hat{\mathcal{L}}_{\mathrm{NSP}}(h, S) = 0$. See that, $\mathbb{P}[A_h] \leq (1-\epsilon)^m \leq e^{-\epsilon m}$. With a simple application of union bound, the failure probability can be bounded by,

$$\mathbb{P}[\bigcup_{h \text{ bad}} A_h] \leq \sum_{h \text{ bad}} \mathbb{P}[A_h] \leq |H| \cdot e^{-\epsilon m} \leq \delta.$$

This implies that when $m \geq \frac{1}{\epsilon}\left(\log|H| + \frac{1}{\delta}\right)$, then with probability $1-\delta$, the error $\mathcal{L}_{\mathrm{NSP}}(h) \leq \epsilon$.

B Preliminaries

Prefix tree DFA is a DFA with a trie-like structure that is consistent with a given set of samples S. The algorithm to construct the prefix tree DFA is given in Algorithm 1. In our setting, to construct a prefix tree DFA, we first create a state for every prefix $\mathrm{Pref}(S)$ in the sample set S. We then create a transition $\delta(x_{:k}, \sigma) = x_{:k+1}$ between any two states corresponding to prefixes $x_{:k}$ and $x_{:k+1}$ if $x_{:k+1} = x_{:k} \cdot \sigma$. For every prefix $x_{:k}$, we have a set of $|\Sigma| + 1$ labels that indicate which symbols are allowed or not and whether the prefix is accepted by the target DFA or not. We mark the states as accept or reject based on the latter label. Based on the first $|\Sigma|$ labels indicating valid continuations we create a transition to dead state for $\sigma \in \Sigma$ that are labelled as not allowed. The dead state is a reject state where all symbols have a transition to the same dead state. This DFA is returned by the $\mathrm{Prefix\text{-}Tree}(S)$ algorithm when it is provided with the set of samples and labels S. At this point for every state we have three types of transitions for the symbols in the alphabet Σ, (a) valid transitions to another non-dead state, (b) transitions to dead states, and (c) undefined transitions for symbols which are valid continuations based on labels. Note that, if we add self-loops in the DFA for states with symbols that are allowed but undefined then we will have a consistent (but not necessarily minimum) DFA.

Ordering. The states are numbered according to the length-lexicographic order which is later used by the Select function to return the respective state. In the prefix tree, every state corresponds to a unique prefix, and the ordering between two prefixes (and the respective states) is determined by the length if two prefixes have different lengths or it is determined lexicographically if two prefixes have the same length. Formally, $x <_{length-lex} y$ iff $|x| < |y| \vee (x <_{lex} y \wedge |x| = |y|)$.

Algorithm 1 Prefix-Tree

Input: a sample set S
Output: Prefix tree $\mathcal{A} = \langle \Sigma, Q, q_\lambda, \mathbb{F}_\mathbb{A}, \delta \rangle$
$\mathbb{F}_\mathbb{A} \leftarrow \emptyset$
$Q \leftarrow \{q_u : u \in \mathrm{Pref}(S)\}$
for each $q_{u \cdot a} \in Q$ **do**
  $\delta(q_u, a) \leftarrow q_{ua}$
end for
for each $q_u \in Q$ **do**
  if u has a positive label **then**
    $\mathbb{F}_\mathbb{A} \leftarrow \mathbb{F}_\mathbb{A} \cup \{q_u\}$
  end if
  for each $\sigma \in \Sigma$ **do**
    if $u \cdot \sigma$ is not allowed **then**
      $\delta(q_u, \sigma) = q_{\mathrm{dead}}$
    end if
  end for
end for
return $\mathcal{A}$

Lemma B.1. *For DFAs with at most n states, $\log |\mathrm{DFA}_n| = \mathcal{O}(n \log n)$.*

This is a well-known result [De la Higuera, 2010] and can be proved in the following way. The proof follows from the fact that for a DFA with at most n states, there are $n^{|\Sigma| \cdot n}$ and each state can either be an accept or reject state (a factor of 2^n and one of the states can be an initial state. Hence, the total number of possible DFAs is $2^n \cdot n^{|\Sigma| \cdot n + 1}$. Thus, $\log |\mathrm{DFA}_n| = \mathcal{O}(n \log n)$.

C Learning DFAs in NSP setting

Theorem C.1. *In the next symbol prediction setting, given m examples labeled by any $\mathcal{A} \in \mathrm{DFA}_n$, the minimum consistent DFA can be computed in $\mathcal{O}(m^2 L)$-time where L is the length of the longest string in the m examples.*

Proof. We show that Algorithm 2 finds the minimum consistent DFA for a given set of examples with the labels corresponding to the NSP setting. The algorithm works as follows:

Overview. The state merging algorithm begins by creating a prefix tree DFA using algorithm 1 and then iteratively merging the states unless no two states can be merged any further. During the run, the states are merged in such a way that the resulting DFA remains consistent with the training set and when two states cannot be merged, then they are provably different in the minimum consistent DFA as well.

The algorithm maintains two sets of states namely *permanent* states (PERM) and *candidate* states (CAND). Initially, the set PERM only contains the initial state q_λ and the set CAND contains all the neighbours of the initial state. The algorithm iteratively tries to merge the candidate states with the permanent states until the set of candidate states is empty and then the algorithm returns the minimum consistent DFA and terminates.

State-merging. The algorithm goes through the set of candidate states and picks a state q_c based on the length-lexicographic order of the prefixes corresponding to the states (Step 7). Then, the algorithm tries to merge the candidate state q_c with each of the permanent states until it finds a permanent state q_p with which the merge succeeds. If a merge between q_c and a permanent state succeeds, then all the outgoing neighbors of permanent states which are not in CAND are added to

the candidate set (Step 13). The outgoing neighbours of any state q_p is defined as the set of states $\{q \mid \delta(q_p, \sigma) = q \text{ is defined} \wedge (q \neq q_{\text{dead}})\}$.

If the merge does not succeed with any of the permanent states then the candidate state q_c is promoted to a permanent state. The $\text{Promote}(q)$ function adds the state q to the set of permanent states PERM and adds all the outgoing neighbors of q_c to the set CAND.

It is important to note that the state merging algorithm is designed in such a way that for every candidate state, there is always only one incoming edge and if that edge is removed then the candidate state is the root of a tree (ignoring the transitions to dead state).

Merge and Fold Algorithm. After a candidate state q_c and the permanent state q_p have been chosen, the merge algorithm works as follows: first, the transition from the parent of the state q_c is reassigned to q_p (Step 2-3 in Alg. 4). This creates a separate tree with q_c as the root which we refer to as F-Tree. The Fold algorithm (Alg. 3) then recursively tries to fold the tree with q_c into the main DFA. The recursive folding process works as follows: the algorithm first tries to fold the candidate state q_c into q_p. Here, by folding we mean that q_c will be replaced by q_p in the original DFA. The fold only proceeds further if both the states are either accept or reject states (Steps 2-4). Additionally, the set of symbols that lead to the dead state q_{dead} from both states should be exactly the same (Steps 5-9). In other words, a fold between two states is only possible if both of them either accept states or reject states and the set of symbols for valid continuation from both states are the same.

If both these criteria are satisfied then the algorithm recursively checks these criteria for the states which are in those paths that are defined from both q_p and q_c. In particular, for a symbol $\sigma \in \Sigma$ if both $\delta(q_p, \sigma) = q_a$ and $\delta(q_c, \sigma) = q_b$ are defined then the algorithm will try to fold q_b into q_a (Step 13). While folding any two states such as q_b into q_a, if for any symbol σ the transition $\delta(q_b, \sigma) = q_k$ is defined but $\delta(q_a, \sigma)$ is not defined then the algorithm adds a transition $\delta(q_a, \sigma) = q_k$. If any of the criteria fails while trying to fold any two states then the merge between q_p and q_c fails and the algorithm tries to merge q_c with the next permanent state.

After the state merging process ends the algorithm adds self-loops to states which have symbols which are not defined since they are valid continuations based on the label. This ensures that the final DFA is consistent with the labels in the training set.

Algorithm 2 State-Merge-NSP

1: **Input:** Samples and labels S
2: **Output:** Minimum DFA $\mathcal{A}$ consistent with S
3: $\mathcal{A} \leftarrow \text{Prefix-Tree}(S)$
4: PERM $\leftarrow \{q_\lambda\}$
5: CAND $\leftarrow \{q_a : a \in \Sigma \cap \text{PREF}(S)\}$
6: **while** CAND $\neq \emptyset$ **do**
7: $q_c \leftarrow \text{Select}(\text{CAND})$
8: CAND $\leftarrow$ CAND $\setminus q_c$
9: *merged* = false
10: **for** each $q_p \in$ PERM **do**
11: **if** $\text{Merge}(q_p, q_c)$ **then**
12: *merged* = true
13: $\text{Make-Candidate}(\mathcal{A})$
14: break
15: **end if**
16: **end for**
17: **if** $\neg$ *merged* **then**
18: $\text{Promote}(q_c)$
19: **end if**
20: **end while**
21: $\text{Add-SelfLoops}(\mathcal{A})$
22: **return** $\mathcal{A}$

Algorithm 3 Fold

1: **Input:** q_p, q_c
2: **if** $F(q_p) \neq F(q_c)$ **then**
3: **return** False
4: **end if**
5: **for** each σ in Σ **do**
6: **if** $\varphi(q_p, \sigma) \neq \varphi(q_c, \sigma)$ **then**
7: **return** False
8: **end if**
9: **end for**
10: **for** each σ in Σ **do**
11: **if** $\delta(q_c, \sigma)$ is defined **then**
12: **if** $\delta(q_p, \sigma)$ is defined **then**
13: **return** Fold($\delta(q_p, \sigma), \delta(q_c, \sigma)$)
14: **else**
15: $\delta(q_p, \sigma) \leftarrow \delta(q_c, \sigma)$
16: **end if**
17: **end if**
18: **end for**
19: return $\mathcal{A}$ with $Q \setminus q_c$

Remarks. There are a few things to note here. First see that, since F-Tree is disconnected from the DFA, the states within F-Tree are either folded into some state in the main DFA or a transition is added from a state in the main DFA to a state in F-Tree. Secondly, while the merging process begins with folding a candidate state into a permanent state, further iterations could fold non-candidate states in F-Tree into non-permanent states in the main DFA.

Consistency. By construction, the state merging algorithm keeps the DFA consistent with labels in the training set. As described earlier, the prefix tree DFA constructed initially is consistent with the training set (with the addition of self-loops).

Let's say after a merge between q_p and q_c, the labels of a particular example are inconsistent with the resulting DFA. We show that this event cannot occur. First, the examples that do not go through q_c remain unaffected since only the states and transitions in the F-Tree (q_c and its children) are altered. Given an example $x = s_1, \ldots, s_k$ such that $s_1, \ldots, s_{i-1}$ leads to the state q_c, it is straightforward to see that the prefix $s_1, \ldots, s_{i-1}$ still remains consistent since only the transition for the symbol s_{i-1} is changed to q_p instead of q_c. By construction, the labels for both q_p and q_c have to be the same for the merge to succeed. For the remaining suffix, let's consider two parts in general terms: the first part is the substring $s_i, \ldots, s_j$ such that the transitions from both q_p and q_c are defined. The second part $s_{j+1}, \ldots, s_k$ is not defined in the path from q_p. In other words, $\delta(\delta(q_p, (s_i, \ldots s_j)), s_{j+1})$ is not defined whereas $\delta(\delta(q_c, (s_i, \ldots s_j)), s_{j+1})$ is defined.

For the first part, note that for any substring $u_1, u_2, \ldots, u_k$ such that both the sequence of transitions $q_p^{(2)} = \delta(q_p, u_1), q_p^{(3)} = \delta(q_p^{(2)}, u_2), \ldots, q_p^{(k)} = \delta(q_p^{(k-1)}, u_k)$ and $q_c^{(2)} = \delta(q_c, u_1), q_c^{(3)} = \delta(q_c^{(2)}, u_2), \ldots, q_c^{(k)} = \delta(q_c^{(k-1)}, u_k)$ are defined, a merge between q_p and q_c succeeds only when $F(q_p^{(i)}) = F(q_c^{(i)})$ and $\varphi(q_p^{(i)}, \sigma) = \varphi(q_c^{(i)}, \sigma)\, \forall \sigma \in \Sigma$ for all $i = 1, \ldots, k$. Hence, the substring $s_i, \ldots, s_j$ in x cannot be inconsistent after the merge operation. For the second part, $s_{j+1}, \ldots, s_k$, the algorithm adds a transition is added from a state in the main DFA to the corresponding state (for prefix $s_1, \ldots, s_{j+1}$) in the F-Tree. Hence, the states and transitions in the path for this suffix remain unaltered and this cannot lead to inconsistency.

Algorithm 4 Merge

1: **Input:** q_p, q_c
2: Let $\delta(q_f, \sigma)$ be such that $\delta(q_f, \sigma) = q_c$
3: $\delta(q_f, \sigma) = q_p$
4: FOLD(q_p, q_c)

The above argument also holds when either of the two parts is an empty string that covers the other two specific cases for the suffix $s_i, \ldots, s_k$ where the entire suffix falls into one of the above two categories.

Minimum. To see that the algorithm leads to the DFA with the minimum number of states, see that whenever the merge between two states fails, it is guaranteed that there exists a string that distinguishes the two states. If the merge fails because one of the states is an accept state while the other is a reject state then we have a distinguishing string. If the merge fails because for a pair of states $\varphi(q_a, \sigma) \neq \varphi(q_b, \sigma)$, then it is guaranteed that a suffix exists such that it leads to the accept state from one of the states and to the reject state from the another.

By construction, a candidate state is only promoted to a permanent state after it is guaranteed that a distinguishing string exists for every existing permanent state. Hence, with only permanent states remaining in the end, it is guaranteed that the number of states cannot be reduced any further.

Complexity. Given m examples where L is the length of the longest string, the prefix tree construction can take at most $\mathcal{O}(mL)$ time. In the worst case, a merge can be attempted between every two states which could take $\mathcal{O}(m^2L^2)$ computational steps. Since the consistency check for each merge attempt can take at most $\mathcal{O}(mL)$ time, the total time complexity of the algorithm is $\mathcal{O}(m^3L^3)$.

□

Corollary C.1.1. *The concept class of* DFA_n *is efficiently PAC-learnable in the Next Symbol Prediction setting.*

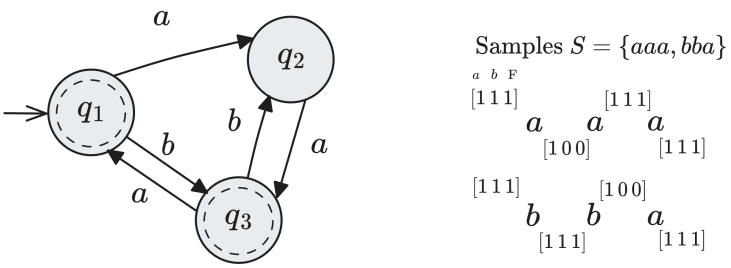


Figure 1: The target DFA and a set of labelled examples for an example run of the algorithm. See Section C.1 for more details.

This immediately follows from the fact that the minimum consistent DFA can be obtained in $\mathcal{O}(m^3L^3)$ time. Using Lemma B.1, we have that the algorithm to compute the minimum consistent DFA PAC-learns the class of DFAs with at most n states in $\mathcal{O}((n \log n)^3 L^3)$ time.

C.1 A Simulation of the Algorithm

We go over an example run of the state-merging algorithm (Alg. 2) to help make it clearer. The target DFA and the set of samples for the simulation are given in Figure 1. The target DFA has three states where q_1 is the start state and the alphabet $\Sigma = \{a, b\}$. The two final states q_1 and q_3 are marked with a dotted inner circle. We only have two samples $S = \{aaa, bba\}$ both of which lead to the final state. In the NSP setting, the algorithm is provided with $|\Sigma| + 1$ labels for each prefix (labelled according to the target DFA) and the corresponding labels for each prefix of inputs are shown in Figure 1 (right). For each prefix, a label vector is shown where the first two labels indicate whether the symbols a and b are allowed as a suffix or not and the last label indicates whether the prefix is accepted by the DFA or not. See Section 2 for more details about problem setting.

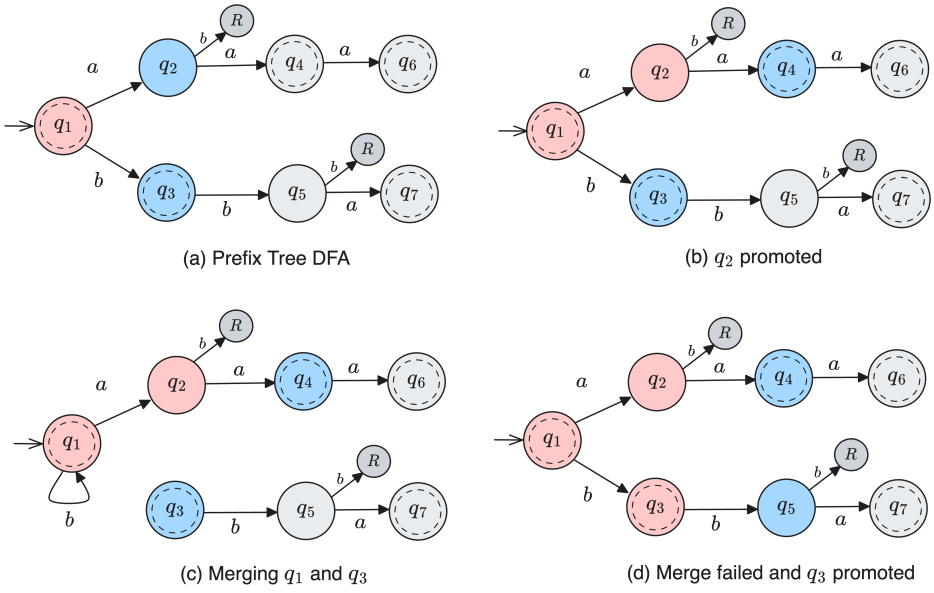


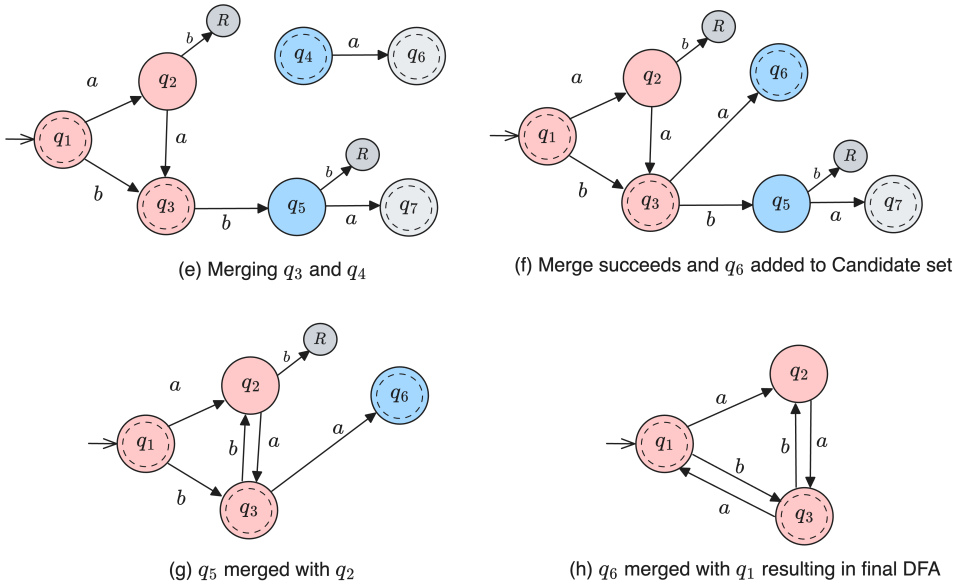
Figure 2: A part of simulation of the state-merging Algorithm 2. The algorithm starts with the prefix-tree DFA. The permanent states are marked as red and the candidate states are marked as blue. The algorithm iteratively tries to merge the candidate states with the permanent states. If a candidate state cannot be merged with any permanent state, it is promoted to a permanent state. See Section C.1 for more details.

Prefix Tree. Algorithm 2 begins with constructing the prefix-tree DFA (Line 3) using the labelled examples (see Algorithm 1). The prefix tree DFA is depicted in Figure 2(a). The permanent states

PERM $= \{q_1\}$ are marked in red and the candidate states CAND $= \{q_2, q_3\}$ are marked in blue. The states are numbers according to the length-lexicographic order described in Section B. Note that the accept and reject states are marked based on labels and the transitions to the reject state q_{dead} are also added based on the labels.

State-Merging. The state-merging process (Lines 7-20) begins after that. The algorithm first tries to merge the states q_2 and q_1 but since q_1 is an accept state and q_2 is a reject state ($F(q_1) \neq F(q_2)$), the merge fails. Since there is no other permanent state to try and merge with, the state q_2 is promoted to a permanent state and its outgoing neighbour q_4 is added to the set of candidate states (see Figure 2(b)). The algorithm then tries to merge the states q_1 and q_3. As defined in the Algorithm 4, the incoming transition to the state q_3 is first changed to q_1 and then the algorithm tries to fold the states q_1 and q_3 (see Figure 2(c)). Since both $\delta(q_1, b)$ and $\delta(q_3, b)$ are defined, the algorithm tries to fold the respective states. However, since $q_1 = \delta(q_1, b)$ and $q_5 = \delta(q_3, b)$ are both not either accept or reject states, the merge fails. The merge between q_3 and q_2 (the other permanent state) fails as well since $F(q_2) \neq F(q_3)$. Hence, the state q_3 is promoted and its outgoing neighbour q_5 is added to the set of candidate states (see Figure 2(d)).

The next candidate state in the set is q_4. The merge between q_1 and q_4 fails because $F(\delta(q_1, a)) \neq F(\delta(q_4, a))$ and the merge with q_2 fails since $F(q_2) \neq F(q_4)$. The algorithm then tries to merge the state q_4 with q_3 (Figure 3(e)) which succeeds and q_6 is added to the set of candidate states (Figure 3(f)). Similarly the merge between states q_2 and q_5 succeeds where the state q_5 is folded with q_2 and the state $q_7 = \delta(q_5, a)$ is folded with $q_3 = \delta(q_2, a)$ (see Figure 3(g)). Lastly, the state q_6 is merged with q_1 which leads to our final DFA which has no more candidate states. Since the transitions from all the states in the final DFA are defined, we do not need to add self-loops (line 21 in Alg. 2). Hence, the algorithm returns the DFA in Figure 3(h) as its output.



(e) Merging q_3 and q_4

(f) Merge succeeds and q_6 added to Candidate set

(g) q_5 merged with q_2

(h) q_6 merged with q_1 resulting in final DFA

Figure 3: A part of simulation of the state-merging Algorithm 2. The algorithm merges the candidate states with the permanent states or promotes the candidate states to permanent states. When no candidate states are left, the algorithm returns the DFA as its output. See Section C.1 for more details.

D Learning Boolean acyclic DFAs

The concept class of Boolean Acyclic DFAs (B-DFA) contains DFAs which represent functions of the form $f : \{0,1\}^* \to \{0,1\}$ where for all $x \notin \{0,1\}^n$, the function $f(x) = 0$. Hence, the class B-DFA can be seen as a class of functions comprising Boolean functions which are of the form $\{0,1\}^n \to \{0,1\}$. Note that with only positive and negative examples as in the standard classification setting, the problem of finding the minimum consistent DFA for Boolean acyclic DFAs is intractable under cryptographic assumptions [Kearns and Valiant, 1994].

Theorem D.1. *In the next symbol prediction setting, given m examples labeled by any $\mathcal{A} \in \text{B-DFA}_n$, the minimum consistent DFA can be computed in time $\mathcal{O}(m^2 L)$ where L is the length of the longest string in the m examples.*

Proof. We show that for any given m examples and labels obtained according to the example oracle in the NSP setting, the minimum consistent DFA can be obtained in $O(m^2 L)$ time. The algorithm begins by constructing a prefix tree DFA (as in Algorithm 1) using the samples and labels provided by the example oracle.

Unlike the case of the general class of DFAs, certain properties of the structure are known. For any B-DFA $f \in \text{B-DFA}$ which represent functions over $\{0,1\}^L \to \{0,1\}$, for every input such that $f(x) = 1$, the path from the initial state to the final state will go through exactly L states (excluding the initial state). For any DFA in the class B-DFA, every state can be reached or identified by prefixes of a unique length. In other words, every equivalence class in the language of the DFA (apart from the dead state q_{dead}) has strings of a fixed length.

Once the prefix tree is constructed using Algorithm 1, for every length in $[L]$, there exists a set of states such that they can be reached with a prefix of length $l \in L$. Let $Q(l)$ denote the set of states in the initial prefix tree which can be reached with a prefix of length exactly l. For simplicity, we refer to the states in $Q(l)$ as the states at *depth* l.

State-merging. The state-merging algorithm works as follows: The algorithm iteratively goes through states at each depth ($Q(l)$) from $L, L-1, \ldots, 1$ and tries to merge the states at each depth l. At any depth l, for every state $q^{(l)} \in Q(l)$, it attempts to merge it with every other state in $Q(l)$. The merge between two states $q_a^{(l)}$ and $q_b^{(l)}$ succeeds if two conditions are satisfied. Condition (i): for every $\sigma \in \{0,1\}$, $\varphi(q_a^{(l)}, \sigma) = \varphi(q_b^{(l)}, \sigma)$ which is to say that two states cannot be merged if a symbol leads to the dead state from one of them and to a non-dead state from the other. Condition (ii): for any $\sigma \in \{0,1\}$ if both the transitions $\delta(q_a^{(l)}, \sigma)$ and $\delta(q_b^{(l)}, \sigma)$ are defined, then the merge can only succeed if $\delta(q_a^{(l)}, \sigma) = \delta(q_b^{(l)}, \sigma)$.

Fold. If the merge between two states $q_a^{(l)}$ and $q_b^{(l)}$ succeeds then the state $q_b^{(l)}$ is folded into state $q_a^{(l)}$. The fold consists of two changes. (a) For any $\sigma \in \{0,1\}$ such that $\delta(q_b^{(l)}, \sigma)$ was defined and $\delta(q_a^{(l)}, \sigma)$ was not defined, the algorithm adds a transition $\delta(q_a^{(l)}, \sigma) = \delta(q_b^{(l)}, \sigma)$. (b) For all states $q^{l-1} \in Q(l-1)$ such that $\delta(q^{l-1}, \sigma) = q_b^{(l)}$, the transitions are replaced by $\delta(q^{l-1}, \sigma) = q_a^{(l)}$.

The state-merging algorithm iteratively merges the states at each depth starting from the final state at depth L to states at depth 1. After the state-merging process is complete, for every remaining state $q \in Q_o$, at least one of σ in $\{0,1\}$ will have a defined transition to the next state. For all states $q \in Q_o$ such that $\delta(q,0)$ is not defined (which also indicates that $\varphi(q,0) = 1$, we add $\delta(q,0) = \delta(q,1)$. We do the same with states where $\delta(q,1)$ is not defined.

This results in the minimum consistent DFA for a given set of examples sampled according to the examples oracle in the NSP setting. By the construction, the DFA always remains consistent throughout the state-merging process since two states are only merged if they are indistinguishable based on the sample set. The resulting DFA is minimal since we know a distinguishing string exists between every pair of states in the DFA. First, see that two states at different depths cannot be merged because for $i \neq j$, every state at depth $L - i$ has a suffix that is accepted whereas for all states at depth $L - j$ every suffix of length i leads to a reject state. For every pair of states q_1 and q_2 at the same depth $1 \leq l \leq L$, the merge fails if a distinguishing string is guaranteed to exist. The merge fails, if for a symbol $\delta(q_1, \sigma) \neq \delta(q_2, \sigma)$ when both the transitions are defined. Since, the symbol leads to transitions to different states, by definition there is a suffix that distinguishes the two states. In the other case where the merge fails, we know that a symbol leads to the dead state from one of the states and not from the other. Consequently, we know that a suffix starting with that symbol exists that leads to an accept state from only one of the states.

Complexity. Given m examples, see that after the prefix tree construction, the number of states at each depth $|Q(l))| \leq m$ and hence the number of merge operations at each depth will be $O(m^2)$. Since we are iterating over L, the time complexity of the state-merging algorithm is $O(m^2 L)$.

□

Corollary D.1.1. *The concept class of* $\mathrm{B\text{-}DFA}_n$ *is efficiently PAC-learnable in the Next Symbol Prediction setting.*

Similar to the previous case, it immediately follows from the fact that the algorithm described above finds the minimum consistent hypothesis in $\mathcal{O}(m^2L)$ time, and given the size of the hypothesis class (Lemma B.1), the algorithm PAC-learns the concept class $\mathrm{B\text{-}DFA}_n$ in $\mathcal{O}((n \log n)^2 L)$ time.