Learning MSO-definable hypotheses on strings

Martin Grohe GROHE@INFORMATIK.RWTH-AACHEN.DE
Christof Löding LOEDING@INFORMATIK.RWTH-AACHEN.DE
Martin Ritzert RITZERT@INFORMATIK.RWTH-AACHEN.DE
RWTH Aachen University
Germany

Editors: Steve Hanneke and Lev Reyzin

Abstract

We study the classification problems over string data for hypotheses specified by formulas of monadic second-order logic MSO. The goal is to design learning algorithms that run in time polynomial in the size of the training set, independently of or at least sublinear in the size of the whole data set. We prove negative as well as positive results. If the data set is an unprocessed string to which our algorithms have local access, then learning in sublinear time is impossible even for hypotheses definable in a small fragment of first-order logic. If we allow for a linear time pre-processing of the string data to build an index data structure, then learning of MSO-definable hypotheses is possible in time polynomial in the size of the training set, independently of the size of the whole data set.

1. Introduction

We study classification problems in a declarative framework (introduced in Grohe and Turán, 2004; Grohe and Ritzert, 2017) where instances are elements or tuples of elements of some background structure and hypotheses are specified by formulas of a suitable logic, using parameters (or constants) from the background structure. The background structure, say B, captures properties of and relations between data points and more generally all kinds of structural information about the data. Over this background structure we can specify a parametric model by a formula $\varphi(\bar{x};\bar{y})$ of some logic L, which has two types of free variables, the *instance variables* $\bar{x} = (x_1, \dots, x_k)$ and the *parameter variables* $\bar{y} = (y_1, \dots, y_\ell)$. Then *instances* of our classification problem are tuples $\bar{u} \in U(B)^k$, where $U(B)$ denotes the universe of B. For each choice $\bar{v} \in U(B)^\ell$ of *parameters*, the formula defines a function $[\![\varphi(\bar{x}\,;\bar{v})]\!]^B : U(B)^k \to \{0,1\}$ by

$$[\![\varphi(\bar{x}\,;\bar{v})]\!]^B(\bar{u}) := \begin{cases} 1 & \text{if } B \models \varphi(\bar{u}\,;\bar{v}), \\ 0 & \text{otherwise,} \end{cases}$$

where $B \models \varphi(\bar{u}\,;\bar{v})$ denotes that the structure B satisfies φ if the instance variables $\bar{x}$ are interpreted by $\bar{u}$ and the parameter variables $\bar{y}$ by $\bar{v}$. We regard $[\![\varphi(\bar{x}\,;\bar{v})]\!]^B$ as a hypothesis over the instance space $U(B)^k$, which we want to generate from a training set of labeled

examples $(\bar{u}_i, \lambda_i) \in U(B)^k \times \{0,1\}$.[1] We call hypotheses of the form $[\![\varphi(\bar{x}\,;\bar{v})]\!]^B$ for an L-formula $\varphi(\bar{x}\,;\bar{y})$ L-*definable hypotheses* over B.

In this paper, background structures are *strings*, which may model text data, but also traces of program executions, DNA sequences, transaction sequences, and in general streams of symbolic data. The logic L that we use to define our models is *monadic second-order logic* MSO, which may be the best studied logic for strings and is closely related to finite automata (see Thomas, 1997). Some of our results, in particular the lower bounds, hold for fragments of MSO such as first-order logic FO and even the existential and quantifier-free fragments of FO.

Within this framework, we may study two kinds of algorithmic problems, *parameter learning* (or *parameter estimation*), where we regard the formula $\varphi(\bar{x}\,;\bar{y})$ as fixed and try to find parameters $\bar{v}$ that fit the data, and *model learning* (or *model estimation*), where we want to find a suitable formula $\varphi(\bar{x};\bar{y})$ and parameters $\bar{v}$. Our algorithms follow an empirical risk minimization paradigm; for the model learning problem, we bound the quantifier rank of the formula $\varphi(\bar{x}\,;\bar{y})$ to avoid overfitting. Hence the algorithmic problem we need to solve is finding a parameter tuple $\bar{v}$, and for the model learning problem a formula $\varphi(\bar{x}\,;\bar{y})$, such that the hypothesis $[\![\varphi(\bar{x}\,;\bar{v})]\!]^B$ is consistent with, or minimizes the error on the training examples.

The input of the learning algorithms (both for parameter and model learning) consists of a set $T = \{(\bar{u}_1, \lambda_1), \ldots, (\bar{u}_t, \lambda_t)\}$ of labeled examples, but the algorithms also need access to the background structure B. We usually think of B as being very large, and we want to avoid holding it in main memory or even looking at the whole structure. That is, we are looking for learning algorithms with a running time that is polynomial in t (the number of training examples), but sublinear in the background structure B under a reasonable model of accessing B. In (Grohe and Ritzert, 2017), the background structure B is a graph, presumably of small degree, and the learning algorithms only have *local access* to B, that is, they can only retrieve the neighbors of vertices that they already hold in memory. Initially, these are the vertices appearing in the training examples. The main result of (Grohe and Ritzert, 2017) is that model learning for first-order logic is possible in time polynomial in the number t of training examples and the maximum degree d of the background graph B. The strings that we study as background structures in this paper are equipped with the $\leq$-relation which is of unbounded degree. Hence the results of (Grohe and Ritzert, 2017) do not apply in this setting. The polynomial that bounds the running time depends on the quantifier rank q of the formula $\varphi(\bar{x}\,;\bar{y})$ and the lengths k, ℓ of the tuples $\bar{x}, \bar{y}$. The crucial point is that this running time is independent of the size n of the background structure (in a uniform cost model; otherwise it is poly-logarithmic in n).

1.1. Our Results

For the strings studied as background structures in this paper, we have also have a natural notion of local access: algorithms are only allowed to (directly) access the successor and predecessors of positions of a string that they already hold in memory. Our first result (Theorem 2) is negative: we prove that every (model or parameter) learning algorithm

1. As such, this framework only allows it to describe binary classification problems, but it is easy to extend it to general classification problems.

producing an FO-definable hypothesis consistent with the training examples (if there is one) necessarily needs time at least linear in n. Only if $\varphi(\bar{x}\,;\bar{y})$ is quantifier-free (Theorem 6) or existential with only one instance variable, that is, $k = 1$, (Theorem 7) we obtain a model learning algorithm for FO running in time polynomial in t, independently of n. We can strengthen our linear lower bound in such a way that it already applies to existential FO-formulas with two free instance variables (Theorem 8).

The negative results are not very surprising, because the local access model to the background string B is extremely restrictive. For example, it is impossible for an algorithm to find the first position in B that is labeled by symbol 'a'. We also consider a less restrictive access model, where we allow a linear time pre-processing of the background string to build an index data structure that allows for more global access to B. The pre-processing takes place before the algorithm sees the training examples. Then in the actual learning phase, the algorithm only has local access to B and the index structure, that is, is only allowed to follow pointers. Our main result (Theorem 9) states that after such a linear time (in n) pre-processing phase, both parameter and model learning for MSO-definable hypotheses are possible in time polynomial in t.

Technically, this theorem heavily relies on the connections between monadic second-order logic, finite automata, and semi-group theory. The index data structure we built in the pre-processing phase is the Simon Factorization Forest (Simon, 1990; Kufleitner, 2008) for a suitable monoid associated with MSO-definable hypotheses.

1.2. Related Work

Closely related to our framework is that of inductive logic programming (ILP) (see, for example, Cohen and Page, 1995; Kietz and Dzeroski, 1994; Muggleton, 1991, 1992; Muggleton and Raedt, 1994). The two main differences are that we encode background knowledge in a background structure, whereas the ILP framework axiomatizes it in a background theory, and that we work with MSO, whereas ILP focuses on FO, possibly in a recursive setting. Other recent logical frameworks for machine learning, mainly in the context of database and verification applications can be found in (Abouzied et al., 2013; Bonifati et al., 2016; Löding et al., 2016; Garg et al., 2016; Jordan and Kaiser, 2016).

There are also numerous results on learning automata and regular languages, negative (Angluin, 1978; Gold, 1978; Pitt and Warmuth, 1993; Kearns and Valiant, 1994; Angluin, 1990) as well as positive (Angluin, 1987; Rivest and Schapire, 1993; Kearns and Vazirani, 1994; Oncina and García, 1992), the latter mainly in an active learning framework. Technically, all these results seem unrelated to ours.

2. Preliminaries

We considers strings (or words) over an alphabet Σ. The set of all such finite strings is denoted by Σ^*, and the empty word by ε. In the logical setting, we view words as structures B over the signature $\tau = \{<, (R_a)_{a\in\Sigma}\}$ with universe $U(B) = \{1, \ldots, n\}$ for words of length n. We also refer to the elements of $U(B)$ as positions of the word. The relation $<$ is the natural ordering of the positions, and each R_a is a unary predicate for the a-labeled positions. Note that we do not have a direct successor relation for positions (this is only relevant for the results in Section 4 which can be extended to include the successor relation

but become much more technical in that setting). In general, we do not distinguish between the relational representation and the sequence of alphabet symbols.

We use standard first-order logic (FO) over these word structures. Monadic second-order logic (MSO) extends FO by additional quantification over sets of positions. We use lowercase letters x, y, z to denote first-order variables and the corresponding uppercase letters for set variables. The *quantifier rank* of a formula φ is the maximal number of nested quantifiers in the formula.

We refer to the introduction for the basic definitions on our learning model. For a formula $\varphi(\bar{x}\,;\bar{y})$ we define the *arity* of $\varphi(\bar{x}\,;\bar{y})$ to be the number of instance variables in $\bar{x}$. For the case of a single instance variable x we speak of *unary* formulas.

For a word B, a training set $T \subseteq U(B)^k \times \{0,1\}$ is called *φ-consistent* if there are parameters $\bar{v} \in U(B)^\ell$ such that for all $(\bar{u}, \lambda) \in T$ the classification is $\lambda = [\![\varphi(\bar{x}\,;\bar{v})]\!]^B(\bar{u})$. We also say that $\bar{v}$ is consistent with φ, B, and T.

As mentioned in the introduction, we distinguish between parameter learning and model learning. For the parameter learning problem, we assume a fixed formula $\varphi(\bar{x}\,;\bar{y})$. A *parameter learner* $\mathcal{L}$ for φ has as input a word B and a training set T over B. Using the access model for B explained in the introduction, $\mathcal{L}$ produces as output a tuple $\bar{v} = \mathcal{L}(B, T)$ of parameters. We say that $\mathcal{L}$ is a *consistent parameter learner* for φ if $\mathcal{L}(B, T)$ is consistent with φ, B, T for all possible B and φ-consistent T.

For the model learning problem, the formula $\varphi(\bar{x}\,;\bar{y})$ has to be found by the algorithm. However, we do fix a maximum quantifier rank q and the number $\ell = |\bar{y}|$ of parameters. Note that the arity $k = |\bar{x}|$ is also fixed, because the instance space of our learning problem is $U(B)^k$. The view that q, k, ℓ are fixed, which will be important for our complexity analysis, is motivated by an analogy with database theory. Our whole declarative learning framework is inspired by the declarative framework of relational database systems. The logic in which we specify our models and hypotheses corresponds to the database query language, and the background structure corresponds to the database. It is common in database theory to analyze the complexity of the query evaluation problem, which corresponds to the learning problems we consider here, by regarding the query as fixed and the database as the variable input; this perspective is known as *data complexity* (Vardi, 1982). The data complexity approach is justified by arguing that queries are usually human-written and not too large, certainly in comparison with the size of the data, and that therefore we can treat the query size as constant. Similarly, if we aim for explanatory and human-understandable models in machine learning, and want to avoid over-fitting, we may want to restrict the quantifier rank and the number of the parameters of the formulas defining the models. Adapting the database terminology, we may say that in this paper we analyze the *data complexity* of parameter and model learning.

Now if we fix q, k, ℓ, there is only a finite number of formulas $\varphi(\bar{x}\,;\bar{y})$ that our algorithm needs to consider, because up to logical equivalence there is a only a finite number of MSO-formulas of quantifier rank at most q with at most $k + \ell$ free variables. We may actually assume that there is a fixed formula $\varphi(\bar{x}\,;\bar{y})$ of quantifier rank q and with $|\bar{y}| = \ell$. Then the goal of our model learning algorithm is to compute, given a training set T and structure B such that T is φ-consistent, a formula $\varphi'(\bar{x}\,;\bar{y}')$ and a parameter tuple $\bar{v}'$ such that $\bar{v}'$ is consistent with φ', B, and T. Furthermore, for simplicity we may assume that the learning algorithm "knows" $\varphi(\bar{x};\bar{y})$. These assumptions are justified by the observation that

otherwise the algorithm can iterate through all the (finitely many) possible formulas. Note that this makes model learning a simpler problem than parameter learning (in the sense that an algorithm for parameter learning yields a model learning algorithm). In (Grohe and Ritzert, 2017, Section 3), there is a simple example illustrating that model learning can be strictly simpler (also see Example 1 below). We may further allow the formula $\varphi'(\bar{x}\,;\bar{y}')$ to have a larger quantifier rank $q' \geq q$ and a larger number $\ell' := |\bar{y}'| \geq \ell$ of parameter variables than φ. We refer to such a learning algorithm $\mathcal{L}$ as a (q',ℓ')-*formula learner*. We say that $\mathcal{L}$ is a *formula learner* for φ if it is a (q',ℓ')-formula learner for φ for some numbers q',ℓ'. Note that each parameter learner for φ is also a (q,ℓ)-formula learner for φ.

Example 1 *Suppose our background string B is over the alphabet $\Sigma = \{a,b\}$.*

Let $\varphi_1(x\,;y) = R_a(x) \wedge x \leq y$. Let $\mathcal{L}_1$ be the algorithm that, given a training set T (and local access to B), returns the largest position v such that $(v,1) \in T$. It is easy to see that $\mathcal{L}_1$ is a consistent parameter learner for φ_1.

Now consider $\varphi_2(x\,;y) = R_a(x) \wedge R_b(y) \wedge x \leq y$. Then a consistent parameter learner for φ_2 has to search for the first position v in B that is labeled by b and is greater than or equal to all u such that $(u,1) \in T$. It may take time linear in $|B|$ to find such a v.

However, it is easy to construct a $(0,1)$-formula learner $\mathcal{L}_2$ for φ: given T, it returns the formula $\varphi_1(x\,;y)$ and as parameter the largest position v such that $(v,1) \in T$.

We only require our learning algorithms to return hypotheses consistent with the training set. In fact, all of our algorithms can be generalized in such a way that they return a hypothesis with minimum training error if there is no consistent one (we leave the details to the full version of this paper).

To justify that such learning algorithms return hypotheses that generalize well, we appeal to the standard result from PAC-learning (also see Section 3.1): A consistent learner is also a PAC-learner over a hypothesis space with bounded VC-dimension using a training sequence of size polynomial in the VC-dimension and the constants of the error bounds (see Shalev-Shwartz and Ben-David, 2014). The following theorem shows that our hypothesis space is of bounded VC-dimension.

Theorem 1 (Grohe and Turán (2004)) *For q,k,ℓ there is a d such that for every string B the family of all hypotheses $[\![\varphi(\bar{x}\,;\bar{v})]\!]^B$, where $\varphi(\bar{x}\,;\bar{y})$ is an MSO-formula of quantifier rank at most q, $|\bar{x}| = k, |\bar{y}| = \ell$, and $\bar{v} \in U(B)^\ell$, has VC-dimension at most d.*

3. Non-learnability of unary FO-definable concepts

Theorem 2 *There is no consistent formula learner for unary FO formulas whose running time is sublinear in the length of the string.*

The theorem follows immediately from the following lemma.

Lemma 3 *There is an FO formula $\varphi(x\,;y) = \exists z \forall z' \psi(z,z',x,y)$ with quantifier-free ψ such that for all (q,ℓ)-formula learners $\mathcal{L}$ with a sublinear running time the following holds. There is a string B and a φ-consistent training set T of size $|T| = 2\ell+3$ such that the hypothesis H_T produced by $\mathcal{L}$ on input B and T is not consistent with T.*

Proof We consider strings over the alphabet $\Sigma = \{a, b, c\}$. We view strings over Σ as consisting of *a-blocks* (sequences of successive a-positions) that are separated by b or c. The *entry* of an a-block is the position directly before that block, which is then labeled b or c (if the string starts with a, then the first block does not have an entry). The formula $\varphi(x\,;y)$ selects those a-positions whose block entry is either before y and labeled b, or behind (including) y and labeled c. So the behavior of the formula switches at the parameter position. The concrete formula is

$$\varphi(x\,;y) = R_a(x) \wedge \exists z(z < x) \wedge ((R_b(z) \wedge z < y) \vee (R_c(z) \wedge z \geq y)) \\ \wedge \forall z'((z < z' < x) \to R_a(z')) \tag{1}$$

Let $\mathcal{L}$ be a (q, ℓ)-formula learner for arbitrary numbers q, ℓ whose running time is sublinear in the length of the string.

We choose s such that for every MSO formula with ℓ parameters and quantifier rank at most q, there is an equivalent finite automaton with at most s states. Then we choose r in such a way that the running time of the learner $\mathcal{L}$ on an input string B of length

$$n = (2\ell + 3)(3s!)(2r + 2)$$

is at most r. This choice of r is possible since the runtime of $\mathcal{L}$ is sublinear in the length of the input string.

Next, we construct strings $B_0, \ldots, B_\ell$ of length n and parameter positions v_i such that $\varphi(x\,;y)$ selects the same set of positions for all B_i with parameter v_i. These strings have the following shape:

$$B_i = (A_b A_c)^i A_b (A_b A_c)^{\ell+1-i},$$

where the strings A_b and A_c are defined as $A_b = (ba^{2r+1})^{3s!}$ and $A_c = (ca^{2r+1})^{3s!}$.

Each B_i contains two successive A_b. The parameter position v_i for B_i is chosen to be the first b in the second of the two successive A_b. Since the behavior of φ flips at the parameter position, it should be clear that φ indeed selects the same set of positions for each B_i with parameter v_i, or formally, $[\![\varphi(x, v_i)]\!]^{B_i} = [\![\varphi(x, v_j)]\!]^{B_j}$ for all i, j. This allows us to construct a single training set that is φ-consistent over all B_i. The training set T contains one position in each substring A_b or A_c of B_i. The positions are chosen in the middle of an a-block (of length $2r+1$), which itself is in the middle of all the a-blocks of the respective substring A_b or A_c. Formally, these are the positions $|A_b| \cdot j + \frac{|A_b|}{2} + r + 2$ for $j \in \{0, \ldots, 2\ell+2\}$ (note that $|A_b| = |A_c|$ so we do not need to distinguish them in the definition of the positions). The classification of these positions starts with 1 and then alternates between 0 and 1 (identical for every B_i).

By the choice of r, the algorithm $\mathcal{L}$ only has access to a-positions when it is executed on B_i with training set T. Hence, it does not see any difference between the B_i and produces the same hypothesis for all B_i. In particular this includes the same set of at most ℓ parameters since these are part of the hypothesis. So there is one string B^* for which no parameter is inside the two successive A_b substrings. We now show that the hypothesis on this B^* cannot distinguish the two positions in the two successive A_b substrings, whereas their classification in the training set is different. This shows that the hypothesis produced by $\mathcal{L}$ on input B^* and T is not consistent with T.

The hypothesis cannot distinguish the examples from the A_bA_b block. To show this we observe that the formula learner $\mathcal{L}$ produces a hypothesis $\psi(x, \bar{v}')$ for B_i and T (we use $\bar{v}'$ for the parameters because the notation v_i is already in use in this proof).

For the hypothesis formula $\psi(x, \bar{y})$ there is an equivalent DFA (deterministic finite automaton) $\mathcal{A}$. This DFA reads words over Σ that are annotated in some form to mark the position u of the instance variable, and the positions of the parameters. In particular, it accepts B_i annotated with a position u and the positions $\bar{v}'$ from the hypothesis if, and only if, $\psi(u, \bar{v}')$ holds in B_i. By the choice of s, there is such a DFA $\mathcal{A}$ with at most s states.

We show that this DFA wrongly classifies one of the examples from the two successive A_b subwords in B_i. More formally, let $u_1 < u_2$ be the positions from the training set that fall into the two successive A_b subwords of B_i. Let B_i^1 and B_i^2 be the strings annotated with the parameters $\bar{v}'$ and additionally B_i^1 is annotated with u_1 as instance for x, and similarly for B_i^2 and u_2. We know that these two positions are classified differently in the training set. Note also that B_i was chosen such that no parameter is inside the two successive A_b's.

We analyze the runs of $\mathcal{A}$ on B_i^1 and B_i^2 on the two successive A_b subwords. The idea is illustrated in Figure 1. Let A' denote A with the middle a-position carrying the marker for the position of x, and let $A^* = A^{\frac{1}{2}s!}A'A^{\frac{1}{2}s!-1}$. Then the two successive A_b subwords in B_i^1 and B_i^2 including the markers for u_1 and u_2 are of the form $C_1 = A^{s!}A^*A^{4s!}$ and $C_2 = A^{4s!}A^*A^{s!}$. This implies that the state before A^* is the same in both runs: After reading the first $s!$ copies of A, the state is some σ_1 for both words (up to this point, B_i^1 and B_i^2 are the same). Then B_i^1 is followed by A^*. In B_i^2, the DFA reads another $3s!$ copies of A. Since $s!$ is a multiple of every possible length of a loop in $\mathcal{A}$, and $\mathcal{A}$ has already read $s!$ copies of A, the state before A^* in B_i^2 is also σ_1. The same argument is used to show that both runs end on the same state σ_2 after having read C_1 and C_2, respectively.

Therefore, $\mathcal{A}$ cannot distinguish C_1 and C_2 and thus accepts both, B_i^1 and B_i^2, or rejects both. This means that $\mathcal{A}$ will either accept or reject both training examples u_1 and u_2.

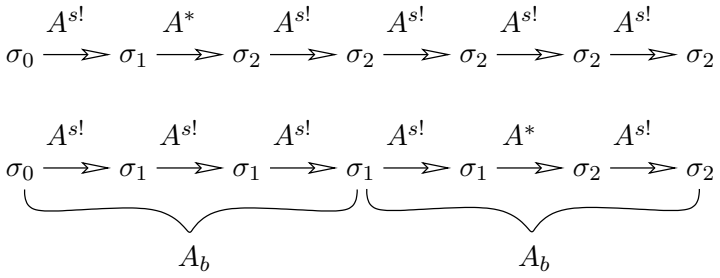

Figure 1: State transitions of $\mathcal{A}$ on the A_bA_b block for the two different training examples

■

3.1. PAC Learning

The general aim in machine learning is to perform well on unseen examples. This is formalized in Valiant's probably approximately correct learning model. The idea is that a learning algorithm is *probably approximately correct* (PAC), if over most of the training sets ('probably') it outputs a classification algorithm (or hypothesis) which has a low expected error on new examples ('approximately'). To get any bounds on the expected error, a fixed underlying but unknown probability distribution $\mathcal{D}$ over the examples is assumed. Then the examples are chosen independently from this distribution $\mathcal{D}$. When talking about training sets, we implicitly assume that those are chosen independently according to some unknown but fixed distribution $\mathcal{D}$.

Here we show that there is no PAC learning algorithm with sublinear runtime.

Definition 4 *Let $\mathcal{L}$ be a learning algorithm which outputs on input of B and T a hypothesis H_T. Then $\mathcal{L}$ is* probably approximately correct (PAC) *if for all probability distribution $\mathcal{D}$ over the instance space*

$$\Pr_{T\sim\mathcal{D}}[\mathrm{err}_{\mathcal{D}}(H_T) < \epsilon] > 1 - \delta.$$

Here the probability is taken over the training set T where $T \sim \mathcal{D}$ means that the training examples are chosen independently according to $\mathcal{D}$. Furthermore, $\mathrm{err}_{\mathcal{D}}(H_T)$ is the expected error on (new) examples chosen according to $\mathcal{D}$.

This means that a learning algorithm is *probably approximately correct (PAC)* if with high probability over the choice of the training set T, the expected error of the hypothesis H_T on new instances is low.

Theorem 5 *Let $\mathcal{L}$ be a sublinear (q, ℓ)-formula learner. Then $\mathcal{L}$ is not a PAC learning algorithm.*

Proof We choose a string B and a training set T of size $|T| = 2\ell+3$ according to Lemma 3. Now we let $\mathcal{D}$ be the uniform distribution over the position appearing in T. Then if we draw examples from the distribution $\mathcal{D}$, the learner only sees examples from T, and hence we know that it makes a mistake. There is a small technical issue here: if we draw the examples randomly from T, the learner $\mathcal{L}$ may actually only see a subset $T' \subseteq T$, because some may be repeated. However, without loss of generality we may assume that $\mathcal{L}$ does not perform worse if it sees more examples. Hence if we denote the hypothesis produced by $\mathcal{L}$ on input B and $T' \subseteq T$ by $H_{T'}$, then $H_{T'}$ is not consistent with T, and we have

$$\mathrm{err}_{\mathcal{D}}(H_{T'}) \geq \frac{1}{2\ell+3},$$

because $H_{T'}$ is wrong on at least one of the $2\ell+3$ elements in the support of $\mathcal{D}$. Thus with $\epsilon = \frac{1}{2\ell+3}$ we have

$$\Pr_{T'\sim\mathcal{D}}[\mathrm{err}_{\mathcal{D}}(H_{T'}) < \epsilon] = 0.$$

This implies that $\mathcal{L}$ is not a PAC learning algorithm. ■

4. Quantifier-free and existential formulas

The non-learnability result from Section 3 applies to formulas with at least one quantifier alternation (Lemma 3). In this section we therefore consider simpler classes, namely quantifier-free and existential formulas. A quantifier-free formula ψ consists of atoms checking membership in the relations $\tau = \{<, (R_a)_{a\in\Sigma}\}$ or boolean combinations of those. Existential formulas are of the form $\varphi(\bar{x}) = \exists \bar{z} \psi(\bar{x}, \bar{z})$ where ψ is quantifier-free.

For quantifier-free and unary existential formulas, there are formula learners running in time polynomial in $|T|$.

Theorem 6 *There is a consistent formula learner for unary quantifier-free formulas whose running time is in $\mathcal{O}(|T|)$ for a training set T. For arbitrary quantifier-free formulas with k instance and ℓ parameter variables there is a consistent formula learner running in time $\mathcal{O}((2|T|k+1)^\ell |T|)$.*

Note that even in the more general case, the runtime of the algorithm is polynomial in $|T|$ for a fixed ℓ. The proof uses the fact that evaluating a quantifier-free formula over an ordered word only depends on the labels and relative positions of the nodes assigned to the free variables. The number of possible different configurations therefore only depends on k and ℓ which means that all of those can be checked. For a proof of this theorem we refer to the full version of this article.

In Theorem 7 we generalize the first part of Theorem 6 to unary existential formulas. In contrast to the quantifier-free case, where ℓ can be taken from φ for some φ-consistent training set, the constructed hypothesis uses a relatively long formula. The formula constructed for the hypothesis is based on the observation that existentially quantified conjunctions can only define an interval per label $a \in \Sigma$. Theorem 8 states that this cannot be extended to arbitrary existential formulas. Again, we refer to the full version of the paper for proofs of these theorems.

Theorem 7 *There is a consistent formula learner for unary existential formulas whose running time is in $\mathcal{O}(|T|)$ for a training set T.*

Theorem 8 *There is no consistent formula learner for binary existential formulas with one parameter that runs in sublinear time.*

5. Indexing

While it is impossible to learn the parameters of a fixed formula in sublinear time, we now consider the case that the learning algorithm can preprocess and index the underlying string before it enters the learning phase for this string. In this setting, the learning algorithm consists of two phases, starting with a linear time (in $|B|$) indexing phase, in which the algorithm can add an auxiliary data structure to B. The learning phase (sublinear in $|B|$) then can use this auxiliary data structure to compute consistent parameters $\bar{v}$ for a given training set T. We refer to the running times of these two phases as *indexing time* and *learning time*, respectively.

The main result in this section is about the learnability of unary MSO formulas in the indexing model, as stated in the next theorem. Most of this section is devoted to the proof

of this theorem. At the end of the section we briefly mention the case of MSO formulas with higher arity.

Theorem 9 *There is a consistent parameter learner for unary MSO formulas with indexing time $O(|B|)$ for a string B and learning time $\mathcal{O}(|T|)$ for a training set T over B.*

As an example, consider the formula used in the proof of Lemma 3. The indexing phase could annotate each position with the information whether it is in an a-block preceded by b or by c. With this additional information it is easy to find a consistent parameter setting for a given training set.

For this specific example, it is sufficient to simply annotate each position in B with some extra information. In the general case, the auxiliary data structure is a tree whose leafs are the positions of B, and each node contains information about the substring of B in the subtree below this node. More formally, we use a factorization tree of the string w.r.t. some finite monoid. We start by introducing monoids and factorization trees, and then define the specific monoid that we use in the learning algorithm. We refer the reader to Colcombet (2011) and Bojańczyk (2012) for some recent expositions that explain the connections between monoids and regular languages in more detail.

A *monoid* $(M, \cdot, 1_M)$ consists of a set M, an associative multiplication operation $\cdot$ on M, and a neutral element 1_M for this operation. Often we simply write M for the monoid, and write the multiplication of two elements m_1 and m_2 as $m_1 m_2$. The set of all finite words over an alphabet Σ with concatenation as multiplication and the empty word as neutral element is called the free monoid (generated by Σ). A mapping $h : M \to M'$ for monoids M and M' is called a monoid morphism (just morphism for short) if $h(m_1 m_2) = h(m_1)h(m_2)$ for all $m_1, m_2 \in M$, and $h(1_M) = 1_{M'}$.

For a finite monoid M, a morphism $h : \Sigma^* \to M$, and a subset $F \subseteq M$ of accepting monoid elements, we define the language $L(M, h, F) = \{A \in \Sigma^* \mid h(A) \in F\}$. A well known theorem implicitly mentioned in early papers of automata theory by Rabin and Scott (1959) states that a language is regular if, and only if, it can be accepted by a finite monoid in this way.

We now turn to factorization trees. These can be seen as index structures for finite words. Our techniques are based on the same ideas as the ones described in Bojańczyk (2012).

Let M be a finite monoid, and let $s = m_1, m_2, \ldots, m_n$ be a sequence of elements from M. A *factorization tree* $\mathcal{T}$ of s is a finite ordered tree (the successors of a node are ordered) whose nodes v are labeled by elements $\mathcal{T}(v)$ of M, such that

- the sequence of leaf labels is s,
- for each inner node v with children $v_1, \ldots, v_i$, the label of v is the product of the monoid elements at its children: $\mathcal{T}(v) = \mathcal{T}(v_1) \cdot \mathcal{T}(v_2) \cdots \mathcal{T}(v_i)$

Note that each node v of $\mathcal{T}$ defines an infix (factor) $m_j, ..., m_{j'}$ of s corresponding to the leafs in the subtree below v. The label $\mathcal{T}(v)$ of v is the product $m_j ... m_{j'}$ of these elements.

For example, one can use a binary tree, whose height is then logarithmic in the length of s. We use a class of factorizations introduced by Simon (1990) that also can have nodes of higher arity with a specific property.

A *Simon factorization tree* $\mathcal{T}$ of s is a factorization tree with the following additional property:

- if a node v of $\mathcal{T}$ has more than two children $v_1, \ldots, v_i$, then the labels of all the children are the same, and this label e is an *idempotent* element of M, that is $ee = e$ (it follows that $\mathcal{T}(v) = e$, too).

We refer to such nodes as *idempotent nodes*.

The following theorem is due to Simon (1990). For the bound of $3|M|$ see Colcombet (2011); Kufleitner (2008).

Theorem 10 (Simon factorization theorem) *For every sequence $s = [m_1, m_2, \ldots, m_n]$ of monoid elements from M, there is a factorization tree of height at most $3|M|$. This factorization tree can be computed in time poly$(|M|) \cdot n$.*

Factorization trees can also be applied for strings over an alphabet Σ (instead of a sequence of monoid elements), given a monoid morphism $h : \Sigma^* \to M$. A *Simon h-factorization tree* for a string $B = a_1 \cdots a_n \in \Sigma^*$ is a Simon factorization tree for the sequence $[h(a_1), \ldots, h(a_n)]$.

We now turn to the monoid that we use for building a factorization tree in the indexing phase of the learning algorithm. We actually define two monoids, where the second one is used for the factorization. Its elements consist of sets of elements of the first monoid that we define.

In the following, let $\varphi(x;\bar{y})$ be a unary MSO formula with parameter variables $y_1, \ldots, y_\ell$. The formula $\varphi(x\,;\bar{y})$ naturally defines a set of strings $\hat{L}(\varphi)$ over the alphabet $\hat{\Sigma} = \Sigma \times 2^{\{y_1,\ldots,y_\ell\}} \times \{?, 0, 1\}$. The first component of a string $\hat{B} \in \hat{\Sigma}^*$ defines a string B over Σ. The third component encodes a training set $T_{\hat{B}}$, where ? indicates that the position is not in $T_{\hat{B}}$, and $0, 1$ correspond to the classification of the position in $T_{\hat{B}}$. The second component is supposed to encode the parameter setting. We say that a string $\hat{B} \in \hat{\Sigma}^*$ *contains* y_i if there is a position v labeled by $(a, Y, b) \in \hat{\Sigma}$ such that $y_i \in Y$, and we say that that $\hat{B}$ contains y_i *exactly once* if there is exactly one such position. If $\hat{B}$ contains each y_i exactly once then the second component encodes a valid parameter setting $\bar{v} = (v_1, \ldots, v_\ell)$.

Then we let $\hat{L}(\varphi)$ be the set of all strings $\hat{B} \in \hat{\Sigma}^*$ such that $\hat{B}$ contains each y_i exactly once, yielding a parameter setting $\bar{v}$, and if $B \in \Sigma^*$ is the projection of $\hat{B}$ to the first component and $T_{\hat{B}}$ is the training set encoded by the third component, then $[\![\varphi(x\,;\bar{v})]\!]^B$ is consistent with $T_{\hat{B}}$.

It is not difficult to see that a finite automaton for the formula $\varphi(x\,;\bar{y})$ can be modified to obtain a finite automaton for the language $\hat{L}(\varphi)$, which means that $\hat{L}(\varphi)$ can also be accepted by a finite monoid. Let $\hat{M}$ be a finite monoid, $\hat{h} : \hat{\Sigma} \to \hat{M}$ be a monoid morphism, and $\hat{F} \subseteq \hat{M}$ such that $(\hat{M}, \hat{h}, \hat{F})$ accepts $\hat{L}(\varphi)$.

For every subset $K \in 2^{\{y_1,\ldots,y_\ell\}}$ of the parameters, we let

$$\hat{M}_K = \{\hat{h}(A) \mid A \in \hat{\Sigma}^* \text{ contains all } y_i \in K \text{ exactly once but does not contain any } y_i \notin K\},$$

and we let

$$\hat{M}_\perp = \{\hat{h}(A) \mid A \in \hat{\Sigma}^* \text{ contains some } y_i \text{ more than once}\}.$$

Without loss of generality we may assume that the sets $\hat{M}_K$ for $K \in 2^{\{y_1,\dots,y_\ell\}} \times \{\bot\}$ are mutually disjoint. It is easy to see this, the idea is that we can introduce copies m_K of all elements m and adjust the homomorphism $\hat{h}$ accordingly.

Then $\hat{M}$ is the disjoint union of the sets $\hat{M}_K$ for $K \in 2^{\{y_1,\dots,y_\ell\}} \cup \{\bot\}$. Observe that $\hat{F} \subseteq \hat{M}_{\{y_1,\dots,y_\ell\}}$ and that no substring of a string in $\hat{F}$ is contained in $\hat{M}_\bot$.

We now define a second monoid $\mathcal{M}$, which is used for the factorizations. The monoid $\hat{M}$ contains information about the parameters as encoded in the strings. In the learning setting, these parameters are unknown and we need to synthesize parameters consistent with the training set. We therefore introduce a monoid that contains information about all possible parameter settings that could be encoded in the strings.

For this purpose, let $\Gamma = \Sigma \times \{?, 0, 1\}$ be the alphabet without the component for the parameters. A string over Γ encodes a string over Σ together with a training set over B.

Let $f : \hat{\Sigma}^* \to \Gamma^*$ be the function that projects strings over $\hat{\Sigma}$ to the corresponding strings over Γ, removing the parameter component of $\hat{\Sigma}$. Based on this projection, each string A over Γ defines a set $h(A)$ of elements of $\hat{M}$ by

$$h(A) = \{m \in \hat{M} \mid \exists \hat{B} \in \hat{\Sigma}^* \text{ with } f(\hat{B}) = A \text{ and } \hat{h}(A) = m\}.$$

This defines a morphism $h : \Gamma^* \to \mathcal{M}$ using a new monoid structure $\mathcal{M} = \{S \mid S \subseteq \hat{M}\}$ with neutral element $1_\mathcal{M} = \{1_{\hat{M}}\}$, the set containing only the neutral element of $\hat{M}$, and multiplication $S_1 \cdot S_2 = \{m_1 \cdot m_2 \mid m_1 \in S_1, m_2 \in S_2\}$.

In the following, we denote by $B_T \in \Gamma^*$ the string that encodes $B \in \Sigma^*$ with training set T over B. In particular, $B_\emptyset$ denotes this string for the empty training set (so $B_\emptyset$ is B extended with ? at every position).

The next lemma states that we can compute parameters that are consistent with a given training set T over a string B, based on a Simon h-factorization of B_T.

Lemma 11 *Let $B \in \Sigma^*$ and let T be a φ-consistent training set over B. Given a Simon h-factorization tree $\mathcal{T}$ of B_T, one can compute in linear time in the height of $\mathcal{T}$ a set of parameters over B such that T is consistent with the parameters.*

Proof A procedure for computing a consistent parameter setting as claimed in Lemma 11 is shown as Algorithm 1. The idea and notations are explained below.

The variable v is used for nodes of $\mathcal{T}$, and m for monoid elements of $\hat{M}$. Recall that the nodes of $\mathcal{T}$ are labeled with elements from $\mathcal{M}$, which are sets of elements of $\hat{M}$.

The algorithm starts in the root of $\mathcal{T}$, and picks some accepting element m_{root} in the label of the root. Such an accepting element exists, since we assume that T is a φ-consistent training set. Thus, there is a parameter setting that is consistent with T. Adding this parameter setting to B_T yields a string $\hat{B}_T \in \hat{\Sigma}^*$. Then $m_{\text{root}} = \hat{h}(\hat{B}_T)$ is accepting and it is contained in the label of the root of $\mathcal{T}$.

The algorithm then descends down the tree to find parameter positions that generate the accepting element chosen at the root. It uses a stack because it has to descend on several paths (to find a position for each parameter).

Recall that $(\hat{M}, \hat{h}, \hat{F})$ accepts $\hat{L}(\varphi)$, that $\hat{M}$ is the disjoint union of the sets $\hat{M}_\bot$ and $\hat{M}_K$ for $K \in 2^{\{y_1,\dots,y_\ell\}}$, that the set $\hat{F}$ of accepting elements is contained in $\hat{M}_{\{y_1,\dots,y_\ell\}}$, and that no string in $\hat{F}$ has a substring in $\hat{M}_\bot$. As all elements $m \in \hat{M}$ the algorithm visits (and

Input: Simon factorization tree $\mathcal{T}$
Output: A consistent parameter setting $(y_1, \ldots, y_\ell)$

```
1  v ← root(T)
2  Pick m_root ∈ T(v) ∩ F̂                    // an accepting element in the label of v
3  push(m_root, v)
4  while not empty stack do
5  |  (m, v) ← pop()
6  |  if leaf(v) then
   |  |                                      // set the parameters traced to this leaf
7  |  |  Let K ⊆ {y_1, ..., y_ℓ} be such that m ∈ M̂_K
8  |  |  Set y_i ← v for each y_i ∈ K
9  |  else
   |  |                                                // descend further down the tree
10 |  |  Let v_1 be the first and v_2 be the last child of v
11 |  |  if v has two children then
12 |  |  |  Pick m_1 ∈ T(v_1), m_2 ∈ T(v_2) with m = m_1 m_2
13 |  |  end
14 |  |  if v has more than two children then
15 |  |  |  Let e be the unique idempotent element in M̂_∅ ∩ T(v)
16 |  |  |  Pick m_1, m_2 ∈ T(v) with m = m_1 e m_2
   |  |  |                                      // See Claim in the proof of Lemma 11
17 |  |  end
18 |  |  if m_1 ∉ M_∅ then push(m_1, v_1);
19 |  |  if m_2 ∉ M_∅ then push(m_2, v_2);
20 |  end
21 end
22 return ŷ = (y_1, ..., y_ℓ)
```

Algorithm 1: Computing a consistent parameter setting from a factorization tree

pushes to the stack in lines 3 and 18) are substrings of $m_{\text{root}} \in \hat{F}$, no such m is an element of $\hat{M}_\perp$.

In the main loop, Algorithm 1 traces monoid elements that are not in $\hat{M}_\emptyset$ further down the tree. Note that the elements $m \in \hat{M}_\emptyset$ correspond to words that do not contain a parameter.

The algorithm pops the next pair (m, v) with $m \in \hat{M}$ and v a node of $\mathcal{T}$ from the stack. If v is a leaf, then there is a set K such that $M \in \hat{M}_K$ and $K \neq \emptyset$ because elements from $M_\emptyset$ are never pushed onto the stack. The leaf v corresponds to a position in the string B. This position is the value for the parameters in K.

If v is an inner node, then m can be written as product of $\hat{M}$ elements in the labels at the children of v. If v has only two children v_1 and v_2, then the algorithm can simply pick elements m_1 and m_2 in the labels of v_1 and v_2 whose product is m. If v has more than two children, then it is an idempotent node. The choices made by the algorithm in this case are based on the following claim. Intuitively, this claim shows that for finding a consistent parameter setting, it is sufficient to consider the first and the last child of idempotent nodes.

Claim: Let $S \subset \hat{M}$ be the label of an idempotent node. Then S contains a unique idempotent element $e \in M_\emptyset$, and each element m of S can be written as a product $m = m_1 e m_2$ with $m_1, m_2 \in S$.

Proof: Let v be an idempotent node with label S. Since $\mathcal{T}$ is a factorization tree of B_T, the node v corresponds to a substring A of B_T, and $S = h(A)$. From the definition of $h(A)$ it follows that S contains exactly one element $e \in \hat{M}_\emptyset$ (the element for the empty parameter annotation of A). The product of two elements from $\hat{M}_\emptyset$ is also in $\hat{M}_\emptyset$. Since S is idempotent, it follows that $ee \in \hat{M}_\emptyset \cap S$, and thus $ee = e$.

We now show that each $m \in S$ can be written as $m = m_1 e m_2$ with $m_1, m_2 \in S$. Let $K \subseteq \{y_1, \ldots, y_\ell\}$ be such that $m \in \hat{M}_K$. Prove this by induction on the size of K.

If $K = \emptyset$, then $m = e = eee$, as shown above, and we let $m_1 = m_2 = e$. Otherwise, since S is idempotent, $m = m_1' m_2'$ with $m_1', m_2' \in S$. From the definition of $\hat{M}_K$ we obtain that $m_1' \in \hat{M}_{K_1}$, $m_2' \in \hat{M}_{K_2}$ with $K_1 \cup K_2 = K$ and $K_1 \cap K_2 = \emptyset$. If $K_1 = \emptyset$, then we choose $m_1 = e$ and $m_2 = m_2'$ and obtain $m_1 e m_2 = e e m_2' = e m_2' = m_1' m_2' = m$. The case $K_2 = \emptyset$ is analogous. If K_1 and K_2 are nonempty, then both are strict subsets of K. By induction $m_1' = m_1'' e m_2''$ for $m_1'', m_2'' \in S$, and hence $m = m_1'' e m_2'' m_2'$ and we can choose $m_1 = m_1''$ and $m_2 = m_2'' m_2'$. Since S is idempotent, $m_2 \in S$. This completes the proof of the claim. ⌟

For the correctness of the algorithm, one can prove that the parameters selected by the algorithm generate the accepting monoid element chosen at the root of $\mathcal{T}$ (by an induction on the height of the node v in the tree). This means that the choice of parameters is consistent with the training set. The running time is linear in the height of $\mathcal{T}$ because for each parameter there is at most one monoid element on the stack, which means that the algorithm follows at most ℓ paths in the tree.

This completes the proof of Lemma 11. ■

In order to apply Lemma 11 in our algorithm, we first have to compute a Simon h-factorization tree $\mathcal{T}$ of B_T for a given training set T. We can do this starting from a factorization of $B_\emptyset$, as stated in the following lemma.

Lemma 12 *Let $B \in \Sigma^*$ and let T be a training set over B. From a Simon h-factorization tree $\mathcal{T}_B$ of $B_\emptyset$, one can compute a Simon h-factorization tree $\mathcal{T}$ of B_T in time $\mathcal{O}(height(\mathcal{T}_B) \cdot |T|)$. The height of $\mathcal{T}$ is in $\mathcal{O}(height(\mathcal{T}_B))$.*

Proof The rough idea is as follows: We cut B into factors at the positions occurring in T. Each position in T becomes one factor (consisting of a single position), and the other factors are the substrings between these positions. Then we insert the modified monoid elements at the positions from T, by changing the ? into the classification of the position in T. For the longer substrings (that are not touched by the training set) we compute a factorization tree, which can easily be done based on the one for the whole string. We obtain one monoid element for each factor (at the root of the trees for the longer strings). For this new sequence of monoid elements we apply Theorem 10, obtaining a Simon factorization, which can be combined with the existing factorization trees for the substrings to obtain a factorization tree for B_T.

More formally, let $u_1, \ldots, u_t \in \{1, \ldots, |B|\}$ be the positions of B occurring in the training set T in ascending order. Let $u_0 = 0$ and $u_{t+1} = |B|+1$ to simplify the following definitions.

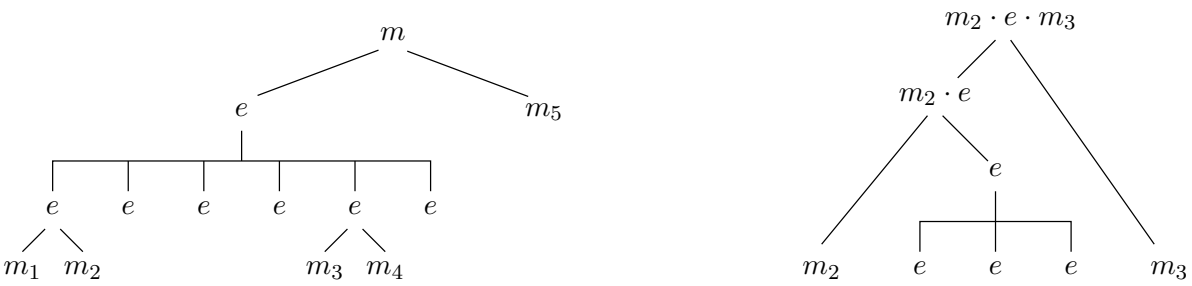

Figure 2: Simon factorization tree

For $i \in \{0, \ldots, t\}$ define B_i to be the substring of $B_\emptyset$ from position $u_i + 1$ to $u_{i+1} - 1$, and for $i \in \{1, \ldots, t\}$ let $\gamma_i = (a, c) \in \Gamma$ be the letter a of B at position u_i, and c the classification of u_i in T. Then $B_0\gamma_1 B_1\gamma_2 \cdots B_{t-1}\gamma_t B_t = B_T$

For the substrings B_i we can compute a Simon h-factorization tree $\mathcal{T}_i$ from $\mathcal{T}_B$. This is done in a similar fashion as described in Bojańczyk (2012) for evaluating queries for substrings on a Simon factorization tree. The idea is illustrated in Figure 2, which shows a Simon factorization tree for the sequence $[m_1, m_2, e, e, e, m_3, m_4, e, m_5]$ on the left-hand side of the figure. The right-hand side of the figure shows a Simon factorization tree for the sub-sequence $[m_2, e, e, e, m_3]$. Basically one has to trace the paths towards the root from the left-most and right-most leaf nodes of the tree for the sub-sequence. Along these paths one has to update some labels, delete some nodes, and inserting some new nodes in order to maintain the structure of a Simon factorization tree. In Figure 2, the nodes labeled with a product of elements are the ones that have been inserted. One observes that the insertion of new nodes might increase the height of the tree, but the height can at most double.

Let m_i be the monoid element obtained at the root of the tree $\mathcal{T}_i$. We obtain the sequence

$$[m_0, h(\gamma_1), m_1, h(\gamma_2), \ldots, m_{t-1}, h(\gamma_t), m_t]$$

of monoid elements alternating between the roots of the trees $\mathcal{T}_i$ and the elements corresponding to the modified positions in B. Then we compute a Simon h-factorization tree $\mathcal{T}'$ for this sequence $[m_0, h(\gamma_1), m_1, h(\gamma_2), \ldots, m_{t-1}, h(\gamma_t), m_t]$ according to Theorem 10. We can now plug in the trees $\mathcal{T}_i$ at the corresponding leafs of $\mathcal{T}'$ for m_i. This results in a Simon h-factorization tree for B_T.

The complexity claims follow from the complexities in Theorem 10 and the fact that the height of the trees $\mathcal{T}_i$ is linear in the height of $\mathcal{T}_B$. ■

Combining Theorem 10 with Lemma 11 and 12, we can build a learning algorithm as claimed in Theorem 9.

- *Indexing Phase:* For a string B, compute a Simon h-factorization $\mathcal{T}_B$ of $B_\emptyset$ according to Theorem 10.
- *Learning Phase:* For a given Training set T, compute a Simon h-factorization tree $\mathcal{T}$ of B_T according to Lemma 12, and then compute a consistent set of parameters according to Lemma 11.

The claimed complexities follow from the ones in Theorem 10 and Lemma 11 and 12. This finishes the proof of Theorem 9.

Formulas of higher arity. The methods developed in this section for unary MSO formulas can be adapted to some extent to MSO formulas of higher arity. However, the learning time of this adapted algorithm is not linear in the size $|T|$ anymore.

Theorem 13 *There is a consistent parameter learner for MSO formulas with indexing time $O(|B|)$ for a string B and learning time $O((k|T|)^\ell)$ for a training set T over B, and ℓ the number of parameters.*

The main difference is that it is not possible to encode a complete training set for examples of higher arity by an annotation of the string B. For this reason, one has to do an exhaustive search over the possible parameter positions relative to the positions appearing in the training set. Again based on a factorization tree, one can check for each such relative positioning if concrete parameters with these relative positions exist that are consistent with the training set. If they exist, one can synthesize them with the same idea as for Algorithm 1. We save the details for a full version of this article.

6. Conclusion

We study the learnability results for MSO-definable hypotheses over string data. The key question we ask is whether learning is possible in time independent of (or at least sublinear in) the size of the background string. We prove that this is only possible if we allow to build an index of the string first, in time linear in the size of the string.

It is an interesting open question whether our results can be extended to tree-structured data (such as XML-documents). Note that there is no direct generalization of the factorization forests for trees.

References

A. Abouzied, D. Angluin, C.H. Papadimitriou, J.M. Hellerstein, and A. Silberschatz. Learning and verifying quantified boolean queries by example. In R. Hull and W. Fan, editors, *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 49–60, 2013.

Dana Angluin. On the complexity of minimum inference of regular sets. *Information and Control*, 39(3):337–350, 1978. doi: 10.1016/S0019-9958(78)90683-6. URL https://doi.org/10.1016/S0019-9958(78)90683-6.

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

Dana Angluin. Negative results for equivalence queries. *Machine Learning*, 5:121–150, 1990. doi: 10.1007/BF00116034. URL https://doi.org/10.1007/BF00116034.

Mikołaj Bojańczyk. Algorithms for regular languages that use algebra. *SIGMOD Record*, 41(2):5–14, 2012.

A. Bonifati, R. Ciucanu, and S. Staworko. Learning join queries from user examples. *ACM Trans. Database Syst.*, 40(4):24:1–24:38, 2016.

W.W. Cohen and C.D. Page. Polynomial learnability and inductive logic programming: Methods and results. *New generation Computing*, 13:369–404, 1995.

Thomas Colcombet. Green's relations and their use in automata theory. In *Language and Automata Theory and Applications - 5th International Conference, LATA 2011, Tarragona, Spain, May 26-31, 2011. Proceedings*, volume 6638 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2011.

P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 499–512, 2016.

E. Mark Gold. Complexity of automaton identification from given data. *Information and Control*, 37(3):302–320, 1978. doi: 10.1016/S0019-9958(78)90562-4. URL https://doi.org/10.1016/S0019-9958(78)90562-4.

M. Grohe and M. Ritzert. Learning first-order definable concepts over structures of small degree. *ArXiv (CoRR)*, arXiv:1701.05487 [cs.LG], 2017. To appear in *Proceedings of the 32nd Annual ACM/IEEE Symposium on Logic in Computer Science*, 2017.

Martin Grohe and Gy Turán. Learnability and definability in trees and similar structures. *Theory of Computing Systems*, 37(1):193–220, 2004.

C. Jordan and L. Kaiser. Machine learning with guarantees using descriptive complexity and smt solvers. *ArXiv (CoRR)*, arXiv:1609.02664 [cs.LG], 2016.

Michael J. Kearns and Leslie G. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. *J. ACM*, 41(1):67–95, 1994. doi: 10.1145/174644.174647. URL http://doi.acm.org/10.1145/174644.174647.

Michael J. Kearns and Umesh V. Vazirani. *An introduction to computational learning theory*. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-11193-4.

J.-U. Kietz and S. Dzeroski. Inductive logic programming and learnability. *SIGART Bulletin*, 5(1):22–32, 1994.

Manfred Kufleitner. The height of factorization forests. In *MFCS 2008*, volume 5162 of *LNCS*, pages 443–454. Springer, 2008.

C. Löding, P. Madhusudan, and D. Neider. Abstract learning frameworks for synthesis. In M. Chechik and J.-F. Raskin, editors, *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 9636 of *Lecture Notes in Computer Science*, pages 167–185. Springer Verlag, 2016.

S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–318, 1991.

S.H. Muggleton, editor. *Inductive Logic Programming*. Academic Press, 1992.

S.H. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19-20:629–679, 1994.

José Oncina and Pedro García. Identifying regular languages in polynomial time. In *Proceedings of the International Workshop on Structural and Syntactic Pattern Recognition*, volume 5 of *Machine Perception and Artificial Intelligence*, pages 99—108. World Scientific, 1992.

Leonard Pitt and Manfred K. Warmuth. The minimum consistent DFA problem cannot be approximated within any polynomial. *J. ACM*, 40(1):95–142, 1993. doi: 10.1145/138027.138042. URL http://doi.acm.org/10.1145/138027.138042.

Michael O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125, April 1959.

Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. In *Machine Learning: From Theory to Applications*, volume 661 of *Lecture Notes in Computer Science*, pages 51–73. Springer, 1993.

S. Shalev-Shwartz and S. Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, 2014.

Imre Simon. Factorization forests of finite height. *Theor. Comput. Sci.*, 72(1):65–94, 1990. doi: 10.1016/0304-3975(90)90047-L. URL http://dx.doi.org/10.1016/0304-3975(90)90047-L.

W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 389–456. Springer-Verlag, 1997.

M.Y. Vardi. The complexity of relational query languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pages 137–146, 1982.