Classes of bounded functions that are semantically equivalent to Turing-machine are PAC learnable

Kevin H. Xu

Froglingo Development Association
Bridgewater, New Jersey, USA

Abstract. A contemporary programming language is a top-down approach in the sense that we know exactly what a function is to be constructed and we construct the exact function. Learning is a bottom-up approach in the sense that we don't know how to exactly program a targeted function but we can program an algorithm that automatically constructs another function that converges to the target by accepting sample data. Learning can be done not only through statistical methods but also through symbolic computing. While statistical learning methods have their unique positions in many applications including pattern recognition, symbolic approaches toward learning persist in keeping the realizability assumption for convergence. In addition to many known symbolic Probably Approximately Correct (PAC) learnables such as conjunctions of Boolean literals and rectangle learning games, there are other symbolic computing systems that are PAC learnable as well. In this paper, we show a class of functions that is semantically equivalent to Turing machine is PAC learnable. This learnability is realized through the Enterprise-Participant (EP) data model, a database language representing such a class of functions, called bounded functions as they have a finite co-domain while an infinite domain. An EP database is mathematically capable of inventorying all the properties of partial recursive functions with the hypothesis of infinite space and time.

Keywords: Learnability · computability · database · bounded function · lambda calculus

1 Introduction

Statistical algorithms for machine learning, such as latent semantic analysis, Laplacian eigenmaps, and neural networks with backpropagation, are powerful and dominate contemporary machine learning research and practice. Symbolic computation is also a way of machine learning as long as it generates a program that is able to produce more meaningful information that is not originally provided as sample inputs. Conjunctions of boolean literals and a rectangle learning game are well known examples, [10]. We are interested in symbolic-base learning because symbolic approaches keeps the realizability assumptions, i.e., always receiving positive sample data, such that targeted programs can be converged to a level we anticipate with a foreseeable sample size.

In this paper, we introduce classes of bounded functions that are PAC learnable because an algorithm exists to construct a database as the program that correctly labels an arbitrary number of fresh inputs beyond a finite set of labeled sample inputs. This learnability is supported by the Enterprise-Participant (EP) data model, a database language system, where an EP database is the constructed program and EP's reduction system supports predictions of the PAC learnability, precisely symbolic reasoning. As a special case of the PAC learnability, for example, an algorithm exists to construct an EP database by taking a finite set of randomly selected paths as samples: $v_2\ v_1$ and $v_1\ v_2\ v_3$. The two paths represent a directed cyclic graph with edges: v_1 to v_2 , v_2 to v_1, and v_2 to v_3. The constructed database is: $D = \{v_1\ v_2 := v_2;\ v_2\ v_1 := v_1;\ v_2\ v_3 := v_3\}$, with which EP supports arbitrary number of queries simulating one's walks along the cyclic graph, i.e., reductions: $v_1\ v_2\ v_3 \to_D v_3$, $v_1\ v_2\ v_1 \to_D v_1$, $v_2\ v_1\ v_2\ ...\ v_1 \to_D v_1$, ... (Note that a sequence of nodes that do not form a path would be reduced to a special value *null*, e.g., $v_3\ v_2 \to_D$ *null*.)

While weights and biases adjustment is the fundamental process in statistical learning that contributes to prediction, symbolic reasoning in EP is a synonym of the prediction of the statistical learning. Though the methods coming to prediction are different, the prediction from both symbolic and statistical learning approaches has the same effect: targeted functions can be approximated by constructed programs and eventually converged with the evolving programs without explicit programming.

A flip side of the synonymousness is the precision of predictions. Statistical learning may not have all sample data labeled correctly. As a result, a constructed program may not converge to its target program. While such agnostic learning is necessary for certain applications, symbolic learning preserves the full meaning of the PAC learnability, i.e., sample size amounts to any designed level of precision.

We say that a language, e.g., a programming language, is Turing-complete or Turing-equivalent, if the language has a finite representation for a class of partial recursive functions. To measure the significance of an approximation to a Turing-equivalent language that is generated by a partial computation in the theory of computability, we also say that the union of the approximations produced by a sequence of partial computations on the given Turing-equivalent language with infinite computation steps $1, 2, ..., s, ...$, where $s \in \mathbf{N}$, is semantically equivalent to the given language, because the union of all the approximations is equal to the entire set of the properties the given language has. Because each of the concept classes in a sequence is (efficiently) PAC learnable, and because the union of all the concept classes is semantically equivalent to the Turing machine, we say that the entire sequence of the classes are (efficiently) PAC learnable in the sense of the hypothesis that we have infinite time and space. This conclusion says nothing else but the EP data model, or the classes of bounded functions, sets up a mathematical bound for program constructions without programming, at least as far as symbolic learning approach is concerned.

The EP data model was first developed to unify programming languages with database management [25], [24]. It was later found in [23] that the EP data model is rooted at the lambda calculus: an EP database can be syntactically converted from an approximation to the lambda calculus, further can be expressed in an extended lambda calculus, and is interpreted as a bounded function. To avoid unnecessary burdens to many readers who are not familiar with the lambda calculus, we first describe the Enterprise-Participant (EP) data model, the reduction rules, and the corresponding bounded functions in Section 2 and 3, which are developed independently from approximations to the lambda calculus. In Section 4, we introduce an enumeration of all possible databases that can be constructed based on the EP database definition given in Section 2. This establishes a foundation for our discussion on PAC learnability in Section 5 because a kind of concept classes (of bounded functions) is formed from the enumerated databases.

Because EP is type free, we note that the size of a database can grow exponentially in the size of allowed basic elements of the database. This determines that the corresponding classes of bounded functions cannot be efficiently PAC learnable when they are proved PAC learnable in Section 5. To show that a database size doesn't grow exponentially in practice, we describe another kind of bounded function classes with a polynomial logarithm of the cardinality in the computation steps of a partial computation to the lambda calculus and therefore that the classes are efficiently PAC learnable. The result is presented in Section 5 while the detail is captured in Appendix E. To facilitate the discussion in Appendix E, we reiterate the material from [23] regarding the partial computation process to the lambda calculus and the syntactical conversion process from an approximation to an EP database in Appendix B and C. While the semantic equivalence of EP with the lambda calculus has been formally proved in [23], Appendix D provides a supplementary material to formally prove that a class of bounded functions is semantically equivalent to the lambda calculus as well.

In Appendix F, we give an alternative view in DS-dimensions on the conclusions we made in Section 5 and point out that the DS dimensions for a class of bounded functions is the number of assignments in one of the largest databases in the class.

In Appendix A, we give supplementary materials for Sections 2 to 5, including proofs for the theories presented in the sections.

2 EP databases

The Enterprise-Participant (EP) data model is a language system and equivalently a data structure with which an EP database can be constructed. The idea behind EP is that we treat all objects to be represented as functions. Given a function f that produces a value m when it is applied to an argument n, denoted as $f\ (n) = m$, let's think of an exercise in which we inventory the properties of f in a database. We can rewrite $f\ (n) = m$ as $f\ n := m$, reading it as: applying f to n is assigned a value m. The set $\{f\ n := m\}$, called a database, is an approximation of f. When we apply f to an additional argument n', we would obtain a better approximation $\{f\ n := m, f\ n' := m'\}$ where $f\ (n') = m'$. In addition, m could be another function such that $m\ (p) = q$ for a given input p. So we can exhibit more properties of f with the accumulated approximation $\{f\ n := m, f\ n' := m', m\ p := q\}$ or equivalently $\{f\ n\ p := q, f\ n' := m'\}$. From the database $\{f\ n := m, f\ n' := m', m\ p := q\}$, we can derive: $(f\ (n))\ (p) = q$.

Definition 2.1 The EP data model is described as a language system $(\mathbf{F}, \cdot, (,), \mathbf{E}, :=, D)$, where

1. $\mathbf{F}$ is a set of identifiers (function names)
2. $\cdot$ is a binary operation that produces a set $\mathbf{E}$ such that
   $m \in \mathbf{F} \implies m \in \mathbf{E}$
   $m, n \in \mathbf{E} \implies (m \cdot n) \in \mathbf{E}$
   Here we simply write $(m \cdot n)$ as $(m\ n)$ and further $m\ n$ when $(m \cdot n)$ is implied [1], where m, n, and $m\ n$ are called a function, an argument, and the corresponding application. For a $x \in \mathbf{E}$, we call x a term. Given an application term $m\ n$, m and n are called proper subterms of $m\ n$, and $m\ n$ is also called a subterm of itself.
3. $:=$ is the Cartesian product $\mathbf{E} \times \mathbf{E}$, i.e., $:= \equiv \mathbf{E} \times \mathbf{E}$. When a pair $(p, q) \in :=$, we denote it as $p := q$, which is called an assignment, where p and q are the assignee and assigner respectively.
4. D, called a database, is a finite set of terms and a finite set of assignments, i.e., $D \subset (\mathbf{E} \cup :=)$, such that for each assignment $p := q \in D$, where $p, q \in \mathbf{E}$, the following constraints are met:
   (a) p has only one assigner, i.e.,
   $$p := q \text{ and } p := q' \in D \implies q \equiv q'$$
   (b) A proper subterm of p cannot be an assignee, i.e.,
   $$p := q \in D \implies \forall x \in SUB^+(p)\ [\forall m \in \mathbf{E}\ [x := m \notin D]]$$
   (c) q can not be an assignee, i.e.,
   $$p := q \in D \implies \forall a \in \mathbf{E}\ [q := a \notin D]$$

Identifiers are the most basic building blocks in EP. Like in programming languages, we can choose alphanumeric tokens as identifiers, such as *abc*123, _*abc*, and more commonly we take words from a natural language vocabulary as identifiers, such as *hello*, *John*, *sport*, *law*, and *person*.

A term is either an identifier $x \in \mathbf{F}$ or an application $x\ y \in \mathbf{E}$ where $x, y \in \mathbf{E}$, such as $x\ x$, $(a\ b\ c)\ (d\ e\ a\ (d\ t\ a))$ are legitimate terms where $x, a, b, c, d, e, t \in \mathbf{F}$.

Given a term, e.g., $m_0\ m_1, \ldots, m_i$ for an $i \in N$, we call all the leftmost subterms of the term, i.e., $m_0, m_0\ m_1, \ldots, m_0\ m_1 \ldots m_i$ a leftmost subterm, denoted as *lms*. Given a term t, we use $|t|$ to denote the size of the term, e.g., $|m_0\ m_1\ \ldots\ m_i| = i + 1$, and $LMS(t)$ to denote the set of all *lms*s in t. (Then we have $t \in LMS(t)$. If $m\ n \in LMS(t)$, so is m.) We further use $LMS^+(t)$ to denotes all the proper *lms*s in t, i.e., $LMS^+(t) = LMS(t) \setminus t$. We further use $SUB(p)$ to denote all the subterms of a term p, i.e., given $p \equiv m\ n$, then $m, n, m\ n \in SUB(p)$. We use $SUB^+(p)$ to denote all the proper subterms of p, i.e., $SUB^+(p) = SUB(p) \backslash \{p\}$.

See Appendix A.2 for more information including sample EP terms and databases.

3 EP database reductions and bounded functions

We are ready to introduce a reduction system over $\mathbf{E}$. First, we identify a special identifier $null \in \mathbf{F}$ that has the default reduction rule: $null\ m \rightarrow_D null$ for any $m \in \mathbf{E}$. Because we explicitly single out the special identifier *null* from $\mathbf{F}$, we further restrict that a *lms* of an assignee cannot be *null* in a database. Before providing the full set of reductions rules, let's first define the notation of EP normal form.

Definition 3.1 Given a database D, a term $n \in \mathbf{E}$ is an EP normal form (or normal form in brief) if and only if

1. n is *null*, i.e, $n \equiv null$; or

[1] When a combination of two terms $m\ n$ is given without surrounding parentheses, we consider the term is parsed by the preference of left association and therefore $(m\ n)$ is implied. For example, $a\ b\ c$ always implies $((a\ b)\ c)$. If one needs to express $(a\ (b\ c))$, then it has to be written explicitly like $a\ (b\ c)$.

2. n is a term in D and not an assignee, i.e., $n \in D$ and $\forall b \in \mathbf{E}\ [n := b \notin D]$.

We use $NF(D)$ to denote the set of all the normal forms under a database D.

Most terms in $\mathbf{E}$ are not normal forms given a database D, For example, $x\ x$ and $x\ x\ x$ are not normal forms in the sample database $\{x\ x := x\}$. We now define a set of rules to reduce an arbitrary term to a normal form.

Definition 3.2 Given a database D, we have one-step reduction rules, denoted as $\Rightarrow$:

1. An assignee is reduced to the assigner, i.e., $a := b \in D \implies a \Rightarrow b$
2. An identifier not in the database is reduced to *null*, i.e., $a \in \mathbf{F}, a \notin D \implies a \Rightarrow null$
3. If a and b are normal forms and $a\ b \notin D$, then $a\ b$ is reduced to *null*, i.e., $a, b \in NF(D), a\ b \notin D \implies a\ b \Rightarrow null$
4. $a \Rightarrow a', b \Rightarrow b' \implies a\ b \Rightarrow a'\ b'$

Definition 3.3 Let $a \Rightarrow a_0, a_0 \Rightarrow a_1, \ldots, a_{n-1} \Rightarrow a_n$ for a number $n \in \mathbf{N}$. We say that a is effectively, i.e., in finite steps, reduced to a_n, denoted as $a \rightarrow_D a_n$.

Definition 3.4 A term a has a normal form b if b is in normal form and $a \rightarrow_D b$.

See Appendix A.3 for a few sample reductions against the databases provided in Appendix A.2.

Any term $m \in \mathbf{E}$ has one and only one normal form and the reduction system is strongly normalizing, i.e., there is another term $n \in NF(D)$ such that $m \rightarrow_D n$ (Theorem 4.5 in [23]).

The set of all the normal forms $NF(D)$ is finite, i.e., $|NF(D)| \leq s$ for a given $s \in \mathbf{N}$, e.g., s could be the number of partial computation steps from which D is transformed (Lemma 4.6 in [23]).

There exists a function $Y(D) : \mathbf{E} \rightarrow_D NF(D)$, where $Y(D) = \{(m, n) \mid m \in \mathbf{E}, n \in NF(D), \text{ and } m \rightarrow_D n\}$, and $Y(D)$ is <u>bounded</u> because it has an infinite domain while only having a finite co-domain (Theorem 4.7 and Theorem 4.8 in [23]). A function $f : X \rightarrow Y$ has a <u>finite support</u> if and only if X is an arbitrary set of objects and Y is a finite set of objects, and there exists a finite set $A \subset X$ and a unique member $a \in Y$ such that

$$\begin{aligned} f(x) &= b, \text{ where } b \in Y \text{ and } b \neq a, && \text{if } x \in A \\ &= a && \text{if } x \in X \backslash A \end{aligned}$$

A function $f : X \rightarrow Y$ is bounded if and only if X is an arbitrary set of objects and Y is a finite set of objects. (Such a bounded function is always recursive, i.e., the computation on $f(x)$ terminates and $f(x) \in Y$ for any $x \in X$.) In this article, we simply call a function <u>finite</u> if it has a finite support. A finite function is bounded, but a bounded function may not be finite.

By saying a function being bounded, we mean that under a given database D, potentially an infinite number of terms $m \in \mathbf{E}$ are meaningful, i.e., reducible to a finite set of normal forms $NF(D)$ (excluding *null*). The ability of mapping infinite objects to finite objects is both the symptom and the pre-condition of the learnability, i.e., one object in the co-domain is represented by multiple objects in the domain, or saying differently one object in the co-domain is derivable from others. Denoting $m\ \bar{n}$ as the term $m\ n_1\ \ldots\ n_k \in \mathbf{E}$ for $k \geq 0$ and denoting $|\overline{n}|$ as the size k, we show that if there exists a sequence of assignments $a_0\ \overline{n_0} := a_1, a_1\ \overline{n_1} := a_2, \ldots, a_{i-1}\ \overline{n_{i-1}} := a_i \in D$ for some $i \geq 1$ such that $a_i \in LMS^{+}(a_0\ \bar{n_0})$, then $Y(D)$ is not finite but bounded. For example, $Y(D) = \{(x, x), (x\ x, x), (x\ x\ x, x) \ldots\}$ is not finite but bounded for the database $D = \{x\ x := x\}$.

Even if $Y(D)$ is finite for a given D, $Y(D)$ may provide derivable information beyond what are defined in D. For example, the database $D = \{a\ b := c; c\ d := e\}$ allows the reduction (derivation): $a\ b\ d \rightarrow_D e$, which is not defined in D. Lastly, an EP database may not have any derivable information, e.g., $D = \{a\ b := c; e\ e := f\}$. Such databases without derivable information have nothing to do with learnability.

When a term m is reduced to *null* under a database D, i.e., $m \rightarrow_D null$ or $Y(D)(m) = null$, m or $Y(D)(m)$ is in a correspondence to "undefined" or "I don't know" in partial recursive functions. Given a partial recursive function, e.g., $f : X \rightarrow Y$, we say an instance $f(x)$ is undefined if $x \notin X$. When $f(x)$ doesn't halt, or may halt eventually but doesn't terminate within a given number of computation steps, we say "I don't know" on $f(x)$ regarding its value, [23].

It has been shown in [23] that the EP data model is semantically equivalent to the lambda calculus, so is a class of bounded functions that are represented by a sequence of EP databases (Appendix D).

4 Concept classes dimension growing exponentially

In this section, we develop classes of databases (and bounded functions) that grows exponentially in its dimension, i.e., the logarithm of their cardinalities [13]. Although such an exponentially growing database cannot be practically constructed in general and the corresponding class of bounded functions cannot be efficiently PAC learnable, it provides the upper bound cardinality of various classes of bounded functions that can be practically constructed in an EP database.

During the construction, we consider two parameters: identifiers from $\mathbf{F}\backslash\{null\}$ and the size of a term, i.e., the number of identifiers $|t|$ in a term $t \in \mathbf{E}$. The goal is to construct all possible databases from a given number of identifiers and a maximum size of a term allowed to be in the databases. We set an upper bound $k \in \mathbf{N}$ for the number of identifiers and at the same time for the number of identifiers in a term that a database is allowed to contain. Further, we restrict a term without parenthesis, i.e., given a term $m_0, ..., m_j$, each m_i is an identifier, where $0 < j \leq k$ and $0 \leq i \leq j$. Provided that the databases with a bound k are constructed, we can continue to find additional databases with the bound $k+1$, where the bounds of the identifiers and the size of a term in a database are increased by 1. We use a pair $[F, k]$ (and sometimes denoted as $[F_k, k]$) to denote the set of the databases generated with a k such that $|F| = k$ and $|t| \leq k$ for all terms $t \in \mathbf{E}$ and $t \in [F, k]$, where $F \subset \mathbf{F}\backslash\{null\}, k \in \mathbf{N}$. For a given $[F, k]$, we use $[[F, k]]$ to denote the corresponding bounded function sets, i.e., $[[F, k]] = \{Y(D) \mid D \in [F, k]\}$. We use $|[F, k]|$ and $|[[F, k]]|$ to denote the cardinalities of the database set and the function sets, and actually we have $|[F, k]| = |[[F, k]]|$.

A database set $[F, k]$ is a simplified version of databases that can grow based on its definition given in Definition 2.1.4. Taking $[\{a, b, c\}, 3]$ for an example, we only consider a term like $a\ b\ c$, but not term like $a\ (b\ c)$. A term like $a\ null\ c$ is not considered either until in Appendix F. The simplified version has a smaller cardinality, but it is sufficient to show its exponential growth to support our discussion in the DS dimensions later.

The class of zero-identifier databases $[\{\}, 0]$ has only the empty database $\{\}$. The class of one-identifier databases, e.g., $[\{a\}, 1]$, consists of two databases without an assignment, e.g., $\{\{\}, \{a\}\}$. The size of the databases is 2, i.e, $|[\{a\}, 1]| = 2$. The class of two-identifier databases $[\{a, b\}, 2]$, where $a, b \not\equiv null$, is constructed as:

1. The allowed identifiers are a, b.
2. The allowed terms with 2 as the maximum size are $a, b, a\ a, b\ b, a\ b, b\ a$.
3. There are a total of 8 allowed assignments, such as $a\ a := a, a\ a := b$, and $a\ b := a$.
4. There are a total of 81 allowed databases, including one empty database , 8 one-assignment databases, 24 two-assignment databases, 32 three-assignment databases, and 16 four-assignment databases. Therefore, the cardinality of the corresponding concept class of the bounded functions is 81 as well.

See Appendix A.4 for a listing of all the databases in $[\{a, b\}, 2]$.

Certainly, we have $[F_k, k] \subset [F_{k+1}, k+1]$, where F_{k+1} has one more identifier on the top of F_k:
Proposition 4.1 (Theorem 7.5 of [23]) $[F_k, k] \subset [F_{k+1}, k+1]$.

In the rest of the paper, we will heavily rely on Theorem 2.2 of [10] and Theorem 5 of [13] to relate the PAC learnability with the cardinalities of various concept classes, i.e., if a concept class $\mathcal{C}$ is efficiently PAC learnable, then the logarithm of the cardinality of a concept class, called the dimension of $\mathcal{C}$ in [13] and denoted as $dim(\mathcal{C}) = log\ |\mathcal{C}|$, must be polynomial.
Proposition 4.2 The size of the largest databases in $[F, k]$ and $dim([[F, k]])$ are exponential in k. (See the proof in Appendix A.4.)

Because the growing speed of the size of a database is too fast to be constructed in practice and because $dim([[F, k]])$ is exponential, the class $[[F, k]]$ for $k \in \mathbf{N}$ is inefficient even it is PAC learnable.
Theorem 4.3 The class $[[F, k]]$ for $k \in \mathbf{N}$ is inefficiently PAC learnable.
Proof It is PAC learnable (shown in Section 5). The learnability is not efficient according to Theorem 2.2 of [10] and Theorem 5 of [13]. □

Fortunately, there are concept classes that have their cardinalities growing polynomially. One kind of such classes are generated with constraints from the partial computation process for generating approximations to the lambda calculus. Such classes are denoted as $\mathbb{Y}_s$, will be discussed in Appendix E, and the result Theorem E.5 is presented in Section 5. Even without the constraints from the partial computation, there are classes that grow polynomially. A graph is an example:

Theorem 4.4 a class of graphs $\mathbf{G}_n$ for a $n \in \mathbf{N}$, precisely the corresponding class of bounded functions, is efficiently PAC learnable. (See the proof in Appendix A.4.)

5 A class of bounded functions is PAC learnable

Regardless of the growing speeds of the concept classes of bounded functions in dimension, we show in this section a class of bounded functions is PAC learnable using the definition of the PAC learnability, [10], [17]. It is essentially determined by the size of the largest databases in a class of databases, where the size of a database determines the cardinality of the class of bounded functions.

Instead of denoting a class as a specific one such as $[[F, k]]$ as given in Section 4 and $\mathbb{Y}_s$ as to be given in Appendix E, we use $\mathfrak{Y}_k$ to denote a general bounded function class over an instance space $\mathfrak{X}_k$ and $\mathfrak{D}_k$ to denote the corresponding database class for any $k \in \mathbf{N}$ in the following discussion of this section. We will also use these notations in Appendix F.

5.1 A learning algorithm

When we say that $\mathfrak{Y}_k$ for a given $k \in \mathbf{N}$ is efficiently (or inefficiently) PAC learnable, we are saying for any unknown $D_c \in \mathfrak{D}_k$ (and therefore $Y(D_c) \in \mathfrak{Y}_k$), there exists an algorithm $\mathcal{A}$ such that:

1. $Y(D_c)$ is the target function (concept) for $\mathcal{A}$ to accumulate assignments into a database, denoting the accumulation process as a sequence of databases $D_0, D_1, ..., D_m$ where $D_i \in \mathfrak{D}_{k+1}$ for all $0 \le i \le m$, such that $Y(D_0), Y(D_1), ..., Y(D_m)$ approximates and converges to $Y(D_c)$ when it receives a sequence of sample pairs $(a_0, Y(D_c)(a_0))$, $(a_1, Y(D_c)(a_1))$, ..., $(a_m, Y(D_c)(a_m))$ that are selected from $Y(D_c)$, with a arbitrary probability distribution $\mathcal{P}$.
2. The approximation and convergence mentioned above are precisely defined as: for every distribution $\mathcal{P}$ on $\mathfrak{X}_s$, and for all $0 < \epsilon < 1/2$ and $0 < \delta < 1/2$, as $\mathcal{A}$ is given access to $EX(Y(D_c), \mathcal{P})$ that produces the sequence $(a_0, Y(D_c)(a_0))$, $(a_1, Y(D_c)(a_1))$, ..., $(a_m, Y(D_c)(a_m))$ and is given inputs ϵ and δ, then with probability at least $1 - \delta$, $\mathcal{A}$ outputs $Y(D_m)$, called a hypothesis, satisfying

   $$error(Y(D_m)) = \mathbf{Pr}_{a \in \mathcal{P}}[Y(D_c)(a) \neq Y(D_m)(a)] \le \epsilon$$

   This probability is taken over the random examples drawn by calls to $EX(Y(D_c), \mathcal{P})$, and any internal randomization of $\mathcal{A}$.
3. $\mathcal{A}$ takes time polynomial (or exponential) in k, $|D_c|$, $1/\epsilon$, and $1/\delta$ when learning the target concept $Y(D_c) \in \mathfrak{Y}_s$.
4. $Y(D_c)$ takes time polynomial (or exponential) in k and $|D_c|$ in calculating $Y(D_c)(a)$ for any $a \in \mathbf{E}$.
5. When $k = \infty$, $\mathfrak{Y}$ is efficiently PAC learnable in a sense of the hypothesis that $\mathcal{A}$ is given infinite time and space to learn.

We first define the learning algorithm $\mathcal{A}$ by assuming a number m exists such that after m calls to $EX(Y(D_c), \mathcal{P})$, we have at least $1 - \delta$ confidence to ensure that the $error(Y(D_m)) \le \epsilon$ for given ϵ and δ. We will show what m is later.
Definition 5.1 The learning algorithm $\mathcal{A}$: we first initiate an empty database $D = \{\}$. While the number of the calls to $EX(Y(D_c), \mathcal{P})$ has not reached m, call $EX(Y(D_c), \mathcal{P})$ to get a pair (a, b) and search D to find if a and b have already been served as the assignee and the assigner of an assignment in D.

1. if (a, b) has been in D already, do nothing.
2. if there are other assignments $a' := b'$ in D such that a is a lms of a', i.e., $a' \equiv a\ q_1\ \dots\ q_i$ for some $i > 0$, then add $a := b$ to D, and for each $a' := b'$, we truncate a' with the following steps:
   (a) delete $a' := b'$ from D.
   (b) for the pair $(b\ q_1\ \dots\ q_i, b')$, we recursively call Step 1 above.
3. if there is an assignment $a' := b'$ in D such that a' is a lms of a, i.e., $a \equiv a'\ q_1\ \dots\ q_i$ for some $i > 0$, then add one assignments $b'\ q_1\ \dots\ q_i := b$.
4. if there is not an assignment $a' := b'$ in D such that a is a lms of a' or a' is a lmf of a, then we create $a := b$ into D.

The algorithm above follows the constraints posted on an EP database, which are given in Definition 2.1.4. As an example, we take a database $\{a\ b := b; b\ a := a\}$ for a graph as the target database D_c and the first sample data $\mathcal{A}$ receives from $EX(Y(D_c), \mathcal{P})$ is $(a\ b\ a, a)$. Because the database D to be constructed is still empty, $\mathcal{A}$ would have to create an assignment $a\ b\ a := a$ and D becomes $\{a\ b\ a := a\}$. Later on, when $\mathcal{A}$ receives another sample data $(a\ b, b)$, then $\mathcal{A}$ will remove $a\ b\ a := a$ and add two more assignments $a\ b := b; b\ a := a$ according to Step 2. When we say that the assignment $a\ b\ a := a$ is removed from D, we also say that the assignee $a\ b\ a$ is "truncated" through the discussion in the rest of the paper, because it is "shortened" by two terms $b\ a$ and $a\ b$ in the database D.

For Step 3, suppose the database D has already had an assignment $a\ b := q$, and $\mathcal{A}$ receives another sample data $(a\ b\ o, p)$, then $\mathcal{A}$ will create another assignment $q\ o := p$ while keeping $a\ b := q$ unchanged.

In Section 5.2 and 5.3 as attached in Appendix A.5, we identify those terms $a \in \mathbf{E}$ that err $Y(D)$ by analyzing how D could deviate from D_c. In the section 5.4 as attached in Appendix A.5, we calculate the number m of calls to $EX(Y(D_c), \mathcal{P})$ that is needed to reach a confidence at least $(1 - \delta)$ of saying that the probability that the resulting database D gives a wrong answer (e.g., $Y(D)(a) \neq Y(D_c)(a)$)) for a term a drawn from $\mathbf{E}$ in the probability distribution $\mathcal{P}$ that has been applied to $EX(Y(D_c), \mathcal{P})$ is less than ϵ. The number m is determined as:

Lemma 5.2 The number m of calls to $EX(Y(D_c), \mathcal{P})$ that the algorithm $\mathcal{A}$ has to make is at least

$$(|D_c|/\epsilon)(ln(|D_c|) + ln(1/\epsilon))$$

such that with probability at least $1 - \delta$ the resulting bounded function $Y(D)$ will have error at most ϵ with respect to $Y(D_c)$ and $\mathcal{P}$. (See Appendix A.5 for the proof.)

In Section 5.5 as attached in Appendix A.5, we show $\mathcal{A}$ runs in polynomial time in $|D|, 1/\epsilon,$ and $1/\delta$.

Lemma 5.3 The complexity of running the algorithm $\mathcal{A}$ is at most

$$2k|D_c|^3(ln(|D_c|) + ln(1/\epsilon))/\epsilon$$

(See Appendix A.5 for the proof.)

Lemma 5.4 The running time for $Y(D(a))$ for $a \in \mathbf{E}$ is at most $3|a||D_c|$.(See Appendix A.5 for the proof.)

We have proved the result below:

Theorem 5.5 The concept class $\mathfrak{Y}_k$ for any $k \in \mathbf{N}$ is efficiently PAC learnable if the target database size $|D_c|$ is linear or polynomial in k. Otherwise, it is inefficiently PAC learnable.

Before ending this section, we present the result from Appendix E regarding $\mathbb{Y}_s$, the concept classes from the partial computation process for generating approximations to the lambda calculus:

Theorem E.5 The concept class $\mathbb{Y}_s$ for any $s \in \mathbf{N}$ is efficiently PAC learnable.

6 Related work

A class of approximations to a n-ary number-theoretic partial recursive function is not learnable. The difference in learnability between the approximations to a n-ary number-theoretical partial recursive functions and to the lambda calculus can be clearly described by the definition of an efficient (α, β)-Occam algorithm ([10]), which says that a consistent learning algorithm is efficient only if the size of a constructed program is significantly smaller than the sample size m.

Traditional database technologies do not have a PAC learning ability. Transitive relations exist in traditional databases, but explicit queries, i.e., functions with type and variables, must be defined to retrieve transitive relational data.

While a partial recursive function with finite representation cannot be learned theoretically, the PAC learnability of a class of bounded functions says that partial properties from the partial recursive function can be learned.

Statistical machine learning has unique strengths in pattern recognition, for which there is not a known and effective symbolic approach. On the other hand, a symbolic approach like EP has its strength in knowledge representation and reasoning, for which there is not a known and effective statistical approach (though many attempts have been made, but the result practically and theoretically is not a satisfaction [19], [3], [14], [1]).

Programming language requiring initial design of classes (data structure) and traditional database management system requiring initial design of data schema are top-down approaches toward software development. Machine learning is a bottom-up approach toward software development by learning from sample data. EP, a type and variable free language, is a bottom up approach.

The learnability of the classes of bounded functions is about multiclass classification. A multiclass classification for a non-statistic learning algorithm can be reduced to binary one by "All-vesus-all" or "All-pair" ([17]). In addition, we have a third reduction method for a class of bounded functions. Given a bounded function $Y(D) = \{(m,n) \mid m \in \mathbf{E}, n \in NF(D), \text{ and } m \to_D n\}$, we can rewrite it as:

1. $H(D)(m,n) = 1$, if $m \in \mathbf{E}$, $n \in NF(D)$, and $m \to_D n$; or
2. $H(D)(m,n) = 0$, if $m \in \mathbf{E}, n \in NF(D)$, and $m \not\to_D n$.

Note when $m \in \mathbf{E}$ and $n \notin NF(D), (m,n)$ is not in the instance space of $H(D)(m,n)$.

Since the learning algorithm in this paper is not statistical but symbolic, a reduction, in any one of the three methods, does not impact the learning performance in convergence.

7 Conclusions

The main contributions presented in this paper include: 1) the process of using the EP data model (structure) to accumulate data is a process of effectively PAC learning; 2) with the hypothesis of infinite time and space, the entire properties of a partial recursive function is efficiently PAC learnable, which says nothing but the capacity of reasoning or predicting with a better precision is maximized.

The efficient PAC learnability of a class of bounded functions introduces an opportunity to improve the performance of integrating symbolic approach, particularly embedding graphs into an Euclidean space, with statistic approaches in machine learning [19], [14], [1], [16], [8], [7]). It also renews the effort of using symbolic computations for natural language processing [12], [15], [20], [22], [21].

Acknowledgement Prof. Anselm Blumer, Prof. Gautam Kamath and many other reviewers, including those from COLT 2025, provided valuable inputs to previous versions of this manuscript, including suggestions on improving the presentation with materials related to the lambda calculus.

References

1. M. Apidianaki. From word types to tokens and back: a Survey of approaches to word meaning representation and interpretation. Computational Linguistics (2023) 49 (2): 465–523.
2. H. P. Barendregt. “The Lambda Calculus - its Syntax and Semantics”. North-Holland, 1984
3. J. Berant, I. Dagan, J. Goldberger (2012). Learning entailment relations by global graph structure optimization. Journal of Computational Linguistics, 38(1):73-111.
4. N. Brukhim, D. Carmon, I. Dinur, S. Moran, and A. Yehudayof. A characterization of multiclass learnability. 2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)
5. S. B. Cooper, Computability Theory. Chapman Hall CRC Mathematics, 2004.
6. A. Daniely and S. Shalev-Shwartz. Optimal learners for multiclass problems. In COLT, pages 287–316, 2014.
7. J. Flach and L. Lamb. A Neural Lambda Calculus: Neurosymbolic AI meets the foundations of computing and functional programming. arXiv:2304.09276.
8. A. Garcez and L. Lamb. Neurosysmbolic AI: The 3rd Wave. Artificial Intelligence Review, 2023.
9. E.M. Gold. Language Identification in the Limit. Information and Controls, 10, 447-474 (1967)
10. M. J. Kearns, U. V. Vazirani. An introduction to computational learning theory, 1994.
11. S.C. Kleene, Introduction to Meta-Mathematics. Ishi Press International, ISBN 0-923891-57-9.
12. M. Minsky, “"Logical vs. Analogical or Symbolic vs. Connectionist or Neat vs. Scruffy", in Artificial Intelligence at MIT., Expanding Frontiers, Patrick H. Winston (Ed.), Vol 1, MIT Press, 1990. Reprinted in AI Magazine, 1991.
13. B. K. Natarajan, On Learning Sets and Functions. Machine learning 4, 67-97, 1989.
14. M. Nickel, V. Tresp, H. P. Kriegel. Factorizing YAGO – Scalable Machine Learning for Linked Data. WWW2012 – Session: Creating and Using Links between Data Objects. April 2012, Lyon, France.
15. J. R. Pierce, J. B. Carroll, E. P. Hamp, D. G, Hays, C, F, Hockett, A. G. Oettinger, A. Perlis, “Language and Machines, Computers in Translation and Linguistics”, A Report by the Automatic Language Processing Advisory Committee, Division of Behavioral Science, National Academy of Sciences, National Research Council, National Academy of Science, National Research Councile, Publication 1416, 1966.

16. D. Rohatgi, T. Marwah, Z.C. Lipton, J.Lu, A.Moitra, A.Risteski. Towards characterizing the value of edge embeddings in Graph Neural Networks. arXiv:2410.09867v1.
17. S. Shalev-Shwartz and S. Ben-David. Understanding machine learning: From theory to algorithms. Cambridge University Press, 2014.
18. L.G. Valiant. A Theory of the Learnable. Communication of the ACM, November 1984, Volume 27, Number 11.
19. C. Seshadhri, A. Sharma, A. Stolman, A. Goel. The impossibility of low rank representation for triangle-rich complex network. The Proceedings of National Academy of Sciences, March 2020.
20. J. Weizenbaum, "Computational Linguistics, ELIZA – A Computer Program for the Study ofNatural Language Communication Between Man and Machine", Communications of the ACM, Volume. 9, Number. 1, January 1966.
21. K. Xu. Outline of a PAC Learnable Class of Bounded Functions Including Graphs. The 7th International Conference on Machine Learning and Intelligent Systems (MLIS 2024)
22. K. Xu. A deductive system based on Froglingo for natural language processing. May 2022, DOI: http://dx.doi.org/10.13140/RG.2.2.13859.32807
23. K. Xu. A class of bounded functions, a database language and an extended lambda calculus. Journal of Theoretical Computer Science, Vol. 691, August 2017, Page 81 - 106.
24. K. Xu, J. Zhang, S. Gao. Higher-level functions and their ordering-relations. The Fifth International Conference on Digital Information Management, 2010.
25. K. Xu, B. Bhargava. An Introduction to the Enterprise-Participant Data Model, the Seventh International Workshop on Database and Expert Systems Applications, September, 1996, Zurich, Switzerland, page 410 - 417.

Appendix A - Supplementary materials for Sections 2 to 5

A.2 Continuation to Section 2 - EP databases

A term alone without an assignment is allowed to be in a database. When a term is in a database, its subterms are considered in the database as well.

By terms alone (without the reduction rules to be introduced in Section 3), we can represent containment relationships. For example, the hierarchical structure of geographical locations can be expressed: $Florida\ Miami$; $France\ Paris$; $the\ United\ States\ of\ America\ (New\ York\ City)\ Manhattan\ (Water\ Street\ 55)$. The terms embed transitive relations, such as we can infer $Miami$ is part of $the\ United\ States\ of\ America$ because $Miami$ is part of $Florida$ and $Florida$ is part of $the\ United\ States\ of\ America$.

The rules in Definition 2.1.4 to form an EP database reflect the constraints to eliminate derivable (redundant) data when an approximation is syntactically transformed to an EP database, as given in Theorem C.4 in Appendix C. The algorithm to be introduced in Section 5 follows these rules to construct learned databases.

In addition to the example database discussed in Section 1, we list a few more sample databases here:

- $\{x\ x := x\}$, for a graph with a single vertex x, counting a vertex having a directed loopback edge
- $\{a\ b\ c := d; d\ (e\ f) := a\ b;\}$, for a random database.
- $\{college\ John\ major := college\ math; college\ math\ math100\ (college\ John)\ grade := A\}$, for a college administration database.
- $\{person\ (Verb\ drive)\ home := person\ (Verb\ arrive)\ (Prop\ at\ home)\ (Prop\ by\ car)\}$, for an English paraphrase in a symbolic approach to natural language processing [22]

A.3 Cont. to Section 3 - EP reductions and bounded functions

Here are a few sample reductions to their normal forms under the example databases provided in Appendix A.2:

$$x\ x\ \ldots\ x \to_D x$$
$$a\ b\ c \to_D d$$
$$a\ b\ c\ (e\ f) \to_D a\ b$$
$$d\ (e\ f)\ c \to_D d$$
$$college\ John\ major\ math100\ (college\ John)\ grade \to_D A$$
$$person\ (Verb\ drive)\ home \to_D person\ (Verb\ arrive)\ (Prop\ at\ home)\ (Prop\ by\ car)$$

A.4 Cont. to Section 4 - Concept classes dimension growing exponentially

The pseudo cube for [F, 2] The subset $S = \{a\ a, b\ b, a\ b, b\ a\}$ that is DS-shattered by the class of the bounded functions, a pseudo cube, defined by the set of databases [a, b, 2] is given in Figure 1 (which is automatically shifted to the end of the document by Latex).
Proposition 4.2 The size of the largest databases in $[F, k]$ and $dim([[F, k]])$ are exponential in k.
Proof Given a $k \in \mathbf{N}$, we can count the numbers of terms allowed in a database:
1) *Instance space.* Given a term $m_0, ..., m_i$, where $m_j \in F$ and $0 \leq j \leq i$ for a given i such that $0 \leq i \leq k$, we have a total of k^i terms with the exact size of i. When $i = k$, we have a total k^k unique terms with the exact size of k.

One of the largest database for a given k consists of all the assignees with the exact size of k, i.e., the number of assignees and therefore assignments in such a database is k^k. This proved that the size of the largest databases in $[F, k]$ is exponential in k. (The constraints on a database as given in Definition 2.1.4 and also Theorem C.4 do not allow a term with a size less than k to join a largest database as an assignee to make the database even bigger. For example, when $a\ b\ c$ with the size of $k = 3$ has already been in one of the largest databases, a term with a size of 2, such as $a\ b$, cannot be an assignee in the same database because a proper lms of an assignee cannot be an assignee simultaneously in a database.)

In $[F, k]$, there are databases with a size less than k^k. These databases includes all $[F_i, i]$ where $0 \leq i < k$, and also include those in which the assignees have different sizes ranging from 1 to k. Nevertheless, the largest databases with the size of k^k dominant $[F, k]$ because the number of the largest databases is larger than the sum of all other databases with size less than k^k. Therefore, we conclude with an estimation: $|[F, k]| < (k + 1)^{k+1}$.
2) *Normal forms.* With more than k^k assignees allowed in a database for a given $[F, k]$, we need to find all normal forms that can be assigners. In a database where the size of all the assignees is k, the allowed normal forms are those terms with the size less than k. There are a total of $k + k^2 + ... + k^{(k-1)}$ normal forms in $[F, k]$.
3) $[[F, k]]$*'s cardinality*: $(k + k^2 + ... + k^{(k-1)})^{k^k} < |[F, k]| = |[[F, k]]| < (k + k^2 + ... + k^{(k-1)})^{(k+1)^{k+1}}$. The dimension of the class is exponential: $dim([[F, k]]) = (k + 1)^{k+1}\ log(k + k^2 + ... + k^{(k-1)})$. □

Graph learnability Let $G_n = \{V, E\}$ to denote a complete (fully connected) undirected graph, where V, also denoted as $G_n.V$, is a set of nodes with a total number of nodes $|V| = n$. Each node in V is uniquely tagged with an identifiers, e.g., $V = \{v_1, ..., v_n\}$. E, also denoted as $G_n.E$, is the set of edges that fully connect the nodes with the number of edges $l = \frac{n\times(n-1)}{2}$. we use (i, j) to represent an edge such that, i.e., $(i, j) \in E$. We define a set of graphs, denoted as $\mathbf{G}_n^i$ for $0 \leq i \leq l$ such that each graph in $\mathbf{G}_n^i$ is a subgraph of G_n, i.e., $G_n^i.V = G_n.V$, $G_n^i.E \subseteq G_n.E$, and $i = |G_n^i.E|$. For a given $\mathbf{G}_n^i$, there are a total of $\binom{l}{i}$ subgraphs. We further define a sequence of such sets of subgraphs $\mathbf{G}_n = \{\mathbf{G}_n^0, ..., \mathbf{G}_n^l\}$, where $|\mathbf{G}_n| = \sum_{0\leq i\leq l} \binom{l}{i} = 2^l = 2^{\frac{n\times(n-1)}{2}}$. Clearly we have $G_n^l \equiv G_n$.

Now let a subgraph $G \in \mathbf{G}_n^i$ for a given $0 \leq i \leq l$ be represented in a database, denoted as D^G. Then we have

$$D^G = \bigcup_{(i,j)\in G.E; i\neq j} \{v_i\ v_j := v_j\} \cup \{v_j\ v_i := v_i\}$$

Because each subgraph in $\mathbf{G}_n$ has a database representation, the total number of the databases for $\mathbf{G}_n$ is $2^{\frac{n\times(n-1)}{2}}$, so is the number of the corresponding bounded functions. Instead of giving separate notations for the databases representing the sets of subgraphs and for the corresponding bounded functions, we simply say that a class of graphs $\mathbf{G}_n$ for a $n \in \mathbf{N}$ is efficiently PAC learnable:
Theorem 4.4 a class of graphs $\mathbf{G}_n$ for a $n \in \mathbf{N}$, precisely the corresponding class of bounded functions, is efficiently PAC learnable.
Proof It is PAC learnable because it has a correspondence with a class of bounded functions (shown in Section 5). The learnability is efficient because the dimension of the class of the bounded functions is polynomial: $log\ 2^{\frac{n\times(n-1)}{2}} \approx \frac{n\times(n-1)}{2}$ according to Theorem 2.2 of [10] and Theorem 5 of [13].□

A.5 Cont. to Section 5 - A class of bounded functions is PAC learnable

As a continuation of Section 5, we identify those terms $a \in \mathbf{E}$ that err $Y(D)$ by analyzing how D could deviate from D_c in Section 5.2 and 5.3. In the section 5.4, we calculate the number m of calls to $EX(Y(D_c), \mathcal{P})$ that is needed to reach a confidence at least $(1-\delta)$ of saying that the probability that the resulting database D gives a wrong answer (e.g., $Y(D)(a) \neq Y(D_c)(a))$) for a term a drawn from $\mathbf{E}$ in the probability distribution $\mathcal{P}$ that has been applied to $EX(Y(D_c), \mathcal{P})$ is less than ϵ. In Section 5.5, we show $\mathcal{A}$ runs in polynomial time in $|D|, 1/\epsilon,$ and $1/\delta$.

Section 5.2 A learned database is a subset of target database When the resulting database D after m calls to $EX(Y(D_c), \mathcal{P})$ is constructed with assignments exactly from D_c, i.e., $D \subseteq D_c$, we have $Y(D) \subseteq Y(D_c)$. This would happen when all the sample terms $S = \{a_0, a_1, ..., a_m\}$ from the sequence $(a_0, Y(D_c)(a_0))$, $(a_1, Y(D_c)(a_1))$, ..., $(a_m, Y(D_c)(a_m))$ that $EX(Y(D_c), \mathcal{P})$ generated are truncated to the corresponding assignees in D_c by Step 2 of $\mathcal{A}$.

Now let's see the terms that deviate their behaviors in D because some assignments in D_c are missing from D. If an assignment $(z, v) \in D_c$ is not made into D, then any term $a \in \mathbf{E}$ where $z \in SUB(a)$ and $Y(D_c(a)) \neq null$ [2] will most likely have a different result in $Y(D)$, i.e., $Y(D)(a) \neq Y(D_c(a))$. This is because most likely we have $z \rightarrow_D null$ and $Y(D)(a[null/z]) \rightarrow_D null$ while $Y(D_c)(a) \not\rightarrow_D null$. (Note the expression $a[null/z]$ is denoted as an substitution of all instances z with "$null$" in term a.) For example, if $z := c \in D_c$ and $c \not\equiv null$ but $z := c \notin D$, then if a term $a \equiv b\ z \rightarrow_{D_c} d$ for $d \not\equiv null$, it will make $Y(D)((b\ z)[null/z]) \neq Y(D_c(a))$ as $Y(D)((b\ z)[null/z]) \rightarrow_D null$.

On the other hand, there is a chance that $Y(D)(a[null/z]) = Y(D_c(a))$ when (z, v) is missing from D, which causes $z \rightarrow_D null$ [3]. For example, if $z := null \in D_c$ but not in D, then if a term $a \equiv b\ z \rightarrow_{D_c} c$ for $c \not\equiv null$, it will make $Y(D)((b\ z)[null/z]) = Y(D_c(b\ z)) = c$. If that's the case, we consider $Y(D)(a) \neq Y(D_c(a))$, i.e., we count it as an error even though it is actually not an error. This treatment is conservative in calculating the sample size m.

The situation where $(z, v) \in D_c$ but $(z, v) \notin D$ causes $Y(D)$ to err only on those terms a in which $a \not\rightarrow_D null$. It is also exactly such terms a that would have caused $\mathcal{A}$ to add (z, v) into D. We use $U(z)$ to denote all such terms a and express it as:

$$U(z) = \{a \mid z \in SUB(a); \exists q[z := q \in D_c]; Y(D_c(a)) \neq null; \forall q[z := q \notin D]\}$$

Section 5.3 A learned database with assignments not in target database After having m calls to $EX(Y(D_c), \mathcal{P})$, $\mathcal{A}$ may end up with a database D with some assignments in D but not in D_c. In this case, there is an assignment in D_c, e.g., $z_1 := v_1 \in D_c$, there is another assignment in D, e.g., $z_2 := v_2 \in D$, $z_2 := v_2 \notin D_c$, and $z_1 \in LMS^+(z_2)$. We observe any terms z between z_1 and z_2, i.e., $z_1 \in LMS(z)$ and $z \in LMS^+(z_2)$, $Y(D)(z) \neq Y(D_c)(z)$, but $Y(D)(z_2) = Y(D_c)(z_2)$ and $Y(D)(z_0) = Y(D_c)(z_0)$ for any $z_0 \in LMS^+(z_1)$. For example, when $(a\ b\ a := a) \in D$ and $\{a\ b := b; b\ a := a\} \subseteq D_c$, we have $(a\ b) \rightarrow_D a\ b$ and $(a\ b) \rightarrow_{D_c} b$. (This scenario is the reason that in Section 5.1, we allowed a learned database D_i to be in $\mathfrak{D}_{k+1}$ even it is targeting a database $D_c \in \mathfrak{D}_k$.)

Let $a \in \mathbf{E}$ and $z \in SUB(a)$, we summarize another set $V(z)$ in which the terms a having z as a subterm err $Y(D)$:

$$V(z) = \{a \mid \exists z_1[z_1 \in LMS(z); \exists q_1[z_1 := q_1 \in D_c];\ \exists z_2[z \in LMS^+(z_2)]; \exists q_2[z_2 := q_2 \in D]]; z \in SUB(a)\}$$

Section 5.4 - Sample size An assignment $z := v$ that is either in D or D_c but not both can cause $Y(D)$ to err on those terms a with z as a subterm. In Section 5.2 and 5.3, we summarized all such terms a with z as a subterms as $U(z)$ and $V(z)$. Let $p(z)$ denote the total probability of such terms a under the distribution $\mathcal{P}$, that is,

[2] It is a trivial case when $Y(D_c(a)) = null$ because $Y(D(a))$ will be reduced to null as well.

[3] This case happens from both practice and the partial computations to the lambda calculus. For example, we can have $(\mathbf{I}\ \Omega, \Omega) \in [\Lambda^0]^+_s$.

$$p(z) = \mathbf{Pr}_{a\in\mathcal{P}}\{U(z) \vee V(z)\} \leq \mathbf{Pr}_{a\in\mathcal{P}}(U(z)) + \mathbf{Pr}_{a\in\mathcal{P}}(V(z))$$

Since every error of $Y(D)$ can be caused by at least one assignment that is either in D or D_c but not both, by the union bound we have $error(Y(D)) \leq \sum_{z:=v\in D} p(z)$. We say that an assignment $z := v$ in either D_c or D but not both is *bad* if $p(z) \geq \epsilon/|D_c|$ [4]. If $Y(D)$ contains no bad assignments, then $error(Y(D)) \leq \sum_{z:=v\in D} p(z) \leq |D|(\epsilon/|D_c|) = \epsilon$. Now it's time to find the upper bound for the probability that a bad assignment will appear in Y(D).

For any fixed bad assignment $z := v$, the probability that this assignment is not added to D or its assignee z is not truncated to a proper lms which is an assignee in D_c after m calls of $\mathcal{A}$ to $EX(Y(D_c), \mathcal{P})$ is at most $(1 - \epsilon/|D_c|)^m$, because the probability the assignment $z := v$ is added or z is truncated by a single call to $EX(Y(D_c), \mathcal{P})$ is $p(z)$ (which is at least $\epsilon/|D_c|$ for a bad assignment). From this we may conclude that the probability that there is *some* bad assignment that is not added to D or its assignee is not truncated after m calls is at most $|D_c|(1 - \epsilon/|D_c|)^m$, where we have used the union bound over the $|D_c|$ possible assignments.

Thus to complete our analysis we simply need to solve for the value of m satisfying $|D_c|(1-\epsilon/|D_c|)^m \leq \delta$, where $1 - \delta$ is the designed confidence. Using the inequality $1 - x \leq e^{-x}$, it suffices to pick m such that $|D_c|e^{-m\epsilon/|D_c|} \leq \delta$, which yields $m \geq (|D_c|/\epsilon)(ln(|D_c|) + ln(1/\epsilon))$. We have proved the Lemma below:

Lemma 5.2 The number m of calls to $EX(Y(D_c), \mathcal{P})$ that the algorithm $\mathcal{A}$ has to make is at least

$$(|D_c|/\epsilon)(ln(|D_c|) + ln(1/\epsilon))$$

such that with probability at least $1 - \delta$ the resulting bounded function $Y(D)$ will have error at most ϵ with respect to $Y(D_c)$ and $\mathcal{P}$.

Section 5.5 - Run time In Section 5.2 and 5.3 as attached in Appendix A.5, we identify those terms $a \in \mathbf{E}$ that err $Y(D)$ by analyzing how D could deviate from D_c. In Section 5.4 as attached in Appendix A.5, we calculates the number m of calls to $EX(Y(D_c), \mathcal{P})$ that is needed to reach a confidence at least $(1 - \delta)$ of saying that the probability that the resulting database D gives a wrong answer (e.g., $Y(D)(a) \neq Y(D_c)(a)$)) for a term a drawn from $\mathbf{E}$ in the probability distribution $\mathcal{P}$ that has been applied to $EX(Y(D_c), \mathcal{P})$ is less than ϵ. The number m is determined as:

Lemma 5.3 The complexity of running the algorithm $\mathcal{A}$ is at most

$$2k|D_c|^3(ln(|D_c|) + ln(1/\epsilon))/\epsilon$$

Proof For each loop in the algorithm with an input (a, b) as defined in Definition 5.1, the time complexity to search the term a is in the worst case $|D_c|$ [5], so does the time for b. In other words, we need $2|D_c|$ time to identify the term $a\ b$.

We assume the worst case in Step 2 that all the assignees in the database need to be truncated. Then we need $|D_c|$ inner loops in running the algorithm $\mathcal{A}$. For each inner loop, the learning algorithm may need to recursively search the database again following Step 2 for investigating the pair $(b\ q_1\ \ldots\ q_i, n')$. The number of the recursive calls may at the worst case reach the size of the term $b\ q_1\ \ldots\ q_i$, i.e., $|n\ q_1\ \ldots\ q_i| \leq k$. Then we have the total time complexity $O(2|D_c| \times |D_c| \times k) = O(2k|D_c|^2)$ for each outer loop. Multiplying the time in each outer loop with the number of total outer loops m, we get $2k|D_c|^2 \times (|D_c|/\epsilon)(ln(|D|) + ln(1/\epsilon))$ $= 2k|D_c|^3(ln(|D_c|) + ln(1/\epsilon))/\epsilon$. □

Lemma 5.4 The running time for $Y(D(a))$ for $a \in \mathbf{E}$ is at most $3|a||D_c|$.

Proof Given an application term $m\ n \in \mathbf{E}$, the time to find m and n is at the worst case $|D_c|$ for each [6]. The time to reduce $m\ n$ to another is at worst case $|D_c|$. Therefore, the time to evaluate an application $m\ n$ is $3|D_c|$. The time to evaluate a term a with the length $|a|$ is at worst case $3|a||D_c|$. □

[4] here $|D_c|$ is the number of assignments in D_c. We choose $|D_c|$ as the divisor because one assignment in D_c but not in D and another assignment in D but not in D_c independently (and further we assume equally) contribute to $error(Y(D))$.

[5] The time complexisty to search a is actually $O(log(|D_c|))$ when EP terms are sorted in a database.

[6] Again the actual time complexity is improved by implementing index.

Appendix B - Approximations to the lambda calculus

To show that the EP data model is semantically equivalent to the lambda calculus (in the hypothesis of infinite time and space), a partial computation process, by taking one index mapping function that maps a closed lambda term to a unique integer, i.e., $\# : \Lambda^0 \to \mathbf{N}$, was developed in [23] to produce a sequence of approximations to the lambda calculus, that in turn are converted to a sequence of databases $D_0, D_1, ...$, and a sequence of bounded functions $Y(D_0), Y(D_1), ...$ In this section, we reiterate this process to further observe what an approximation is and how fast the size of an approximation grows when the partial computation steps increase. This discussion because essential when we extend the partial computation from a single index mapping function $\#$ to infinite index mapping functions $\#_0, \#_1, ...$, from which we develop a sequence of bounded function classes: $\mathbb{Y}_0, \mathbb{Y}_1, ...$, where each element $\mathbb{Y}_s$ for a $s \in \mathbf{N}$ is a target class that is proved to be efficiently PAC learnable in Appendix E.

Given a n-ary number-theoretic partial recursive function $\phi_e : \mathbf{N}^n \to \mathbf{N}$, e.g., in the form of Kleene's systems of equations [11], [5], and a number $s \in \mathbf{N}$, i.e., $\{0,\ 1,\ \ldots\}$, we have a partial computation that produces a finite approximation, denoted as $\phi_{e,s}$, of ϕ_e such that:

$$\phi_{e,s}(x_1, x_2, \ldots, x_n) = y \quad \text{if} \;\; x_1, \ldots, x_n, y \le s \;\text{ and }\; \phi_e(x_1, x_2, \ldots, x_n) \downarrow_s y$$

where $\phi_e(x_1, x_2, \ldots, x_n) \downarrow_s y$ denotes that $\phi_e(x_1, x_2, \ldots, x_n)$ converges within s steps and the resulting value is y. The approximation $\phi_{e,s}$ can be rewritten as a finite set of ordered pairs:

$$\phi_{e,s} = \{< (x_1, x_2, \ldots, x_n),\ y > \mid x_1, \ldots, x_n, y \le s \text{ and } \phi_e(x_1, x_2, \ldots, x_n) \downarrow_s y\}$$

and the union of all such approximations is semantically equivalent to ϕ_e itself, i.e., $\phi_e = \cup_{s \in \mathbf{N}}\ \phi_{e,s}$, where

$$\phi_e = \{< (x_1, x_2, \ldots, x_n),\ y > \mid x_1, \ldots, x_n \in \mathbf{N} \text{ and } \phi_e(x_1, x_2, \ldots, x_n) \downarrow y \text{ for } y \in \mathbf{N}\}$$

For a given ϕ_e, a n and a number of computation steps $s \in \mathbf{N}$, the instance space of $\phi_{e,s}$ is $\{y_0, y_1, ..., y_s\}^{\{x_0, x_1, ..., x_n\}}$ with the size of s^n, where $x_i, y_j \le s, 0 \le i \le n, 0 \le j \le s$. Although the exponential growth of the instance space, the actual cardinality of an approximation is bounded by s, i.e., $|\phi_{e,s}| \le s$.

In this section, we develop a partial computation of the lambda calculus by mimicking the partial computation of n-ary number-theoretic partial recursive functions. We note that many adjustments are required.

B.1 The properties of closed λ terms

Given a term $M_0 \in \Lambda^0$, where Λ^0 denotes the set of all closed lambda terms, we would like to have a partial computation for M_0. Instead of a fixed number n in the n-ary number-theoretical approximation approach, we need to consider all the terms that have M_0 as a leftmost subterm (lms), i.e., $M_0, M_0M_1, ..., M_0M_1 \ldots M_n, \ldots$ for all $M_1, M_2, \ldots M_n, \ldots \in \Lambda^0$ when we enumerate the properties for M_0:

$$[M_0] = \{(M_0\ M_1 \ldots M_n, Q)\ | M_1, \ldots, M_n, Q \in \Lambda^0; M_0\ M_1 \ldots M_n \to_\beta Q; n \in \mathbf{N}\} \tag{1}$$

where $\to_\beta$ is meant reductions in the β-reduction rule with the leftmost reduction strategy of the lambda calculus ([2]), and $[M^0]$ denotes all the properties M_0 has, i.e., pairs $(M_0\ M_1 \ldots M_n, Q)$ for any $n \in \mathbf{N}$ and any $M_i \in \Lambda^0$ where $0 < i \le n$. In the set (1) above, a term M_0 can be any of the following:

1. A closed term having normal form, i.e., $M_0 \to_\beta R$ where R is a normal form (cannot be further reduced, abbreviated as nf). We use NF to denote all terms having a normal form.
2. A closed term having head normal forms (hnf), i.e., there is a term $M_0\ M_1\ ...\ M_i \to_\beta \mathbf{I}$ where $i \ge 0$ and $\mathbf{I} \equiv \lambda x.x$. We use HNF to denote all terms having hnf. (Note that $NF \subset HNF$.). Example: $\lambda x.\mathbf{I}x\Omega$, where $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$, has no nf but a hnf $\lambda x.x\Omega$. We can find a term $(\lambda x.x\Omega)(\lambda xy.y)\mathbf{I} \to_\beta \mathbf{I}$.
3. A closed term having weak head normal forms (whnf), i.e., $M_0 \to_\beta R$ where R is an abstraction. We use $WHNF$ to denote all terms having whnf. (Note that $HNF \subset WHNF$.). Example, $\lambda x.\Omega$ doesn't have a hnf but is an abstraction.
4. A closed zero term, i.e., a term that don't have a whnf. We use $\Lambda^0 \backslash WHNF$ to denote the entire set of the zero terms. Example: Ω is a zero term.

The set (1) above gives the properties for a single closed term M_0, we would like to define a set for the properties of the entire set of the closed lambda terms:

$$[\Lambda^0]^{full} = \{(M_0\ M_1 \ldots M_n, Q) \mid M_0, M_1, \ldots, M_n, Q \in \Lambda^0; M_0\ M_1 \ldots M_n \to_\beta Q; n \in \mathbf{N}\} \tag{2}$$

To develop an approximation to the lambda calculus, we will consider all the terms having a *whnf*. We do not consider closed zero terms because applying any term to a zero term either yields the zero term itself, e.g., $\Omega \to_\beta \Omega$, or yields a term with a size [7] longer than the given zero term, i.e., $\Omega_3 \to_\beta \Omega_3(\lambda x.xxx)$ where $\Omega_3 \equiv (\lambda x.xxx)(\lambda x.xxx)$. With these said, we give the following definitions:

$$[\Lambda^0]' = \{(M_0\ M_1 \ldots M_n, Q) \mid M_0, M_1, \ldots, M_n, Q \in \Lambda^0; M_0\ M_1 \ldots M_n \to_\beta Q; M_0\ M_1 \ldots M_n \not\equiv Q; n \in \mathbf{N}\} \tag{3}$$

where $\not\equiv$ is meant not identical. Any zero terms that are β-reduced to themselves, like Ω, are filtered out from the set above.

To further filter out the zero terms with their size growing along with β-reductions, we define the following with a reference to $[\Lambda^0]'$:

$$[\Lambda^0] = \{(M_0\ \ldots\ M_n, Q) \mid (M_0\ \ldots\ M_n, Q) \in [\Lambda^0]';\ \exists (P, R) \in [\Lambda^0]'\ [Q \in SUB^+(P)]\} \tag{4}$$

Here, we use $SUB(P)$ to denote all the subterms of a term P, i.e., given $P \equiv M\ N$, then $M, N, M\ N \in SUB(P)$. We use $SUB^+(P)$ to denote all the proper subterms of P, i.e., $SUB^+(P) = SUB(P) \backslash \{P\}$ [8].

The set $[\Lambda^0]$ says that $(M_0\ \ldots\ M_n, Q) \notin [\Lambda^0]$ unless the size of Q is controlled (limited). If the size $|Q|$ of Q is not under control, i.e., growing as long as it could go, say infinite, the term P with $(P, R) \in [\Lambda^0]'$ in which Q is a subterm would be even longer, too long to spell them out. To control $|Q|$, we must not allow a zero term with a growing size to be M_0 in a $(M_0\ \ldots\ M_n, Q) \in [\Lambda^0]$. It has been proved that $(M_0\ \ldots\ M_n, Q) \in [\Lambda^0]$ if and only if M_0 is a *whnf* (Corollary 6.3 and 6.4 in [23]). Let $\mathbf{I} = \lambda x.x$, for example, the pair $(\mathbf{I}\ \mathbf{I}, \mathbf{I})$ will be in the set (4). As another example, $\lambda x.\Omega$ can be a M_0 such that $((\lambda x.\Omega)N, \Omega)$ for any $N \in \Lambda^0$ is in $[\Lambda^0]$. While a zero term cannot be M_0, it may be a M_i where $0 < i \leq n$ in $(M_0\ \ldots\ M_n, Q) \in [\Lambda^0]$. For example, we can have $(\mathbf{I}\ \Omega, \Omega)$ and $((\lambda xy.y)\ \Omega, \lambda y.y) \in [\Lambda^0]$ while Ω is the second element of the pair $((\lambda x.\Omega)\ N, \Omega) \in [\Lambda^0]$.

Another characteristic of the set $[\Lambda^0]$ is that given a term M_0, all the terms $M_0\ M_1,\ \ldots, M_0\ M_1\ \ldots\ M_n, \ldots$ are considered to be further β-reduced even when a term $M_0\ M_1\ \ldots\ M_n$ has been reduced with a result Q, i.e., $(M_0\ \ldots\ M_n, Q) \in [\Lambda^0]$. Recall that a term M_0 has a *hnf* if and only if there exists $M_1, \ldots, M_n \in \Lambda^0$, where $n \geq 0$, such that $M_0\ M_1\ \ldots\ M_n$ has a normal form (or reduced to $\mathbf{I}$). If a term $M_0\ M_1\ \ldots\ M_n$ with a fixed n doesn't have a normal form, it doesn't mean that the term $M_0\ M_1\ \ldots\ M_n\ M_{n+1}\ \ldots\ M_{n+i}$ doesn't have a normal form for a $i > 0$. Even if a normal form has been found for a term $M_0\ M_1\ \ldots\ M_n$, we continue to enumerate terms $M_0\ M_1\ \ldots\ M_n\ M_{n+1}\ \ldots\ M_{n+i}$ for $i > 0$ to explore potentially additional normal forms. For example, all the pairs $(\mathbf{I}\ \mathbf{I}, \mathbf{I}), (\mathbf{I}\ \mathbf{I}\ \mathbf{I}, \mathbf{I}), \ldots$ are in $[\Lambda^0]$. Clearly, some elements in $[\Lambda^0]$ are derivable from others (or we call some elements redundant). A reduction process to eliminate such derivable (redundant) terms is repeatedly used in defining EP databases and a learning algorithm later.

The set $[\Lambda^0]$ is the entire properties of the closed lambda terms in the lambda calculus, and we call it the semantics of the lambda calculus that have a correspondence with EP databases and bounded functions later. In the coming section, we first describe a partial computation process that produce approximations to the set $[\Lambda^0]$.

B.2 Partial computations for approximations

To give a partial computation process to produce approximations to the lambda calculus, precisely $[\Lambda^0]$, we assume (and we know it is possible) that all the terms in Λ^0 are enumerated in a sequence. We assume a specific sequence in this section (we will talk about other sequences later). Given $M \in \Lambda^0$, we use $\#M$ to

[7] the size of a term t, denoted as $|t|$, is the number of the symbols in t, [2].

[8] Later, we use $LMS(P)$ to denotes the set of all leftmost subterms of P for a $P \in \Lambda^0$. (Then we have $P \in LMS(P)$. If $M\ N \in LMS(P)$, so is M.) Further $LMS^+(P)$ denotes all the proper leftmost subterms of P, i.e., $LMS^+(P) = LMS(P) \backslash \{P\}$. (Clearly, $LMS(P) \subseteq SUB(P)$).

denote its index in the sequence. Using Cantor's diagonal method, we give the following set for a number of computation steps $s \in \mathbf{N}$ in a correspondence to the set $[\Lambda^0]'$:

$$[\Lambda^0]'_s = \{(M_0 \ldots M_n, Q) \mid n, \#M_0, \ldots, \#M_n, \#Q \leq s;\ M_0 \ldots M_n \rightarrow_{(\beta,s)} Q;\ M_0 \ldots M_n \not\equiv Q\} \quad (5)$$

where $\rightarrow_{(\beta,s)}$ is meant β-reductions that are performed within s computation steps. The set $[\Lambda^0]'_s$ is the result of the partial computations toward $[\Lambda^0]'$ that are performed within s steps. The size of the set is finite, i.e., $|[\Lambda^0]'_s| \leq s \ll |[\Lambda^0]'|$.

With a reference to $[\Lambda^0]'_s$, we define another set in a correspondence to the set $[\Lambda^0]$:

$$[\Lambda^0]_s = \{(M_0 \ldots M_n, Q) \mid (M_0 \ldots M_n, Q) \in [\Lambda^0]'_s;\ \exists (P, R) \in [\Lambda^0]'_s\ [Q \in SUB^+(P)]\} \quad (6)$$

Alternatively, we rewrite it as:
Definition B.1 An approximation to the lambda calculus is defined as the following set for a given number of partial computation steps $s \in \mathbf{N}$:

$$[\Lambda^0]_s = \{(M_0 \ldots M_n, Q) \mid n, \#M_0, \ldots, \#M_n, \#Q \leq s; M_0 \ldots M_n \downarrow_s Q\} \quad (7)$$

where $M_0 \ldots M_n \downarrow_s Q$ denotes that $M_0 \ldots M_n$ is β-reduced to Q within s steps using the leftmost reduction strategy, $M_0 \ldots M_n \not\equiv Q$, and there exists a pair $(P, R) \in [\Lambda^0]'_s$ such that $Q \in SUB^+(P)$.

As an approximation to $[\Lambda^0]$, the set $[\Lambda^0]_s$ for a given $s \in \mathbf{N}$ is finite as its size is limited by s:
Proposition B.2 (Theorem 2.5 in [23])

1. $[\Lambda^0]_s$ is finite, i.e., $|[\Lambda^0]_s| \leq s$.
2. $[\Lambda^0]_s \subseteq [\Lambda^0]_{s+1}$
3. $[\Lambda^0] = \cup_{s \in \mathbf{N}} [\Lambda^0]_s$

The conclusion $|[\Lambda^0]_s| \leq s$ comes from the assumption that a computation step makes only one β-reduction and adds at most one property pair into $[\Lambda^0]_s$.

Given a pair $(P, R) \in [\Lambda^0]_s$ for a $s \in \mathbf{N}$, we call P a s-redex and Q a s-reduct in $[\Lambda^0]_s$, where P is called a "s-redex" because it is reducible in s steps of the partial computations and Q is called "s-reduct" because it is a reduction result after s steps of the partial computations. The set $[\Lambda^0]_s$ retains all the pairs of s-redexes and s-reducts that we are interested in. This set as an approximation will be transformed to an EP database in the coming section.

Appendix C - EP databases from the approximations

With a finite set $[\Lambda^0]_s$, we move our attention away from the reduction process of lambda terms and focus on relationships among the terms available in the set $[\Lambda^0]_s$. Without the β-reduction rule, a term in $[\Lambda^0]_s$ no longer represents what it is originally represented with the β-reduction rule. Instead, it barely serves as a name representing one of its approximations specified in $[\Lambda^0]_s$ and the occurrences of the name in $[\Lambda^0]_s$ establish relationships with other terms. Further, the relationships among the terms in $[\Lambda^0]_s$ are preserved even if the closed terms Λ^0 are syntactically substituted with another set of identifiers. The set

$$[\Lambda^0]_s = \{(\mathbf{I}\ \mathbf{I}, \mathbf{I}); (\mathbf{I}\ \mathbf{W}, \mathbf{W}); (\mathbf{W}\ \mathbf{I}, \mathbf{I})\}$$

where $\mathbf{I} \equiv \lambda x.x$ and $\mathbf{W} \equiv \lambda x.xx$, for example, is equivalent to the set

$$\{a\ a := a; a\ b := b; b\ a := a\}$$

We say that the two sets are equivalent because they give the same relationships between two objects, represented by $\mathbf{I}$ and $\mathbf{W}$ in the first set or by a and b in the second set.

Before formally introducing a notion of database, we exclude some pairs that are derivable from others in $[\Lambda^0]_s$, based on the characteristic of the partial computation that some functions may be enumerated multiple times. We alternatively use the word "redundant" in the place of "derivable" in this article. The same redundant term elimination process is reused when a learning algorithm is introduced to prove the learnability of a concept class of bounded functions later.

In Section 2, we allowed two terms sharing the same lms to be s-redexes in $[\Lambda^0]_s$, i.e., for some $i > 0$:

$$(M_0\ M_1\ \ldots\ M_n,\ Q) \in [\Lambda^0]_s \tag{8}$$
$$(M_0\ M_1\ \ldots\ M_n M_{n+1}\ \ldots\ M_{n+i},\ Q') \in [\Lambda^0]_s \tag{9}$$

The lambda calculus tells us that there exists a third expression $Q\ M_{n+1}\ \ldots\ M_{n+i}$ such that it can be reduced to Q'. Therefore we may have a third pair in $[\Lambda^0]_s$, i.e.,

$$(Q\ M_{n+1}\ \ldots\ M_{n+i},\ Q') \in [\Lambda^0]_s \tag{10}$$

For example, When we have both $(\mathbf{I}\ \mathbf{I}\ ...\ \mathbf{I}, \mathbf{I})$ and $(\mathbf{I}\ \mathbf{I}, \mathbf{I})$ in $[\Lambda^0]_s$, we only need the latter and we can completely remove the former pair. Therefore, we exclude the second pair (9) from $[\Lambda^0]_s$ and the resulting set is denoted as $[\Lambda^0]_s^+$:
Definition C.1 For an $s \in \mathbf{N}$, a set, denoted as $[\Lambda^0]_s^+$, is defined as

$$[\Lambda^0]_s^+ = \{(M_0\ \ldots\ M_n, Q) \mid n, \#M_0, \ldots, \#M_n, \#Q \leq s; M_0\ \ldots\ M_n \downarrow_s Q;\ \forall P \in SUB^+(M_0\ \ldots\ M_n)[P \uparrow_s]\} \tag{11}$$

where $P \uparrow_s$ for a $P \in [\Lambda_0]$ denotes that there is not a $R \in \Lambda^0$ such that $(P, R) \in [\Lambda^0]_s^+$.

The set $[\Lambda^0]_s^+$ will be soon called an EP database right after we substitute its lambda term elements with independent identifiers (or function names). Before we start this process, we would like to see the effect of removing elements like the second pair (9) from $[\Lambda^0]_s^+$: the size of the s-redex P in each pair $(P, R) \in [\Lambda^0]_s^+$ is most likely reduced but not necessarily all the time. (We do this analysis because the size of a term impacts the cardinality of the database sets that can be produced from the partial computations to the lambda calculus as discussed in Section 5.) The size of a term is defined as the number of symbols in the term [2]. For a convinence, we can roughly view the size of a lambda term to be the number of its lmss, i.e., given a lambda term $M\ N$, its size $|M\ N| = |M| + 1$. Recall that the partial computation starts with a given term M_0 and subsequently evaluates all applications, $M_0\ M_1$, ..., $M_0\ M_1\ M_2\ ...\ M_n$ for $n \leq s$. Also recall a pair $(M\ N, Q)$, where $M\ N \to_\beta Q$, will be in $[\Lambda^0]_s^{'}$ as long as $M\ N \not\equiv Q$. Would the size of the term M be always 2 for all $(M, Q) \in [\Lambda^0]_s^+$? Many terms in $[\Lambda^0]_s^+$, such as $(\mathbf{I}\ \mathbf{I}, \mathbf{I})$, will have size 2. However, the answer in general is no. When a pair $(M\ N, Q)$ is calculated through a partial computation within computation steps s, where $M\ N \not\equiv Q$, the pair cannot be in $[\Lambda^0]_s$ if there doesn't exist another pair $(P, R) \in [\Lambda^0]_s^{'}$ and $Q \in SUB^+(P)$. Let $\mathbf{R} \equiv \lambda xyz.z$, for example, when $\mathbf{R}\ \mathbf{R}$ is being calculated, $\mathbf{R}\ \mathbf{R} \to_\beta (\lambda yz.z)\mathbf{R}$, where $(\lambda yz.z)\mathbf{R}$ has not appeared in $[\Lambda^0]_s^+$ (or $[\Lambda^0]_s^{'}$), then $(\mathbf{R}\ \mathbf{R}, (\lambda yz.z)\mathbf{R}) \notin [\Lambda^0]_s^{'}$. When the partial computation process continue to evaluate $\mathbf{R}\ \mathbf{R}\ \mathbf{R}$, since $\mathbf{R}\ \mathbf{R}\ \mathbf{R} \to_\beta \mathbf{R}$, we have $(\mathbf{R}\ \mathbf{R}\ \mathbf{R}, \mathbf{R})$ to be qualified to be in $[\Lambda^0]_s^+$.

Now we are ready to transform $[\Lambda^0]_s^+$ to an EP database. We first introduce a set (precisely a sequence) of infinitely many identifiers, denoted as $\mathbf{F}$. In $\mathbf{F}$, a given identifier, e.g., a, always has a fixed index number denoted as $\#a$. When the closed lambda terms Λ^0 is ordered in a sequence by a index mapping function $\#$, $\mathbf{F}$ is bijective with Λ^0 under $\#$. For $m \in \mathbf{F}$ and $M \in \Lambda^0, m$ is said to the identifier for the lambda term M if $\#M = \#m$. Through the article, we use capital letters to represent closed lambda terms, e.g., $M \in \Lambda^0$, and small letters to represent identifiers, e.g., $m \in \mathbf{F}$. Given a letter with different cases, e.g., m and M, we imply that $\#M = \#m$.
Definition C.2 For a set of ordered pairs of closed lambda terms S, a new set, denoted as $S(\mathbf{F}/\Lambda^0)$, is obtained by replacing closed terms in S with their corresponding identifiers in $\mathbf{F}$ and rewriting a pair as an assignment, that is

$$S(\mathbf{F}/\Lambda^0) = \{m_0\ \ldots\ m_i := q \mid (M_0\ \ldots\ M_n, Q) \in S\}$$

A set S in the definition above can be $[\Lambda^0]$, $[\Lambda^0]_s$, or $[\Lambda^0]_s^+$. We rewrite them as
$[\Lambda^0](\mathbf{F}/\Lambda^0) = \{m_0\ \ldots\ m_n := q \mid n, \#m_0,\ \ldots,\ \#m_i, \#q \geq 0; M_0\ \ldots\ M_n \downarrow Q\}$
$[\Lambda^0]_s(\mathbf{F}/\Lambda^0) = \{m_0\ \ldots\ m_n := q \mid n, \#m_0,\ \ldots,\ \#m_i, \#q \leq s; M_0\ \ldots\ M_n \downarrow_s Q\}$

$[\Lambda^0]^+_s(\mathbf{F}/\Lambda^0) = \{m_0 \dots\ m_n := q \mid n, \#m_0,\ \dots,\ \#m_i, \#q \leq s; M_0 \dots\ M_n \downarrow_s Q; \forall P \in SUB^+(M_0\ ...\ M_n)[P \uparrow_s]\}$

We take a closer look at the structure of the set $[\Lambda^0]^+_s(\mathbf{F}/\Lambda^0)$ for a $s \in \mathbf{N}$, which we call a database. In $[\Lambda^0]^+_s$, an element is an ordered pair, separated by ",", to demonstrate that the first element is reduced to the second element. In $[\Lambda^0]^+_s(\mathbf{F}/\Lambda^0)$, an element is an ordered pair, separated by ":=". We call such a pair an assignment to demonstrate that the assignee is assigned with the assigner as a value. With the definition, assignee, assigner, and their sub expressions in a correspondence inherit the structure of s-redex, s-reduct, and their subterms in $[\Lambda^0]^+_s$. To be able to construct databases without referencing the lambda calculus, we define them independently.

Definition C.3 Based on the set $\mathbf{F}$, we introduce a set of expressions, denoted as $\mathbf{E}$, that is defined inductively as follows

$$x \in \mathbf{F} \implies x \in \mathbf{E}$$
$$x, y \in \mathbf{E} \implies (x\ y) \in \mathbf{E}$$

We call a member of $\mathbf{E}$ an Enterprise-Participant (EP) *term*, or briefly a term whenever we can tell the differences between a lambda term and an EP term. Clearly, the assignees, the assigners, and their sub expressions in Definition C.3 are EP terms. Also note that $\mathbf{E}$ and Λ^0 are bijective too, i.e., each m in $\mathbf{E}$ has a corresponding M in Λ^0, and each M in Λ^0 has a corresponding m in $\mathbf{E}$.

We adopt all applicable notations of the lambda calculus for EP terms, e.g., (proper) subterm, (proper) leftmost subterm (*lms*), and omitting parentheses whenever possible when considering a term being parsed by the preference of left association, and $|m\ n| = |m| + 1$.

The set $[\Lambda^0]^+_s(\mathbf{F}/\Lambda^0)$ for a $s \in \mathbf{N}$ has a constrained structure as following, which becomes the constraints for EP databases (see Theorem 3.4 in [Xu 2017] for the proof):

Theorem C.4 Given a set $[\Lambda^0]^+_s(\mathbf{F}/\Lambda^0)$, renamed as D:

1. D is finite, i.e., a finite set of assignments.
2. Each assignee has only one assigner, i.e.,

$$p := q_1 \text{ and } p := q_2 \in \mathbf{D} \implies q_1 \equiv q_2$$

3. A proper subterm of an assignee cannot be an assignee, i.e.,

$$p := q \in D \implies \forall x \in SUB^+(p)[\forall m \in \mathbf{E}\ [x := m \notin D]]$$

4. An assigner must be a proper subterm of an assignee. i.e.,

$$p := q \in D \implies \exists c, d \in \mathbf{E}\ [c := d \in D \text{ and } q \in SUB^+(c)]$$

Note that an application in Definition C.3 can be mapped in different ways. Suppose that M and N are mapped to m and n in Definition C.3, where $m, n \in \mathbf{F}$ and therefore $\#m$ and $\#n$ are defined (and $\#(m\ n)$ is not defined since $(m\ n)$ is not in $\mathbf{F}$). We have two options to map the application $M\ N$. One is to map it to $m\ n$ and the other is to map it to a $p \in \mathbf{F}$ such that $\#p = \#(M\ N)$. Nevertheless, it makes no difference in choosing either $m\ n$ or p in $[\Lambda^0]^+_s(\mathbf{F}/\Lambda^0)$, as long as all the occurrences of $M\ N$ in $[\Lambda^0]^+_s$ are converted in the same manner. The reason is that the semantics of p and $m\ n$ converge to be identical when the computation steps s approaches infinity.

Definition C.5 A database D is a set of assignments $p := q$, where $p \in \mathbf{E}$ and $q \in \mathbf{E}$, such that the set meets the properties described in Theorems C.4.1, C.4.2, C.4.3, and C.4.4.

When an assignment $p := q$ is in a database D, denoted as $p := q \in D$, we also say that a subterm m of p or q is in the database D, denoted as $m \in D$.

Appendix D: The class Y is equivalent to Z

It was shown that the EP data model is semantically equivalent to the lambda calculus in [23]. The proof was done via a another function $Z(D_i)$ for all $i \geq 0$, where $\bigcup_{i \in \mathbf{N}} Z(D_i)$ is semantically equivalent to the lambda calculus (Theorem 7.2 in [23]). In this section, we relate $Y(D_i)$ closely together with $Z(D_i)$ and show that the sequence of $Y(D_0), Y(D_1), ...$ is semantically equivalent to the lambda calculus as well.

For any sequence of normal forms $m_0, \dots, m_k \in NF(D)$ and $k \geq 0$ under a given database D, there exists a normal form $q \in NF(D)$ such that $m_0 \, \dots \, m_k \to_D q$. That is, the set of all such pairs is:

$$Z(D) = \{(m_0 \, \dots \, m_k, q) \mid m_0, \dots, m_k, q \in NF(D); k \geq 0; m_0 \, \dots \, m_k \to_D q\}$$

The set $Z(D)$, from Corollary 4.9 in [23], was the key notion that allows an EP database to be expressed in a λ-term with *whnf* and therefore leads the proof that the EP data model is semantically equivalent to the lambda calculus in the hypothesis of infinite time and space in [23]. In the rest of the section, we show that $Y(D)$ and $Z(D)$ are semantically equivalent and therefore the infinite sequence (class) of bounded functions $Y(D_0), Y(D_1), ...$, where the sequence of databases $D_0, D_1, ...$ are converted from a sequence of approximations $[\Lambda^0]_0, [\Lambda^0]_1, ...$, is semantically equivalent to the lambda calculus.

Lemma D.1 Given a database D, $Z(D)$ is a proper subset of $Y(D)$, i.e., $Z(D) \subset Y(D)$.

Proof The function $Y(D) : \mathbf{E} \to NF(D)$ is defined as:

$$Y(D) = \{(m, n) \mid m \in \mathbf{E}, n \in NF(D), \text{ and } m \to_D n\}$$

The set (actually a function) $Z(D)$ is defined as

$$Z(D) = \{(m_0 \, \dots \, m_k, q) \mid m_0, \dots, m_k, q \in NF(D); k \geq 0; m_0 \, \dots \, m_k \to_D q\}$$

1. For any pair $(p, q) \in Z(D)$, the pair (p, q) is in $Y(D)$ as well, i.e., $(p, q) \in Y(D)$. This is because that $m_0, \dots, m_k, q \in NF(D)$ implies $m_0, \dots, m_k, q \in \mathbf{E}$, as $NF(D) \subset \mathbf{E}$, and because of the strong normalization of the EP reduction rules: any EP term is effectively reduced to one and only one normal form (Theorem 4.5 of [23]). Therefore, we conclude $Z(D) \subseteq Y(D)$.
2. There are some elements in $Y(D)$ that are not in $Z(D)$. Some terms that have *null* as the normal form are not considered in the set $Z(D)$. If $m \; null \; n := q \in D$ and $p \notin D$, for example, we have $m \; p \; n \to_D q$ according to Definition 4.2.2, but $(m \; p \; n, q) \notin Z(D)$. Therefore, $Z(D) \subset Y(D)$. □

Lemma D.1 states that the set $Y(D)$ is a true enumeration of a database D. The set $Z(D)$ is not an enumeration of a database D but $Z(D)$ is more essential than $Y(D)$, i.e., $Y(D)$ can be derived from $Z(D)$:

Lemma D.2 Given a database D, there is an enumeration process that produces $Y(D)$ from $Z(D)$.

Proof For any $m \equiv m_0 \; m_1 \; ... \; m_k \in \mathbf{E}$ for a $k \in \mathbf{N}$, we check individual terms $m_0, m_1, ..., m_k$ against the finite set of the normal forms $NF(D)$ in a given database D. If a term m_i for $0 \leq i \leq k$ is not a normal form, i.e., $m_i \notin NF(D)$, we reduce it into a normal form, say $m_i^{'} \in NF(D)$ and replace $m_i \in NF(D)$ with $m_i^{'} \in NF(D)$ in m. The resulting term, i.e., $m' \equiv m[m_0^{'}/m_0, m_1^{'}/m_1, ..., m_k^{'}/m_k]$, is the element in $Z(D)$ from which m can be derived such that $m \to_D m'$. □

Because $Z(D)$ is a proper subset of $Y(D)$, by Lemma D.1, and because $Y(D)$ can be derived from $Z(D)$, by Lemma D.2, we conclude that $Y(D)$ and $Z(D)$ are semantically equivalent:

Theorem D.3 1) Given a database D, $Y(D)$ and $Z(D)$ are semantically equivalent.
2) The sequence (class) of bounded functions $Y(D_0), Y(D_1), ...$, where the sequence of databases $D_0, D_1, ...$ are converted from a sequence of approximations $[\Lambda^0]_0, [\Lambda^0]_1, ...$, is semantically equivalent to the lambda calculus.

Proof 1) by Lemmas D.1 and D.2. □
2) Because the sequence $Z(D_0), Z(D_1), ...$ is semantically equivalent to the lambda calculus, where $D_0, D_1, ...$ are converted from a sequence of approximations $[\Lambda^0]_0, [\Lambda^0]_1, ...$ by Theorem 7.2 of [23], so is the sequence of $Y(D_0), Y(D_1), ...$ according to Theorem D.3.1. □

Appendix E: Concept classes from approximations to the lambda calculus

In Appendix B and C, we reviewed the partial computation process that takes a fixed sequence of the closed lambda terms, determined by an index mapping function #, to generate a sequence of approximations to the lambda calculus, which was originally introduced in [23]. By syntactically converting a sequence of the approximations, we obtain a sequence of databases, denoted as $D_0, ..., D_s$ for computation steps $s \in \mathbf{N}$, which are further interpreted as a sequence of bounded functions, denoted as $Y(D_0), ..., Y(D_s)$. This sequence of the bounded functions is not a concept class we are going to discuss regarding learnability because given a fixed mapping function #, only one database D_s is generated for a number of computation steps s, and $Y(D_s)$ is only one function. For a given s, we need a class of databases and a class of bounded functions from which a target function can be learned. In this section, we will introduce a sequences of classes that has its dimension (logarithm) growing polynomially and therefore is efficiently PAC learnable.

Rather than a single index mapping function #, we develop concept classes from the approximations that are generated from infinite many index mapping functions, denoted as $\#_0, \#_1, ...$, which are in correspondence to a fixed sequence of identifiers in $\mathbf{F}$ such as $a_0, a_1, ...$. For example, the λ term $\mathbf{I} \equiv \lambda x.x$ may be assigned with an index number 0 in $\#_0$ in a correspondence to the identifier a, and $\mathbf{W} \equiv \lambda x.xx$ is assigned with the same index number 0 in $\#_1$ in the same correspondence to the identifier a. We further give a learning algorithm that constructs a database to approximate another target database and show that the concept classes developed in this section are efficient PAC learnable.

For a given sequence of computation steps $0, 1, ..., s \in \mathbf{N}$ and for index mapping functions $\#_0, \#_1, ...$, we use $[\Lambda^0]_{\#_i,j}$ to denote the approximation, in a correspondence to a $[\Lambda^0]_s$ in Definition B.1, that are generated from partial computations to the set of closed lambda terms Λ^0 ordered by the index mapping function $\#_i$ within computation steps j. Therefore, given a s as the maximum computation steps, we will have sequences:

$$[[\Lambda^0]]_s = \{[\Lambda^0]_{\#_0,s}, [\Lambda^0]_{\#_1,s}, ...\}$$
$$\mathbf{D}_s = \{D_{\#_0,s}, D_{\#_1,s}, ...\}$$
$$\mathbf{Y}_s = \{Y(D_{\#_0,s}), Y(D_{\#_1,s}), ...\}$$

where $[[\Lambda^0]]_s$ is a sequence of approximations generated from partial computations up to computation steps s to the lambda terms ordered under index mapping functions $\#_0, \#_1, ...$ respectively, $\mathbf{D}_s$ is the corresponding sequence of databases converted from $[[\Lambda^0]]_s$, where $D_{\#_i,s}$ for $i \in \mathbf{N}$ is the database generated from the partial computations within steps s to the lambda terms ordered by the index mapping function $\#_i$, and $\mathbf{Y}_s$ is the sequence of bounded functions in a correspondence to $\mathbf{D}_s$.

Because s is finite and the number of identifiers in $\mathbf{D}_s$ is finite as the index mapping functions $\#_0, \#_1, ...$ are always in a correspondence to a fixed sequence of identifiers in $\mathbf{F}$ such as $a_0, a_1, ...$, it is clear that $\mathbf{D}_s$ has only a finite number of databases while each database has a finite number of assignments, as infinite many databases are identical although the number of unique approximations in $[[\Lambda^0]]_s$ is infinite. With this said, we redefine $\mathbf{D}_s$ and $\mathbf{Y}_s$:

$$\mathbf{D}_s = \bigcup_{i \in \mathbf{N}} \{D_{\#_i,s}\}$$
$$\mathbf{Y}_s = \bigcup_{i \in \mathbf{N}} \{Y(D_{\#_i,s})\}$$

Proposition E.1 1) A single databases $D_{\#_i,s}$ for any $i, s \in \mathbf{N}$ takes at most s assignments, i.e., $|D_{\#_i,s}| \leq s$. 2) $|\mathbf{D}_s|$ and $|\mathbf{Y}_s|$ are equal and they are finite.
Proof 1) The result is based on Proposition B.2 and Definition C.1, where each computation step produces at most one assignment.
2) We have $|\mathbf{D}_s| = |\mathbf{Y}_s|$ because $\mathbf{D}_s$ has a one to one correspondence with $\mathbf{Y}_s$. They are finite as we have discussed earlier in this subsection. □
Proposition E.2 The set of assignees a single database $D_{\#_i,s}$ for any $i, s \in \mathbf{N}$ can potentially take is $X_s = \{a_0, ..., a_s\}^{|\{a_0,...,a_s\}|}$ with the size of at most s^s.
Proof In $[\Lambda^0]^+_s(\mathbf{F}/\Lambda^0)$ as defined in Definition C.3, each assignee $m_0 \ldots m_n$ has the maximum size of s because $n \leq s$. Each identifier m_i for $0 \leq i \leq n$ in an assignee can be any one of the first s identifiers from the given sequence of identifiers $\mathbf{F}$. Therefore, the maximum number of potential assignees in a database is at most s^s. □

We had said earlier that the total number of databases in $\mathbf{D}_s$ is finite because the total number of identifiers and the size of each term in the database are finite. We are giving a better estimation on how many databases in $\mathbf{D}_s$.

Proposition E.3 The cardinality of the set of bounded functions $\mathbf{Y}_s$ is $|\mathbf{Y}_s| < s^{(2s^2)}$.

Proof Like in $[F, k]$, we only consider the databases with the assignees having the largest size s in the set $\mathbf{D}_s$. Such databases dominates $\mathbf{D}_s$ as the number of these databases is larger than the number of the others with smaller sizes.

1) *Instance spaces for databases in* $\mathbf{D}_s$ Let X be the set of assignees in a database $D \in \mathbf{D}_s$, alternatively called an instance space. For two different databases $D_1, D_2 \in \mathbf{D}_s$, although their sizes are the same in general, i.e., $|D_1| \approx |D_2| \leq s$, the instance spaces X_1 and X_2 may be completely different. Let $\mathbf{X}_s$ denote the set of all such instance spaces. The maximum number of different instance spaces of the set of databases in $\mathbf{D}_s$ is the number of choices to select s assignees without repetition from the total number of potential assignees s^s, i.e.,

$$|\mathbf{X}_s| \leq \binom{s^s}{s} < s^{(s^2)}$$

2) *Normal forms for database sets* The maximum size of a term in a database is at most s according to Definition C.2 (defined in Appendix C). Therefore a normal form must be a term with size less than s according to the definition of database in Definition 2.1. Therefore, there are a total of $1 + 1^1 + 2^2 ... + (s-1)^{s-1} < s^s$ unique normal forms, denoted as $|NF|$, in $\mathbf{D}_s$.

3) *The size* $|\mathbf{D}_s|$ *and* $|\mathbf{Y}_s|$ For a unique instance space X, the number of the total databases taking the instance space X is $|NF|^s \leq (s^s)^s = s^{(s^2)}$ as a database has only at most s assignments. For a total of $|\mathbf{X}| < s^{(s^2)}$ different instance spaces for databases in $\mathbf{D}_s$, there are at most $|NF|^s \times |\mathbf{X}_s| < s^{(2s^2)}$ databases. Therefor $|\mathbf{D}_s| = |\mathbf{Y}_s| < s^{(2s^2)}$.□

With the class of bounded functions $\mathbf{Y}_s$ over the set of instance spaces $\mathbf{X}_s$ for a given $s \in \mathbf{N}$, we can define a family of such classes $\mathbb{Y}_s = \bigcup_{0 \leq i \leq s} \mathbf{Y}_i$ over $\mathbb{X}_s = \bigcup_{0 \leq i \leq s} \mathbf{X}_i$. In correspondence, we denote the family of databases as $\mathbb{D}_s = \bigcup_{0 \leq i \leq s} \mathbf{D}_i$.

Proposition E.4 1. $|\mathbb{D}_s| \leq |\mathbb{D}_{s+1}|$
2. $|\mathbb{Y}_s| \leq |\mathbb{Y}_{s+1}|$
3. $dim(\mathbb{D}_s)$ and $dim(\mathbb{Y}_s)$ are in polynomial in s: $log\ |\mathbb{D}_s| = log\ |\mathbb{Y}_s| < (2s^2+1) log(s)$.

Proof The conclusions 1 and 2 are clear from the definitions of $\mathbb{D}_s$ and $\mathbb{Y}_s$. The conclusion 3 comes from Proposition E.3:

$$log\ |\mathbb{D}_s| = log(|\mathbf{Y}_0| + ... + |\mathbf{Y}_s|) < log(s \times |\mathbf{Y}_s|) < log\ s \times s^{(2s^2)} = (2s^2+1) log(s)$$

□

The polynomially growing dimension of $\mathbb{Y}_s$ has determined that $\mathbb{Y}_s$ is efficiently PAC learnable:

Theorem E.5 The concept class $\mathbb{Y}_s$ for any $s \in \mathbf{N}$ is efficiently PAC learnable.

Proof According to Theorem 2.2 of [10] and Theorem 5 of [13], the concept class $\mathbb{Y}_s$ for any $s \in \mathbf{N}$ is efficiently PAC learnable because $log|\mathbb{Y}_s| < (2s^2+1) log(s)$, a polynomial in s. □

Appendix F - DS dimensions of concept classes

In this section, we discuss the DS dimensions of the concept classes we have discussed so far. There are no new conclusions from this section. But it provides an integral and consistent view.

In parallel to a sequence of database sets $\mathbb{D} = \{\mathbb{D}_0, \mathbb{D}_1, ..., \}$ (accordingly $[F_0, 0], [F_1, 1], ...$) and correspondingly a sequence of bounded function sets $\mathbb{Y} = \{\mathbb{Y}_0, \mathbb{Y}_1, ..., \}$ (accordingly $[[F_0, 0]], [[F_1, 1]], ...$), in this section, we continue to use general notations $\mathfrak{D}_0, \mathfrak{D}_1, ...$ and $\mathfrak{Y}_0, \mathfrak{Y}_1, ...$ that are applicable to an arbitrary sequence of database classes and corresponding sequence of bounded function classes in addition to $\mathbb{Y}_k$ and $[F, k]$. We show the DS dimension of a class $\mathfrak{Y}_k$ for a $k \in \mathbf{N}$ is the size of the corresponding database.

Before we do so, we need to give additional notations. Given a class $\mathfrak{D}_k$ for any $k \in \mathbf{N}$, we use $\mathbf{NF}(\mathfrak{D}_k)$ to denote all the normal forms in individual databases of $\mathfrak{D}_k$, i.e., $\mathbf{NF}(\mathfrak{D}_k) = \bigcup_{D \in \mathfrak{D}_k} NF(D)$. Given a $\mathfrak{D}_k$, we use $\mathbf{LMS}(\mathfrak{D}_k)$ and $\mathbf{LMS}^+(\mathfrak{D}_k)$ for all the lmss and the proper lmss in $\mathfrak{D}_k$ respectively. Further we use $\mathbf{LMS}^{-}(\mathfrak{D}_k)$ to denote all improper *lms*s, i.e., $\mathbf{LMS}(\mathfrak{D}_k)\backslash\mathbf{LMS}^{+}(\mathfrak{D}_k)$. For example, $\mathbf{LMS}^{-}(\mathfrak{D}_k) = \{a\ a, a\ null\}$ for the class of the databases $\mathfrak{D}_k$ is

$$\{\{\}, \{a\ null := a\}, \{a\ a := a\}, \{a\ null := a; a\ a := a\}\}$$

We single out the improper *lms*s $\mathbf{LMS}^{-}(T)$ to show that a subset of such *lms*s determines the DS dimension of the corresponding class of bounded functions.

To be consistent with literature such as [17], [6], [4] in notation, we exchangeably use $\mathcal{H}$ in the place of $\mathfrak{Y}_k : \mathbf{E} \times \mathbf{NF}(\mathfrak{D}_k)$, which can be written as $\mathfrak{Y}_k \subseteq (\mathbf{NF}(\mathfrak{D}_k))^{\mathbf{E}}$.

We use $\mathcal{Y}$ to denote the co-domain $\mathbf{NF}(\mathfrak{D}_k)$ of $\mathfrak{Y}_k$, and S to denote a finite subset of $\mathbf{E}^d$, i.e., $S \subset \mathbf{E}^d$, where $d \in \mathbf{N}$, such that S would be DS-shattered by $\mathfrak{Y}_k$. For $h : \mathbf{E} \to \mathbf{NF}(\mathfrak{D}_k)$, and $S = (x_1, \ldots, x_d) \in \mathbf{E}^d$, the projection $h|_S$ of h to S is the map from $[d]$ to $\mathfrak{Y}_k$ defined by $i \mapsto h(x_i)$, where $[d] = \{1, 2, \ldots, d\}$. The projection of $\mathfrak{Y}_k$ to S is $\mathcal{H}|_S = \{h|_S : h \in \mathcal{H}\} \subseteq (\mathbf{NF}(\mathfrak{D}_k))^d$. (Note that h is used for a bounded function $Y(D)$ where $D \in \mathfrak{D}_k$.)

Given a $\mathfrak{Y}_k$, there is a $S \subseteq \mathbf{LMS}^{-}(\mathfrak{D}_k)$ such that $(\mathbf{NF}(\mathfrak{D}_k))^{|S|} \subseteq \mathfrak{Y}_k$ shatters S. As special cases, $[F_0, 0] \equiv \{\{\}, 0\}$ and $[F_1, 1] \equiv \{\{a\}, 1\}$ do not have an assignee, therefore they have 0 as the DS dimension, a trivial case. Before we look at $[F_2, 2]$, we see the database class above having the the pseudo cube:

	a a	*a null*
{}	*null*	*null*
{a null := a}	*null*	*a*
{a a := a}	*a*	*null*
{a null := a; a a := a}	*a*	*a*

where the columns *a a* and *a null* show the outputs of $Y(D)(x)$ when $x \in \{a\ a, a\ null\}$ and D is an element in the above database class. In the example above, $|(\mathbf{NF}(\mathfrak{D}_k))^{\mathbf{LMS}^{-}(\mathfrak{D}_k)}| = |\mathfrak{Y}_k|$, i.e., $\mathfrak{Y}_k$ shatters its own instance space.

We will see the DS dimension of $[F_k, k]$ with $k > 1$ in general. But we give an analysis on $[F_2, 2] = [\{a, b\}, 2]$ first. Because there are a total of 3 normal forms $a, b, null$ and there are a total of 4 assignees: $a\ a, a\ b, b\ b, b\ a$, one of the largest databases has 4 assignees (assignments) and we have a total of 81 databases. In this case, $[[\{a, b\}, 2]]$ shatters the 4 assignees $\{a\ a, a\ b, b\ b, b\ a\}$. See Appendix A.1 for a 4×81 matrix as the pseudo cube of $[[\{a, b\}, 2]]$.

For a $[F_k, k]$ in general, we will have the concept class $[[F_k, k]]$ to shatter the set of the entire set of improper *lms*s with the size of $k + k^2 + ... + k^{(k-1)}$. For $[F_3, 3]$, there are a total of 13 normal forms (including *null*), 27 improper assignees, and $|[[F_3, 3]]| = 13^{27}$. Therefore the pseudo cube is represented in a matrix with size of 27×13^{27}.

Because a database cannot be practically constructed exponentially to fill up the maximum capacity of k^k assignments in a class $[F_k, k]$, as if a linked list cannot be fully populated in practice, we are not bothered to have the negative result that $[[F_k, k]]$ is not efficiently PAC learnable. (The high cardinality actually demonstrates a high expressiveness of the EP data model as a positive characteristic.) Instead, our focus is on those databases that can be constructed linearly or polynomially, which is the nature of the entire computation theory having the Turing machine that can only sequentially (and parallelly) produce partial computational results and the nature of our computation practice today and tomorrow. A good example of such applications we are interested in is a class of graphs defined in Section 4 (as attached in Appendix A.4) that is efficiently PAC learnable.

Regarding a database class $\mathbb{D}_s$ for a given $s \in \mathbf{N}$ converted from approximations to the lambda calculus, we have a maximum number of s identifiers and of assignees in a database. Such a linear growth of the cardinality of a database says nothing but that not all improper *lms*s as assignees in $\mathbb{D}_s$ are shattered by $\mathbb{Y}_s$. This is due to the natural of the partial computations to the lambda calculus: By the time the computation steps s ends up with a set of improper *lms*s $\mathbf{LMS}^{-}(\mathbb{D}_s)$, it is most likely that not all the databases have been generated in $\mathbb{D}_s$. In other words, the corresponding concept class $\mathbb{Y}_s$ most likely has not been filled with adequate components to DS-shatter the instance set of the improper *lms*s $\mathbf{LMS}^{-}(\mathbb{D}_s)$. However, there is a smaller computation steps $s^{'}$ such that the concept class $\mathbb{D}_{s'}$ has been fully constructed and $\mathbb{Y}_{s'}$ (and therefore $\mathbb{Y}_s$) DS-shatters the instance set of the improper *lms*s $\mathbf{LMS}^{-}(\mathbb{D}_{s'})$. For example, $\mathbb{Y}_{s'}$ for a large s

would include $[[\{a, b\}, 2]]$ that guarantees the DS dimension of $\mathbb{Y}_{s'}$ is at least 4. Although the instance space of $\mathbb{Y}_s$ includes additional assignees like $a\ b\ c$, but none of the databases in $\mathbb{D}_s$ are constructed enough to have the additional assignees being DS-shattered.

With that being said, we will show that given a $\mathfrak{D}_k$ with a corresponding pseudo cube $[[F, k^{'}]]$, the instances S shattered by $\mathfrak{Y}_k$ is the set of the improper assignees $\mathbf{LMS}^{-}([F, k^{'}])$.

Lemma F.1 Given a $\mathfrak{D}_k$ with a corresponding pseudo cube $[[F_{k'}, k']]$, let $D \in [F_{k'}, k']$ and $m := p \in D$. Then m or a term t, where $t \to_D m$, must be an instance of S.

Proof Assume m and a term t are not in S. We can find another $D_1 \in [F_{k'}, k^{'}]$ and $Y(D_1) \in [[F_{k'}, k^{'}]]$ such that $D = D_1 \ \cup \{m := p\}$ because D is always constructed on the top of another database (and also the pseudo cube $[[F_{k'}, k']]$ requires such an D_1 exists). In this case, the two corresponding bounded functions show the same behavior, i.e., $Y(D)(x_i) = Y(D_1)(x_i)$ for all $i \in [d]$ in S. This contradicts that S is shattered. (Note when $m := p$ is in D but not in D_1, D_1 that we choose as a proper subset of D would never have an assignment $n := q$ such that the assignee n is a proper lms of m, i.e., $n \in LMS^{-}(m)$. For example, if a single-assignment database is $D \equiv \{a\ b\ c := p\}$, then we would choose $\{\}$ but not $\{a\ b := p\}$ as D_1.) □

Lemma F.2 Given a $\mathfrak{D}_k$ with a corresponding pseudo cube $[[F_{k'}, k']]$, let $D \in [F_{k'}, k']$ and two equivalent terms t_1 and t_2 , i.e., $t_1 =_D t_2$. The two terms t_1 and t_2 cannot be two instances in S simultaneously.

Proof Assume t_1 and t_2 are both in S and $t_2 \to_D t_1$ for a $D \in [F_{k'}, k']$. Assume $t_1 \to_D n$, the normal form of t_1 under D. Then $t_2 \to_D n$ as well. Certainly, we can find another database $D_1 \in [F_{k'}, k']$ such that $t_1 \to_{D_1} n$ and $t_2 \to_{D_1} n$ (when the DS-dimension of $[[F_{k'}, k']]$ is larger than 2), such as when there is a single assignment in both D and D_1 that drives the same reduced result n. But we are more interested in finding another database $D_2 \in [F_{k'}, k']$ such that $t_1 \to_{D_2} p$, where $p \not\equiv n$, and at the same time $t_2 \to_{D_2} q$, where $q \not\equiv n$. Certainly, there is possibly a database $D_3 \in [F_{k'}, k']$, such that $t_1 \to_{D_3} p$ and $t_2 \to_{D_3} n$. But if that is the case, we can always find D_2, where D_2 and D_3 are identical except for $t_2 \to_{D_2} q$ and $t_2 \to_{D_3} n$. The properties of the pseudo cube $[[F_{k'}, k']]$ guarantees that such D_2 and D_3 exist in $[F_{k'}, k']$. This concludes: $Y(D)(t_1) \not\equiv Y(D_2)(t_1)$ and $Y(D)(t_2) \not\equiv Y(D_2)(t_2)$. This is a contradiction to the conclusion that S is shattered by $[[F_{k'}, k']]$ □

Lemma F.3 A normal form cannot be an instance in S.

Proof Given $m := p \in D$ and $t \to_D m$, where we have $m \to_D p$. If p, a normal form, is in S, then both m and t cannot be in S according to Lemma F.2. But this is contradict to Lemma F.1 which requires either m or t be in S. □

Theorem F.4 Given a $\mathfrak{D}_k$ with the largest pseudo cube $[[F_{k'}, k']]$ having the instances set S being DS-shattered by $[[F_{k'}, k']]$ (and therefor $\mathfrak{Y}_k$), the elements of S come from $\mathbf{LMS}^{-}(\mathfrak{D}_k)$, i.e., $S \subseteq \mathbf{LMS}^{-}(\mathfrak{D}_k)$ and precisely $|S| = |\mathbf{LMS}^{-}([F_{k'}, k'])|$.

Proof Since a normal form cannot be an instance in S (Lemma F.3), two equivalent terms cannot be in S simultaneously (Lemma F.2), and either an assignee or another term that is reducible to the assignee must be in S (Lemma F.1), we choose assignee to be in S.

If an assignee t_1 is a proper lms of another assignee t_2 in $\mathbf{NF}(\mathfrak{D}_k)$, then t_1 is no longer an assignee but a normal form in a database D where t_2 is an assignee. Because a normal form cannot be an instance in S, t_1 is not an instance of S.

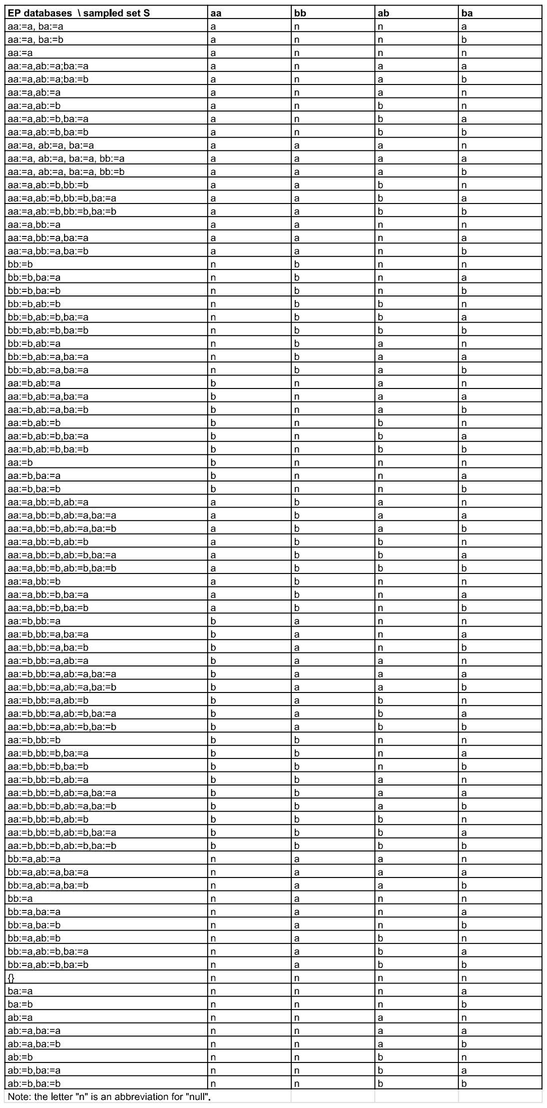
Because the concept class $\mathfrak{Y}_k$ has the given (largest) pseudo cube $[[F_{k'}, k']]$, the instances shattered by $\mathfrak{Y}_k$ are precisely those $\mathbf{LMS}^{-}([F_{k'}, k'])$, i.e., $|S| = |\mathbf{LMS}^{-}([F_{k'}, k'])|$. □

The number of assignees are the same number of assignments in a database. Therefore, we can say that the DS dimension of $\mathfrak{Y}_k$ is proportional to the number of assignments in one of the largest databases in $\mathfrak{D}_k$. We also say that a concept class $\mathfrak{Y}_k$ is efficiently PAC learnable if a database is practically constructible, i.e., the size of the largest database in the corresponding database class grows linearly or polynomially in k.

Theorem F.5 A concept class $\mathfrak{Y}_k$ is efficiently PAC learnable if a database is practically constructible, i.e., the size of the largest database in the corresponding database class grows linearly or polynomially in k.

Proof Given a class of databases $\mathfrak{D}_k$, let the number of assignees in one of the largest databases and the number of the maximum size of a term t are k^n for a constant $n > 0$. This is nothing more than Theorem E.5 except that the number s is replace by k^n. Therefore $log|\mathbb{Y}_{k^n}| < (2(k^n)^2 + 1)log(k^n) = 2k^{2n}\ n\ log\ k$, a polynomial in k

According to Theorem 2.2 of [10] and Theorem 5 of [13], $\mathfrak{Y}_k$ is efficiently PAC learnable because $log|\mathbb{Y}_{k^n}| < 2k^{2n}\ n\ log\ k$, a polynomial in k. □



EP databases \ sampled set S	aa	bb	ab	ba
aa:=a, ba:=a	a	n	n	a
aa:=a, ba:=b	a	n	n	b
aa:=a	a	n	n	n
aa:=a,ab:=a;ba:=a	a	n	a	a
aa:=a,ab:=a;ba:=b	a	n	a	b
aa:=a,ab:=a	a	n	a	n
aa:=a,ab:=b	a	n	b	n
aa:=a,ab:=b,ba:=a	a	n	b	a
aa:=a,ab:=b,ba:=b	a	n	b	b
aa:=a, ab:=a, ba:=a	a	a	a	n
aa:=a, ab:=a, ba:=a, bb:=a	a	a	a	a
aa:=a, ab:=a, ba:=a, bb:=b	a	a	a	b
aa:=a,ab:=b,bb:=b	a	a	b	n
aa:=a,ab:=b,bb:=b,ba:=a	a	a	b	a
aa:=a,ab:=b,bb:=b,ba:=b	a	a	b	b
aa:=a,bb:=a	a	a	n	n
aa:=a,bb:=a,ba:=a	a	a	n	a
aa:=a,bb:=a,ba:=b	a	a	n	b
bb:=b	n	b	n	n
bb:=b,ba:=a	n	b	n	a
bb:=b,ba:=b	n	b	n	b
bb:=b,ab:=b	n	b	b	n
bb:=b,ab:=b,ba:=a	n	b	b	a
bb:=b,ab:=b,ba:=b	n	b	b	b
bb:=b,ab:=a	n	b	a	n
bb:=b,ab:=a,ba:=a	n	b	a	a
bb:=b,ab:=a,ba:=a	n	b	a	b
aa:=b,ab:=a	b	n	a	n
aa:=b,ab:=a,ba:=a	b	n	a	a
aa:=b,ab:=a,ba:=b	b	n	a	b
aa:=b,ab:=b	b	n	b	n
aa:=b,ab:=b,ba:=a	b	n	b	a
aa:=b,ab:=b,ba:=b	b	n	b	b
aa:=b	b	n	n	n
aa:=b,ba:=a	b	n	n	a
aa:=b,ba:=b	b	n	n	b
aa:=a,bb:=b,ab:=a	a	b	a	n
aa:=a,bb:=b,ab:=a,ba:=a	a	b	a	a
aa:=a,bb:=b,ab:=a,ba:=b	a	b	a	b
aa:=a,bb:=b,ab:=b	a	b	b	n
aa:=a,bb:=b,ab:=b,ba:=a	a	b	b	a
aa:=a,bb:=b,ab:=b,ba:=b	a	b	b	b
aa:=a,bb:=b	a	b	n	n
aa:=a,bb:=b,ba:=a	a	b	n	a
aa:=a,bb:=b,ba:=b	a	b	n	b
aa:=b,bb:=a	b	a	n	n
aa:=b,bb:=a,ba:=a	b	a	n	a
aa:=b,bb:=a,ba:=b	b	a	n	b
aa:=b,bb:=a,ab:=a	b	a	a	n
aa:=b,bb:=a,ab:=a,ba:=a	b	a	a	a
aa:=b,bb:=a,ab:=a,ba:=b	b	a	a	b
aa:=b,bb:=a,ab:=b	b	a	b	n
aa:=b,bb:=a,ab:=b,ba:=a	b	a	b	a
aa:=b,bb:=a,ab:=b,ba:=b	b	a	b	b
aa:=b,bb:=b	b	b	n	n
aa:=b,bb:=b,ba:=a	b	b	n	a
aa:=b,bb:=b,ba:=b	b	b	n	b
aa:=b,bb:=b,ab:=a	b	b	a	n
aa:=b,bb:=b,ab:=a,ba:=a	b	b	a	a
aa:=b,bb:=b,ab:=a,ba:=b	b	b	a	b
aa:=b,bb:=b,ab:=b	b	b	b	n
aa:=b,bb:=b,ab:=b,ba:=a	b	b	b	a
aa:=b,bb:=b,ab:=b,ba:=b	b	b	b	b
bb:=a,ab:=a	n	a	a	n
bb:=a,ab:=a,ba:=a	n	a	a	a
bb:=a,ab:=a,ba:=b	n	a	a	b
bb:=a	n	a	n	n
bb:=a,ba:=a	n	a	n	a
bb:=a,ba:=b	n	a	n	b
bb:=a,ab:=b	n	a	b	n
bb:=a,ab:=b,ba:=a	n	a	b	a
bb:=a,ab:=b,ba:=b	n	a	b	b
{}	n	n	n	n
ba:=a	n	n	n	a
ba:=b	n	n	n	b
ab:=a	n	n	a	n
ab:=a,ba:=a	n	n	a	a
ab:=a,ba:=b	n	n	a	b
ab:=b	n	n	b	n
ab:=b,ba:=a	n	n	b	a
ab:=b,ba:=b	n	n	b	b

Note: the letter "n" is an abbreviation for "null".

Fig. 1. The pseudo cube for [F, 2]