# (Agnostic) PAC Learning Concepts in Higher-order Logic

K.S. Ng

Symbolic Machine Learning and Knowledge Acquisition
National ICT Australia Limited
kee.siong@nicta.com.au

**Abstract.** This paper studies the PAC and agnostic PAC learnability of some standard function classes in the learning in higher-order logic setting introduced by Lloyd et al. In particular, it is shown that the similarity between learning in higher-order logic and traditional attribute-value learning allows many results from computational learning theory to be 'ported' to the logical setting with ease. As a direct consequence, a number of non-trivial results in the higher-order setting can be established with straightforward proofs. Our satisfyingly simple analysis provides another case for a more in-depth study and wider uptake of the proposed higher-order logic approach to symbolic machine learning.

## 1 Introduction

Symbolic machine learning is traditionally studied in the field of Inductive Logic Programming (ILP). Within ILP, there is a rich body of work on the PAC-learnability (and non-PAC-learnability) of different classes of first-order logic programs. See, for surveys, [9] and [5]. The arguments used in this kind of analyses are usually intimately and intricately linked to the computation model of first-order logic programming. It is not clear whether these results, which reflect the nature of learning with a first-order language like Prolog, reflect the nature of learning with rich expressive languages in general. To bridge this gap in our understanding, we need to explore learnability issues in formalisms other than first-order logic programming. This paper is an attempt in this endeavour.

In particular, we will look at the higher-order logic approach to symbolic learning expounded in [12]. We will examine the PAC and agnostic PAC learnability of several common function classes definable in this new logical setting. Our main observation is that the similarity in nature between learning in higher-order logic and traditional attribute-value learning allows many results from computational learning theory to be 'ported' with ease to the higher-order setting. A direct consequence of this is that a number of non-trivial results in the logical setting can be shown with relatively straightforward proofs. The simplicity of our analysis, when compared to similar but more technical analyses in ILP, provides another piece of evidence that symbolic machine learning can be fruitfully studied in the higher-order setting proposed in [12].

The paper is organized as follows. I review the learning in higher-order logic setting in §2. The main results of this paper are in §3. §4 then concludes.

## 2 Learning in Higher-order Logic

The logic underlying our learning setting is a polymorphically typed, higher-order logic based on Church's simple theory of types. The form of the language is similar to that of a standard functional programming language like Haskell. Indeed, the approach grew out of research into a functional logic programming language called Escher [11]. In what follows, I will assume the reader is familiar with the syntax and terminology of functional programming languages.

We consider only binary classification problems in this paper. In standard attribute-value learning, the set of individuals $X$ is a subset of $\mathbb{R}^m$ for some $m$, and the hypothesis space $H$ consists of predicates (boolean functions) defined on $\mathbb{R}^m$. The logical setting introduced in [12] extends this basic setup in two ways.

1. The set $X$ is equated with a class of terms called basic terms in a higher-order logic. This class of terms includes $\mathbb{R}^m$ and just about every data type in common use in computer science.
2. The set $H$ is extended to admit any subset of computable predicates on basic terms definable by composing simpler functions called transformations.

We now examine these two points in some detail.

*Representation of Individuals* We first look at how training examples are represented in the logic. The basic idea is that each individual $x$ in a labelled example $(x, y)$ should be represented as a closed term. All information is captured in one place. In this sense, learning in higher-order logic is close to the learning from interpretations [6] and learning from propositionalized data [10] settings in ILP. The formal basis for this is provided by the concept of a *basic term*. Essentially, one first defines the concept of a term in higher-order logic. A suitably rich subset is then identified for data modelling. The details of this development can be found in [12]. For the purpose of this paper, it is sufficient to know that a rich catalogue of data types is provided via basic terms, and these include integers, floating-point numbers, strings, tuples, sets, multisets, lists, trees, graphs and composite types that can be built up from these.

We now introduce a simple multiple-instance problem to illustrate the representation language. More complicated applications can be found in [3] and [15]. We have a collection of bunches of keys and a door. A bunch of keys is labelled true ($\top$) iff it contains at least one key that opens the door, false ($\bot$) otherwise. Given the bunches of keys and their labels, the problem is to learn a function to predict whether any given bunch of keys opens the door.

We model a bunch of keys as a set of keys. Each key, in turn, is modelled as a tuple capturing two of its properties: the company that makes it and its length. This leads to the following type declarations.

$$\mathit{type\ Bunch} = \{\mathit{Key}\} \qquad \mathit{type\ Key} = \mathit{Make} \times \mathit{Length}$$

The types *Make* and *Length* have the following data constructors.

$$\mathit{Abloy}, \mathit{Chubb}, \mathit{Rubo}, \mathit{Yale} : \mathit{Make} \qquad \mathit{Short}, \mathit{Medium}, \mathit{Long} : \mathit{Length}$$

A training example may look like this.

$$\mathit{opens}\ \{(\mathit{Abloy}, \mathit{Medium}), (\mathit{Chubb}, \mathit{Long}), (\mathit{Rubo}, \mathit{Short})\} = \top$$

Given a set of such examples, we want to learn the function $\mathit{opens} : \mathit{Bunch} \to \varOmega$. Here and in the following, $\varOmega$ denotes the type of the booleans.

Given such data, one can proceed with distance-based learning methods by plugging into standard learning algorithms suitable kernels and distance measures defined on basic terms. In this paper, however, we will adopt a more symbolic approach to the problem. For each learning problem, we will need to explicitly define a space of predicates in which to search for a suitable candidate solution. We next describe the mechanism used to define predicate spaces.

*Predicate Construction* Predicates are constructed incrementally by composing basic functions called transformations. Composition is handled by the function $\circ : (a \to b) \to (b \to c) \to (a \to c)$ defined by $((f \circ g)\ x) = (g\ (f\ x))$.

**Definition 1.** *A* transformation $f$ *is a function having a signature of the form*

$$f : (\varrho_1 \to \varOmega) \to \cdots \to (\varrho_k \to \varOmega) \to \mu \to \sigma$$

*for* $k \geqslant 0$. *The type* $\mu$ *is called the* source *of the transformation; and* $\sigma$, *the* target *of the transformation. The number* $k$ *is called the* rank *of the transformation.*

Every function is potentially a transformation — just put $k = 0$. Transformations are used to define a particular class of predicates called standard predicates.

**Definition 2.** *A* standard predicate *is a term of the form*

$$(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$$

*for some* $n \geq 1$, *where* $f_i$ *is a transformation of rank* $k_i$, *the target of* $f_n$ *is* $\varOmega$, *and each* $p_{i,j_i}$ *is a standard predicate.*

In applications, we first identify a class of transformations $H$ of relevance to the problem domain. Having done that, we then build up a class of standard predicates by composing transformations taken from $H$ in appropriate ways. To illustrate this process, we will first look at some useful transformations for the multiple-instance Keys problem introduced earlier.

*Example 3.* The transformation $\mathit{top} : a \to \varOmega$ defined by $(\mathit{top}\ x) = \top$ for each $x$ is the weakest predicate on the type $a$ one can define.

*Example 4.* Given terms of type *Key*, which are tuples, we can introduce the function $\mathit{projMake} : \mathit{Key} \to \mathit{Make}$ defined by $(\mathit{projMake}\ (t_1, t_2)) = t_1$ to project out the first element. Likewise, we can define *projLength*.

*Example 5.* Given terms of type *Key*, we can introduce the transformation $\wedge_2 : (\mathit{Key} \to \varOmega) \to (\mathit{Key} \to \varOmega) \to \mathit{Key} \to \varOmega$ defined by $(\wedge_2\ p\ q) = \lambda x.((p\ x) \wedge (q\ x))$ to conjoin predicates on *Key*.

*Example 6.* Given terms of type *Bunch*, which are sets of keys, we can introduce the transformation $\mathit{setExists}_1 : (\mathit{Key} \to \Omega) \to \mathit{Bunch} \to \Omega$ defined by $(\mathit{setExists}_1\ p\ q) = \exists x.((x \in q) \wedge (p\ x))$. The term $(\mathit{setExists}_1\ p\ q)$ evaluates to $\top$ iff there exists a key in $q$ that satisfies $p$. This transformation directly capture the notion of multiple-instance learning.

*Example 7.* Given the transformations identified so far, we can construct (complex) standard predicates on *Bunch*. For example, the predicate

$$\mathit{setExists}_1\ (\wedge_2\ (\mathit{projMake} \circ (= \mathit{Rubo}))\ (\mathit{projLength} \circ (= \mathit{Medium})))$$

evaluates a set $s$ of keys to $\top$ iff there exists a Rubo key of medium length in $s$. One can easily construct other examples.

To efficiently enumerate a class of predicates for a particular application, we use a construct called predicate rewrite systems. It is similar in design to Cohen's antecedent description grammars [4]. We give an informal description of it now. A *predicate rewrite* is an expression of the form $p \rightarrowtail q$, where $p$ and $q$ are standard predicates. The predicate $p$ is called the *head* of the predicate rewrite; $q$, the *body*. A *predicate rewrite system* is a finite set of predicate rewrites. The following is a simple predicate rewrite system for the Keys problem.

$$\begin{aligned}
&\mathit{top} \rightarrowtail \mathit{setExists}_1(\wedge_2\ \mathit{top}\ \mathit{top})\\
&\mathit{top} \rightarrowtail \mathit{projMake} \circ (= C) \quad \text{for each constant } C : \mathit{Make}\\
&\mathit{top} \rightarrowtail \mathit{projLength} \circ (= C) \quad \text{for each constant } C : \mathit{Length}
\end{aligned}$$

Roughly speaking, predicate generation works as follows. Starting from an initial predicate $r$, all predicate rewrites that have $r$ (of the appropriate type) in the head are selected to make up child predicates that consist of the bodies of these predicate rewrites. Then, for each child predicate and each redex in that predicate, all child predicates are generated by replacing each redex by the body of the predicate rewrite whose head is identical to the redex. This generation of predicates continues to produce the predicate class. For example, the following is a path in the predicate space defined by the rewrite system given above.

$$\begin{aligned}
&\mathit{top} \rightsquigarrow \mathit{setExists}_1(\wedge_2\ \mathit{top}\ \mathit{top}) \rightsquigarrow \mathit{setExists}_1(\wedge_2\ (\mathit{projMake} \circ (= \mathit{Abloy})\ \mathit{top})\\
&\rightsquigarrow \mathit{setExists}_1(\wedge_2\ (\mathit{projMake} \circ (= \mathit{Abloy})\ (\mathit{projLength} \circ (= \mathit{Short})))
\end{aligned}$$

The space of predicates defined by a predicate rewrite system $\rightarrowtail$ is denoted $S_{\rightarrowtail}$.

Given a predicate rewrite system $\rightarrowtail$, we can define more complex function classes in terms of predicates defined by $\rightarrowtail$. Two function classes in actual use [15] we will look at in this paper are

1. $k$-DT($\rightarrowtail$) – the class of all decision trees of maximum depth $k$ taking predicates in $S_{\rightarrowtail}$ as tests in internal nodes; and
2. $k$-DL($\rightarrowtail$) – the class of all decision lists [16] where each test in an internal node is a conjunction of at most $k$ predicates in $S_{\rightarrowtail}$. (We can close $S_{\rightarrowtail}$ under negation if we wish to.)

Other common function classes can be defined (and analysed) in a similar way.

## 3 PAC Learnability of Higher-order Concepts

We examine the PAC learnability of $k$-DL($\rightarrowtail$) and $k$-DT($\rightarrowtail$) in this section, focusing mainly on the former. We assume familiarity with the standard definitions of PAC and agnostic PAC learning. The efficient computability of higher-order predicates is studied in §3.1. Sample complexity questions are briefly looked at in §3.2. With these in place, some results on the efficient (agnostic) PAC learnability of $k$-DL($\rightarrowtail$) and $k$-DT($\rightarrowtail$) are given in §3.3 and §3.4.

### 3.1 Efficiently Computable Predicates

As pointed out in [5], polynomial computability of concept classes is a prerequisite for efficient PAC learnability. We now give a sufficient condition on predicate rewrite systems that will ensure the production of only polynomially computable predicates. Predicate classes defined on such restricted rewrite systems can then be shown to contain only concepts that can be efficiently evaluated.

**Definition 8.** *A transformation* $f : (\varrho_1 \to \Omega) \to \cdots \to (\varrho_k \to \Omega) \to \alpha \to \sigma$ *is said to be* polynomial-time computable qua transformation *if* $(f\, p_1 \ldots p_k)$ *is polynomial-time computable given that each* $p_i$ *is polynomial-time computable.*

**Proposition 9.** *Let* $f : \alpha \to \sigma$ *and* $g : \sigma \to \phi$ *be polynomial-time computable functions. Then the function* $f \circ g$ *is polynomial-time computable.*

**Proposition 10.** *Let* $T$ *be a set of transformations and let* $S_T$ *be the set of all standard predicates that can be formed using transformations in* $T$*. If every* $f \in T$ *is polynomial-time computable qua transformation, then every* $p \in S_T$ *composed of a finite number of transformations is polynomial-time computable.*

*Proof.* The proof proceeds by induction on the number of transformations in $p$ and uses Proposition 9. □

In defining a predicate rewrite system $\rightarrowtail$, if we restrict ourselves to transformations that are polynomial-time computable qua transformation, then we can be assured that every $p \in S_{\rightarrowtail}$ we will ever construct is polynomial-time computable since $S_{\rightarrowtail} \subseteq S_T$. Further, given such a $\rightarrowtail$, it's easy to show that $k$-DL($\rightarrowtail$) and $k$-DT($\rightarrowtail$) contain only polynomially computable predicates.

### 3.2 Sample Complexity

It is well-known that the (agnostic) PAC learnability of a function class is tightly governed by its VC dimension [2]. The problem of calculating the VC dimension of general predicate classes definable using predicate rewrite systems was previously considered in [14]. The main conclusion reached there is that in the higher-order logic setting, the VC dimension of a predicate class is usually not much lower than the upper bound given by the logarithm of the size of the predicate class. Given this observation, we consider only finite function classes here. A rewrite system $\rightarrowtail$ is said to be *finite* if $S_{\rightarrowtail}$ is finite. We assume from now onwards all predicate rewrite systems are finite. Since all finite function classes are (agnostic) PAC learnable, we are interested primarily in *efficient* learnability.

### 3.3 Efficient (Agnostic) PAC Learnability of $k$-DL($\rightarrowtail$)

We now look at the learnability of higher-order decision lists, starting with the PAC learnability of $k$-DL($\rightarrowtail$). Under reasonable assumptions on the encoding sizes of individuals and the target function $t$ ($size(t) = \log_2 |k\text{-DL}(\rightarrowtail)|$), one can show the following is true; Rivest's proof [16] goes through essentially unchanged.

**Proposition 11.** *Let $X$ be a set of individuals and $\rightarrowtail$ a finite predicate rewrite system made up of only transformations that satisfy Definition 8. Then the class $k$-DL($\rightarrowtail$) is efficiently PAC learnable with sample complexity*

$$m(\epsilon, \delta) \leqslant \frac{1}{\epsilon}(O((S_{\rightarrowtail})^l) + \ln \frac{1}{\delta})$$

*for some constant $l$.*

The ease with which Proposition 11 can be established should not belie its importance. It extends Rivest's theorem beyond simple decision lists defined on boolean vectors to arbitrarily rich efficiently computable higher-order decision lists defined on arbitrarily complex structured data. In fact, it's worth pointing out that $k$-DL($\rightarrowtail$) is probably the largest class of functions that has ever been shown to be efficiently PAC learnable.

We now study the learnability of $k$-DL($\rightarrowtail$) in the more realistic agnostic PAC learning setting. The correct (and only) strategy in this setting is simple: find the predicate in the predicate class with the lowest error on the training examples. (See, for details, [2, Chap. 23].) Computing the decision list with the lowest empirical error given a set of training examples is, unfortunately, a computationally difficult problem. Indeed, one can show that there is no efficient algorithm for this optimization problem in the propositional setting, which is a special case of the general problem, unless P=NP. We now sketch a proof.

The argument is an adaptation of the proof for [2, Thm 24.2]. Consider the following two decision problems.

1. Vertex-Cover : Instance: A graph $G = (V, E)$ and an integer $k \leqslant |V|$.
   Question: Is there a vertex cover $U \subseteq V$ such that $|U| \leqslant k$?
2. DL-Fit : Instance: $z \in (\{0,1\}^n \times \{0,1\})^m$ and an integer $k \in \{1, \ldots, m\}$.
   Question: Is there $h \in$ 1-DL($n$) such that $\hat{er}(h, z) \leqslant k/m$?

A vertex cover of a graph $G = (V, E)$ is a set $U \subseteq V$ of vertices such that at least one vertex of every edge in $E$ is in $U$. In the definition of DL-Fit , $\hat{er}(h, z)$ is defined to be $|\{(x, y) \in z : h(x) \neq y\}|/m$ and 1-DL($n$) is as defined in [16].

It is known that Vertex-Cover is NP-hard. One can show that every Vertex-Cover problem can be reduced in polynomial time to a DL-Fit problem. Consider an instance $G = (V, E)$ of Vertex-Cover where $|V| = n$ and $|E| = r$. We assume that each vertex in $V$ is labelled with a number from $\{1, 2, \ldots, n\}$ and we denote by $ij$ an edge in $E$ connecting vertex $i$ and vertex $j$. The size of the instance is $\Omega(r + n)$. We construct $z(G) \in (\{0,1\}^n \times \{0,1\})^{r+n}$ as follows. For any two integers $i, j$ between 1 and $n$, let $e_{i,j}$ denote the binary vector of length $n$ with ones in positions $i$ and $j$ and zeroes everywhere else. The sample $z(G)$ consists of the labelled examples $(e_{i,i}, 1)$ for $i = 1, 2, \ldots, n$ and, for each edge $ij \in E$, the labelled example $(e_{i,j}, 0)$. The size of $z(G)$ is $(r+n)(n+1)$, which is polynomial in the size of the original Vertex-Cover instance.

*Example 12.* Consider the graph $G = \{\{1, 2, 3, 4\}, \{11, 12, 13, 14, 23, 33\}\}$. Then

$$\begin{aligned} z(G) = \{&(1000, 1), (0100, 1), (0010, 1), (0001, 1), \\ &(1000, 0), (1100, 0), (1010, 0), (1001, 0), (0110, 0), (0010, 0)\}. \end{aligned}$$

Using the fact that 1-DL($n$) is a subset of linear threshold functions [1], it is easy to show that the following is true.

**Proposition 13.** *Given any graph $G = (V, E)$ with $n$ vertices and $r$ edges and any integer $k \leqslant n$, let $z(G)$ be as defined above. There is $h \in \texttt{1-DL}(n)$ such that $\hat{er}(h, z(G)) \leqslant k/(n + r)$ iff there is a vertex cover of $G$ of cardinality at most $k$.*

Now, if there is an algorithm that, given a set $\mathcal{E}$ of examples, can find in polynomial time $\arg\min_{h \in 1\text{-}DL(n)} \hat{er}(h, \mathcal{E})$, then it can be used to solve DL-Fit in polynomial time. But since DL-Fit is NP-hard, such an algorithm cannot exists unless P=NP. This means $k$-DL($\rightarrowtail$) is not efficiently agnostic PAC learnable under standard complexity-theoretic assumptions.

### 3.4 Efficient (Agnostic) PAC Learnability of *k*-DT($\rightarrowtail$)

We end the section with brief remarks on the learnability of 1-DT($\rightarrowtail$) (higher-order decision stumps) and $k$-DT($\rightarrowtail$) for $k > 1$ (higher-order decision trees).

Given a set $X$ of individuals and a predicate rewrite system $\rightarrowtail$, the class 1-DT($\rightarrowtail$) is not efficiently (agnostic) PAC learnable. This is unsurprising since (agnostic) PAC learning 1-DT($\rightarrowtail$) entails an exhaustive search of $S_{\rightarrowtail}$. The run time thus cannot be bounded by a polynomial in $size(t)$.

The efficient learnability of decision trees remains one of the longest-standing open problems in computational learning theory. In particular, it is not known whether the problem of computing the most accurate decision tree given a set of examples is hard. The weak learning framework provides probably the best chance for obtaining positive results; see [8].

## 4 Discussion and Conclusion

We conclude by comparing our analysis with similar studies in ILP. The comparison is centred around the basic setup of a learning problem. There are two main logical settings in first-order learning: learning from entailment [13] and learning from interpretations [7]. Learning in higher-order logic, being a direct generalization of attribute-value learning, is closer to the latter. The two share the following features with attribute-value learning, which are not found in the learning from entailment setting:

1. examples and background knowledge are separated;
2. examples are separated from one another.

The advantage of making such separations is argued convincingly in [6]. In particular, making such separations allows many results and algorithms from propositional learning to be easily 'upgraded' to the richer settings. This paper provides further evidence in support of this general observation.

In the learning from interpretations setting, there is a price in making such separations in that recursive predicates cannot be learned. This limitation can be overcome in the higher-order setting via the use higher-order functions like `foldr` that package up recursion into convenient forms. This suggests that the basic setup of the learning from interpretations setting is right, but one needs to work in a richer language. Viewed this way, learning in higher-order logic can be understood as taking the natural next step in the direction suggested by the learning from interpretations formulation. This is of course only a retrospective viewpoint; the two formulations were developed very much independently.

## References

1. M. Anthony. Decision lists and threshold decision lists. Technical Report LSE-CDAM-2002-11, London School of Economics, 2002.
2. M. Anthony and P. L. Bartlett. *Neural Network Learning: Theoretical Foundations.* Cambridge University Press, 1999.
3. A. F. Bowers, C. Giraud-Carrier, and J. W. Lloyd. A knowledge representation framework for inductive learning. `http://rsise.anu.edu.au/~jwl/`, 2001.
4. W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68(2):303–366, 1994.
5. W. W. Cohen and D. Page. Polynomial learnability and inductive logic programming: Methods and results. *New Generation Computing*, 13:369–409, 1995.
6. L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.
7. L. De Raedt and S. Džeroski. First order *jk*-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.
8. M. Kearns and Y. Mansour. On the boosting ability of top-down decision tree learning algorithms. *J. of Computer and System Sciences*, 58(1):109–128, 1999.
9. J.-U. Kietz and S. Džeroski. Inductive logic programming and learnability. *SIGART Bulletin*, 5(1):22–32, 1994.
10. S. Kramer, N. Lavrač, and P. Flach. Propositionalization approaches to relational data mining. In *Relational Data Mining*, chapter 11. Springer, 2001.
11. J. W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3, 1999.
12. J. W. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data.* Cognitive Technologies. Springer, 2003.
13. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.
14. K. S. Ng. Generalization behaviour of Alkemic decision trees. In *Proc. of the 15th International Conference on Inductive Logic Programming*, pages 246–263, 2005.
15. K. S. Ng. *Learning Comprehensible Theories from Structured Data.* PhD thesis, Computer Sciences Laboratory, The Australian National University, 2005.
16. R. L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.