(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. Positive results are usually obtained by analyzing concrete algorithms for variously restricted classes of logic programs. Negative results, in turn, are usually shown by reducing known difficult problems like 3-SAT and the PAC-predictability of boolean formulae in disjunctive normal form to different aspects of learning syntactically restricted logic programs. See, for a survey, [23], [12], [18], [7], [8], [15] and [3]. All these analyses have a strong syntactic flavour to them, and the arguments are usually intimately and intricately linked to the computational 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 that uses resolution theorem proving for computation, 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 [21]. 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 [21].

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 Lloyd's logical 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 [20]. In what follows, I will assume the reader is familiar with the syntax and terminology of functional programming languages. This, I hope, is not an unreasonable assumption since functional programming is a recommended core subject in every computer science curriculum.

2.1 Extending Attribute-Value Learning

We consider only binary classification problems in this paper. (This is not a restriction because other forms of classification learning can be *reduced* to binary classification learning [4].) The basic learning problem is simple to state: Given a set $\{(x_i, y_i)\}_{i=1}^n, x_i \in X,\ y_i \in \{\top, \bot\}$ of training examples generated from some unknown distribution D on $X \times \{\top, \bot\}$ and a class H of predicates (boolean functions) on X, find $h \in H$ such that the error of h with respect to D is minimized. In standard attribute-value learning, X is a subset of $\mathbb{R}^m$ for some m. The logical setting introduced in [21] 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. The class of basic terms includes $\mathbb{R}^m$ and just about every data type in common use in computer science.
2. The set H is allowed to be any subset of computable predicates on basic terms definable by composing simpler functions called transformations.

We now examine these two points in some detail. References to relevant ILP literature will be made when appropriate. Unfortunately, a detailed comparison between learning in higher-order logic and ILP cannot be made here due to space constraints. For that, I refer the reader to [5, §9] and [25, Chap. 7].

2.2 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 [10] and learning from propositionalized data [19] 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. We now do this informally. The details are in [21].

We have a set of variables and a set of constants with signatures (declared types). Terms are defined inductively in the usual way. Each variable is a term. Each constant is a term. If s and t are terms having appropriate types, then $(s\ t)$ is a term. If x is a variable and t is a term, then $\lambda x.t$ is a term.Finally, if $t_1, t_2, \ldots, t_n$ are terms, then $(t_1, t_2, \ldots, t_n)$ is a term.

There are two kinds of constants: functions and data constructors. Functions have definitions; data constructors do not. For example, constants like $1, 2, 3, \ldots$ and the list constructors # and [] are data constructors. They are used to represent individuals. To arrive at the set of basic terms, we restrict ourselves to use only data constructors in the formation of terms.

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 [5] and [25]. 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 now see how this simple problem can be modelled.

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

$$type\ Bunch = \{Key\}$$
$$type\ Key = Make \times Length \times Width$$

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

$$Abloy, Chubb, Rubo, Yale : Make$$
$$Short, Medium, Long : Length$$
$$Narrow, Normal, Broad : Width.$$

Listed here are some training examples.

$$opens\ \{(Abloy, Medium, Broad), (Chubb, Long, Narrow)\} = \top$$
$$opens\ \{(Yale, Short, Narrow), (Yale, Long, Normal),$$
$$(Chubb, Short, Broad), (Chubb, Medium, Broad)\} = \bot$$

Given such a set, we want to learn the following function. $opens : Bunch \to \Omega$. Here and in the following, Ω denotes the type of the booleans.

Given such data, one can proceed with distance-based learning methods by plugging into standard learning algorithms (like support vector machines and k-means) suitable kernels and distance measures defined on basic terms; see [14]. 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.

2.3 Predicate Construction

Predicates are constructed incrementally by composing more 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*

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

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

Every function is potentially a tranformation — 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 Ω, 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 $top : a \to \Omega$ defined by $(top\ x) = \top$ for each x. is the weakest predicate on the type a one can define.

Example 4. Given a constant like $Abloy : Make$, we can define

$$(= Abloy) : Make \to \Omega$$
$$((= Abloy)\ x)\ =\ (x = Abloy)$$

which evaluates an x to $\top$ iff x is $Abloy$. In a similar way, we can define $(= C)$ for every other constant C we introduced earlier in §2.2.

Example 5. Given terms of type Key, which are tuples, we can define

$$projMake : Key \to Make$$
$$(projMake\ (t_1, t_2, t_3)) = t_1$$

to project out the first element. We can define *projLength* and *projWidth* likewise.

Example 6. Given terms of type Key, we can define the following transformation

$$\wedge_2 : (Key \to \Omega) \to (Key \to \Omega) \to Key \to \Omega$$
$$(\wedge_2\ p\ q) = \lambda x.((p\ x)\ \wedge\ (q\ x))$$

to conjoin predicates on Key.

Example 7. Given terms of type $Bunch$, which are sets of keys, we can define

$$setExists_1 : (Key \to \Omega) \to Bunch \to \Omega$$
$$(setExists_1\ p\ q) = \exists x.((x \in q)\ \wedge\ (p\ x)).$$

The term $(setExists_1\ p\ q)$ evaluates to $\top$ iff there exists a key in q that satisfies p. This transformation directly capture the notion of multiple-instance learning.

Example 8. Given the transformations identified so far, we can construct (complex) standard predicates on $Bunch$. For example, the predicate

$$setExists_1\ (\wedge_2\ (projMake \circ (= Rubo))\ (projLength \circ (= Medium)))$$

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 [6]. 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.

$$top \rightarrowtail setExists_1(\wedge_2\ top\ top)$$
$$top \rightarrowtail projMake \circ (= C) \quad \text{for each constant } C : Make$$
$$top \rightarrowtail projLength \circ (= C) \quad \text{for each constant } C : Length$$
$$top \rightarrowtail projWidth \circ (= C) \quad \text{for each constant } C : Width$$

One should think of a predicate rewrite system as a kind of grammar for generating a class of predicates. Roughly speaking, this 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 predicate rewrite system given above.

$$top \rightsquigarrow setExists_1(\wedge_2\ top\ top) \rightsquigarrow setExists_1(\wedge_2\ (projMake \circ (= Abloy)\ top)$$
$$\rightsquigarrow setExists_1(\wedge_2\ (projMake \circ (= Abloy)\ (projWidth \circ (= Broad)))$$

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 [25] 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 [26] 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 will examine the PAC learnability of k-DL($\rightarrowtail$) and k-DT($\rightarrowtail$) in this section, focussing mainly on the former. We start by reviewing the basic concepts of PAC learning in §3.1. The efficient computability of higher-order predicates – a prerequisite for efficient PAC learnability – is studied in §3.2. Following this, we take a short detour into information theory by looking at the sample complexity of learning higher-order concepts in §3.3. All that preparation will then allow us to state some results on the efficient (agnostic) PAC learnability of k-DL($\rightarrowtail$) and k-DT($\rightarrowtail$) in §3.4 and §3.5.

3.1 PAC Learning

Let X be the set of individuals and H a set of predicates over X. The class H is said to be *PAC learnable* if there exists a learning algorithm L satisfying the following: given any $\epsilon, \delta \in (0, 1)$, there is an integer $m(\epsilon, \delta)$ such that for all $m \geqslant m(\epsilon, \delta)$, for any $t \in H$ and any probability distribution μ on X, with probability at least $1-\delta$, given a sample of size m drawn independently according to μ and labelled with t, the error of the hypothesis $h \in H$ output by L with respect to t and μ defined by $er_\mu(h, t) = \mu\{x \in X : h(x) \neq (t\ x)\}$ is less than ϵ. The number $m(\epsilon, \delta)$ is called the sample complexity of learning H.

The class H is said to be *efficiently PAC learnable* if, in addition to the above, L runs in time polynomial in m, $1/\epsilon$, $1/\delta$, the encoding length of instances in X, and the encoding length of the target function $size(t)$.

In the agnostic PAC setting, we do not presuppose the existence of a target function but assume that examples are generated independently according to an unknown probability distribution P on $X \times \{\top, \bot\}$. The aim of learning is similar: find a function in H that is arbitrarily close to the function $t^* \in H$ that has the smallest true error with respect to P.

3.2 Efficiently Computable Predicates

As pointed out in [9], 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.

In the following, the concept of an algorithm is assumed to be realized by some standard model of computation like Turing machines. Also assumed is an appropriate encoding scheme for the set of individuals X, together with a function for computing the size of the encoding of every $x \in X$, denoted $|x|$. (This is not hard to formalize. We can associate unit size to basic data types like integers, floating-point numbers, characters, etc. The encoding size of more complex individuals with type α can then be defined by induction on the structure of α. For example, the encoding size of a tuple $x = (t_1, \ldots, t_n) : \alpha_i \times \cdots \times \alpha_i$ can be defined by $|x| = \sum_i |t_i|$. Also, the encoding size of a set $x : \alpha \to \Omega$ can be defined by $|x| = n \cdot \max_{x_i \in x} |x_i|$ where n is the cardinality of x. Etc.)

Definition 9. *A function $r : \alpha \to \sigma$ is said to be* polynomial-time computable *if there exists an algorithm A that computes r and a polynomial $p(n)$ such that the number of steps required by A to compute $(r\, x)$ for any x is at most $p(|x|)$.*

The following is a standard result in computability theory.

Proposition 10. *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.*

Definition 11. *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 12. *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 is by induction on the number of transformations in p. Suppose the result holds for standard predicates that have $< m$ transformations and p has m transformations. Now p has the form $(f_1\ p_{1,1} \ldots p_{1,k_1}) \circ \cdots \circ (f_n\ p_{n,1} \ldots p_{n,k_n})$. By the inductive hypothesis, each $p_{i,j}$ is polynomial-time computable. Since each f_i is polynomial-time computable qua transformation, it follows that each

$f_i\ p_{i,1} \ldots p_{i,k_i}$, $1 \leqslant i \leqslant n$, is polynomial-time computable. By Proposition 10, compositions of polynomial-time computable functions yield another polynomial-time computable function. □

Note: The reader should not confuse polynomially computable predicates with big polynomial constants and non-polynomially computable predicates.

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.

3.3 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 [24]. The main conclusion reached in that paper is that in the rich higher-order setting we work in, the VC dimension of predicate classes is usually not too much lower than the simple upper bound given by the logarithm of the size of the predicate class.

To illustrate, consider the predicate rewrite system given earlier for the Keys problem. Since $|S_{\rightarrowtail}| = 34$, we have $VCD(S_{\rightarrowtail}) \leq \log_2 \lfloor |S_{\rightarrowtail}| \rfloor = 5$. In [24], we can find tools for obtaining lower bounds. A typical result from [24] is the following.

Proposition 13. *Let X be a set and suppose $\mathcal{F}$ is a class of predicates over X. Let $\mathcal{G} = \{g_f : f \in \mathcal{F}\}$ where each $g_f : 2^X \rightarrow \{\top, \bot\}$ is defined by*

$$g_f(t) = \begin{cases} \top & \text{if } \exists x \in t.\ f(x) = \top \\ \bot & \text{otherwise.} \end{cases}$$

If there exists a finite $S \subseteq X$ s.t. $|S| \geqslant 2$ and for all $x \in S$, there exists an $f \in \mathcal{F}$ s.t. $f(x) = \top$ and $f(y) = \bot$ for all $y \in S \setminus \{x\}$, then $VCD(\mathcal{G}) \geq \lfloor \log_2 |S| \rfloor$.

Using Proposition 13, it is easy to establish that $VCD(S_{\rightarrowtail}) \geq 4$. Other examples showing similar matching upper and lower bounds can be found in [24].

Given the observations of [24], we consider only finite function classes in this paper. A predicate 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 in this paper.

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

We look at the learnability of higher-order decision lists in this section, starting with the PAC learnability of k-DL($\rightarrowtail$). Under reasonable assumptions on the encoding sizes of individuals and the target function t ($size(t) = \log_2 |k\text{-DL}(\rightarrowtail)|$), it is easy to show that the following is true; Rivest's proof [26] goes through essentially unchanged.

Proposition 14. *Let X be a set of individuals and $\rightarrowtail$ a finite predicate rewrite system made up of only transformations that satisfy Definition 11. 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 14 was 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. (Indeed, taken together the observations of [26], [13] and [2, Chap. 24], one suspects k-DL($\rightarrowtail$) is one of the largest classes, if not the largest class, that can be 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. This demonstrates that k-DL($\rightarrowtail$) is not efficiently agnostic PAC learnable. We now give a proof.

The argument is an adaptation of the proof for [2, Thm 24.2]. We associate true with 1 and false with 0 in the following for convenience. 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 between 1 and 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 [26].

It is known that Vertex-Cover is NP-hard. We now 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 15. Consider the graph $G = \{\{1, 2, 3, 4\}, \{11, 12, 13, 14, 23, 33\}\}$. Then

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

Proposition 16. *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 $h \in$ 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.*

Proof. ($\rightarrow$) Suppose there is such an $h \in$ 1-DL(n). From [1], we know that 1-DL$(n) \subseteq$ LT(n), where LT(n) is the set of all linear threshold functions over $\{0, 1\}^n$. Thus there exists an $h' \in$ LT(n) that is equivalent to h. We represent h' by its weight vector $w = (w_1, w_2, \ldots, w_n, b)$. We now construct a subset U of V as follows.

1. For each $(e_{i,i}, y) \in z(G)$, if $h'(e_{i,i}) = 0$, then include i in U.
2. For each $(e_{i,j}, 0) \in z(G)$, $i \neq j$, if $h'(e_{i,j}) = 1$, then include one of i, j in U.

The set U so-constructed contains at most k vertices since

$$\hat{er}(h', z(G)) = \hat{er}(h, z(G)) \leqslant k/(n+r).$$

We now show that U is a vertex cover for G. Consider an arbitrary edge ij in E. If either $h'(e_{i,i}) = 0$ or $h'(e_{j,j}) = 0$ then we're done. Suppose not, that is, $h'(e_{i,i}) = h'(e_{j,j}) = 1$. Then we may deduce that $w_i \geqslant b$ and $w_j \geqslant b$. This implies that $h'(e_{i,j}) = 1$. Because of the way U is constructed, it follows that at least one of the vertices i, j is in U. Since ij is an arbitrary edge, we conclude that U is indeed a vertex cover.

($\leftarrow$) Suppose $U = \{i_1, \ldots, i_{|U|}\} \subseteq V$ is a vertex cover of G and $|U| \leqslant k$. Using the syntax of [26], we define $h \in$ 1-DL(n) to be

$$h = (x_{i_1}, 0), \ldots, (x_{i_{|U|}}, 0), (\mathbf{true}, 1).$$

We claim that $\hat{er}(h, z(G)) \leqslant k/(n+r)$. Observe that if $ij \in E$, then since U is a vertex cover, one of i, j belongs to U and thus $h(e_{i,j}) = 0$. This means that all the examples in $z(G)$ arising from the edges of G are correctly classified. Consider now examples in $z(G)$ of the form $(e_{i,i}, 1)$. We have $h(e_{i,i}) = 0$ if $i \in U$ and $h(e_{i,i}) = 1$ otherwise. It follows that

$$\hat{er}(h, z(G)) = \frac{|U|}{n+r} \leqslant \frac{k}{n+r}. \qquad \square$$

Given that $z(G)$ can be computed from G in time polynomial in the size of G, we have established the fact that DL-FIT is NP-hard. If there is an algorithm that can find in polynomial time $\arg\min_{h \in \text{1-}DL(n)} \hat{er}(h, \mathcal{E})$ given a set $\mathcal{E}$ of examples, 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 assumptions.

3.5 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 $\rightarrowtail$ defined on X, 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 over $S_{\rightarrowtail}$. 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. There are hardness results on the problem of computing the *smallest* decision trees given training examples; see [16]. 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 [17].

4 Discussion and Conclusion

We conclude by comparing our analysis with similar studies in ILP. I assume the reader is familiar with logical settings in ILP [10], as that is needed to appreciate the conclusion I give here.

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 [22] and learning from interpretations [11]. 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 [10]. 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 by introducing into the hypothesis language 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.

References

1. M. Anthony. Decision lists and threshold decision lists. Technical Report LSE-CDAM-2002-11, Department of Mathematics and Centre for Discrete and Applicable Mathematics, London School of Economics, 2002.

2. M. Anthony and P. L. Bartlett. *Neural Network Learning: Theoretical Foundations*. Cambridge University Press, 1999.
3. M. Arias and R. Khardon. Learning closed horn expressions. *Information and Computation*, 178:214–240, 2002.
4. A. Beygelzimer, V. Dani, T. Hayes, J. Langford, and B. Zadrozny. Reductions between classification tasks. *Electronic Colloquium on Computational Complexity*, TR04-077, 2004.
5. A. F. Bowers, C. Giraud-Carrier, and J. W. Lloyd. A knowledge representation framework for inductive learning. `http://rsise.anu.edu.au/~jwl/`, 2001.
6. W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68(2):303–366, 1994.
7. W. W. Cohen. PAC-learning recursive logic programs: Efficient algorithms. *Journal of Artificial Intelligence Research*, 2:501–539, 1995.
8. W. W. Cohen. PAC-learning recursive logic programs: Negative results. *Journal of Artificial Intelligence Research*, 2:541–573, 1995.
9. W. W. Cohen and D. Page. Polynomial learnability and inductive logic programming: Methods and results. *New Generation Computing*, 13:369–409, 1995.
10. L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95:187–201, 1997.
11. L. De Raedt and S. Džeroski. First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.
12. S. Džeroski, S. Muggleton, and S. Russell. PAC-learnability of determinate logic programs. In *Proc. of the Workshop on Computational Learning Theory*, 1992.
13. T. Eiter, T. Ibaraki, and K. Makino. Decision lists and related boolean functions. *Theoretical Computer Science*, 270(1-2):493–524, 2002.
14. T. Gärtner, J. W. Lloyd, and P. A. Flach. Kernels and distances for structured data. *Machine Learning*, 57:205–232, 2004.
15. T. Horváth and T. György. Learning logic programs with structured background knowledge. *Artificial Intelligence*, 128(1-2):31–97, 2001.
16. L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, 1976.
17. 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.
18. J.-U. Kietz and S. Džeroski. Inductive logic programming and learnability. *SIGART Bulletin*, 5(1):22–32, 1994.
19. S. Kramer, N. Lavrač, and P. Flach. Propositionalization approaches to relational data mining. In *Relational Data Mining*, chapter 11. Springer, 2001.
20. J. W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 3, 1999.
21. J. W. Lloyd. *Logic for Learning: Learning Comprehensible Theories from Structured Data*. Cognitive Technologies. Springer, 2003.
22. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.
23. S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive Logic Programming*, pages 281–298. Academic Press, 1992.
24. K. S. Ng. Generalization behaviour of Alkemic decision trees. In S. Kramer and B. Pfahringer, editors, *Proceedings of the 15th International Conference on Inductive Logic Programming*, pages 246–263, 2005.
25. K. S. Ng. *Learning Comprehensible Theories from Structured Data*. PhD thesis, Computer Sciences Laboratory, The Australian National University, 2005.
26. R. L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.