A Framework for Higher-Order Inductive Machine Learning

A.F. Bowers, C. Giraud-Carrier, C. Kennedy, J.W. Lloyd, R. MacKinney-Romero
Department of Computer Science
University of Bristol
Bristol BS8 1UB, UK

Abstract

This position paper presents a framework for inductive machine learning which includes higher-order concepts and is sufficiently general to include most of the extant (symbolic) inductive learning frameworks and systems.

1 Introduction

Inductive machine learning focuses on techniques for learning from examples and background knowledge. Traditionally, inductive learners have used the attribute-value language to represent examples. Though the relative simplicity of this representation allows the construction of efficient algorithms, it also restricts the applicability of the learner, as induced theories are essentially propositional. In recent years, research in inductive logic programming (ILP) has extended the applicability of learning systems by investigating and successfully designing efficient methods of induction within first-order predicate logic. Most implemented ILP systems are based on Prolog and hence can take advantage of Prolog's well-defined, efficient operational model. However, they also inherit Prolog's limitations. In particular, Prolog is essentially first-order and provides neither higher-order constructs nor declarative meta-programming facilities.

Escher is a new language which effectively combines the best features of logic and functional programming languages. Escher's programs are equational theories and Escher's computational model is based on rewriting. Hence, Escher offers a unique opportunity to broaden the definition of ILP and to investigate induction in higher-order logics. What follows is thus a framework for (symbolic) inductive learning which is sufficiently general to include most of the extant (symbolic) inductive learning frameworks and systems. It treats the logic programming and the functional programming cases uniformly. Furthermore, it generalises existing frameworks in that higher-order concepts are included. Thus higher-order functions (and, in particular, predicates) can appear in the background theory, and the target function whose definition is to be induced can be higher order. We believe the extensions considered here will prove to be significant for a growing class of applications of inductive learning.

Section 2 gives a brief overview of Escher, with emphasis on the constructs useful for induction. Section 3 reformulates the inductive learning problem in our extended context and Section 4 presents the form taken by induced functions. Section 5 outlines a general learning algorithm. Finally, Section 6 concludes the position paper with some remarks.

2 Escher

We work in Escher logic which is the logic of the Escher programming language. This is a higher-order logic based on Church's simple theory of types [Chu40] with several extensions.

The following constructs of Escher logic are particularly relevant to the construction of the definition of the target function in an inductive learning problem:

- connectives $\wedge$, $\vee$, $\rightarrow$, and $\neg$.

- Atoms of the form $x = f(x_1, ..., x_n)$, where x is a variable, f is a function, and the x_i are variables.
- Atoms of the form $p(x_1, \ldots, x_n)$, where p is a predicate and the x_i are variables.
- existential quantification $\exists x.t$, where t is a conjunction of atoms.
- universal quantification $\forall x.(s \rightarrow t)$, where s and t are formulas.
- set abstraction $\{x : t\}$.
- applications of the form $s \in \{x : t\}$.
- lambda abstraction $\lambda x.t$.
- applications of the form $(\lambda x.t\ s)$.

Most existing ILP systems confine attention to formulas consisting of atoms and the single connective conjunction. A few consider negation. Existential quantification only appears implicitly in the existing logic programming frameworks. The other constructs are almost always ignored.

Escher is an integrated functional and logic programming language with a syntax that coincides with Haskell for the functional subset, but also includes syntactic extensions for the quantifiers and set constructs. Escher programs are equational theories in which individual statements have the form
$f(t_1, \ldots, t_n) = b$
where f is a function, the t_i are terms, and b is a term. The term $f(t_1, \ldots, t_n)$ is called the *head* and the term b is called the *body* of the statement. The *definition* of a function f consists of all the statements which have f as the top-level function in their head.

We say a closed term is a *value* if it consists solely of data constructors. For example, the list $[1, 2, 3]$, the boolean *True*, and the integer 42 are all values.

Further details about Escher logic and the Escher programming language are contained in [Llo95] and [Llo97].

3 Formulation of the Problem

An inductive learning problem has the following components:

1. *Background theory.*
   The background theory consists of the knowledge available to the learning algorithm. It is a (consistent) Escher program.

2. *Signature of the target function.*
   The target function $f : \sigma \rightarrow \tau$ is the function whose definition has to be learned from examples. The symbol f is assumed not to be in the alphabet of the background theory. The type σ can be arbitrarily complex; for example, it can be higher-order. However, τ typically corresponds to a very simple domain consisting of just a few elements. Very often, τ is the type o (the booleans).

3. *Examples.*
   These have the form $f(s) = t$, where s and t are closed terms. Typically, s is a tuple of values and t is a value.

4. *Pragma.*
   These are hints and/or directives to the learning algorithm as to the form of the definition to be learned. For example, one may specify that certain arguments of a predicate are input or output. Similarly, one may specify the number of occurrences of a predicate (including the connectives) that may appear in the induced definition or restrict the complexity (using some measure) of the induced definition.

Let us say a definition for the target function f *covers* an example $f(s) = t$ if the term $f(s) = t$ is a logical consequence of the theory consisting of the background theory together with the definition for f. Then, given the above components, the *inductive learning problem* is as follows:

Find a definition for the target function which covers all the examples and satisfies the pragma.

The inductive learning problem is intrinsically a heuristic search problem. There are potentially many possible definitions for the target function and the space consisting of these definitions must be searched. The major research issues are concerned with finding good heuristics to reduce the amount of search required to a minimum.

We conclude this section with a remark concerning the issue of learning from positive data only. Due to some artefacts of Prolog (e.g., closed-world assumption, negation as failure), learning tasks have traditionally been split into positive and negative examples. It is our contention that such a separation is unnatural. Functions, including predicates, simply map inputs to outputs, irrespective of the value of that output. Hence, induction in our framework learns the characterisation of one output class at a time, whilst all other classes are used as counter-examples for purposes of constraining the search for a solution.

4 Form of the Induced Definition

We now explain the form that the induced definition of the target function takes.

An important decision which has to be made concerns the number of statements which should appear in the definition. Suppose $f : \sigma \to \tau$, where σ is a product $\sigma_1 \times \ldots \times \sigma_m (m \geq 1)$. Then one choice is that the definition should consist of exactly one statement and the head of the statement should be of the form
$f(x_1, \ldots, x_m)$
where $x_1, \ldots, x_m$ are distinct variables. Outside applications to program synthesis, a (non-recursive) definition containing a single statement with a most general head is the most common form of induced definition.

For program synthesis, induced definitions usually have more than one statement (and they are usually recursive). The reason is that functions operate on data types with several data constructors and it is natural to structure definitions so that there is a separate statement for each data constructor. For example, here are the Escher definitions for the `member`, `concat`, and `split` functions.

```
member :: (a, [a]) -> Bool;
member(x, []) =
    False;
member(x, y : z) =
    (x == y) || member(x, z);

concat :: ([a], [a]) -> [a];
concat([], x) =
    x;
concat(x : y, z) =
    x : concat(y, z);

split :: ([a], [a], [a]) -> Bool;
split([], x, y) =
    x == [] && y == [];
split(x : y, v, w) =
    (v == [] && w == x : y) || exists [z] (v == x : z && split(y, z, w));
```

The pattern to these definitions, which is typical of Escher code, is that one argument of the function is chosen to distinguish the cases given by the different data constructors. In the case

of lists, there are two data constructors, `[]` for the empty list and `:` for non-empty lists. For each statement, the distinguished argument contains a term which has the corresponding data constructor at the top level with distinct variables as arguments, while all other arguments in the head of the statement are distinct variables. Another important aspect of the above definitions is that the recursive statement corresponds to the `:` data constructor and the base statement corresponds to the `[]` data constructor. (By the way, this is *not* true of the usual definition of the `member` predicate in Prolog!)

The algorithm induces definitions in this form by first choosing the distinguished argument and then partitioning the examples according to the constructor appearing in them in this argument. Each statement is then learned independently, except for the contribution previously learned statements make when checking examples. As an example, suppose we want to learn the `member` function above and the examples are

```
member(2, []) = False;
member(1, [1,2]) = True;
member(1, [2,3,4]) = False;
member(4, []) = False;
member(2, [1,2,3]) = True;
```

Then the algorithm partitions the examples according to whether the second argument has a `[]` or `:` at the top level. Thus the examples

```
member(2, []) = False;
member(4, []) = False;
```

are used to learn the base statement

```
member(x, []) = False;
```

and the examples

```
member(1, [1,2]) = True;
member(1, [2,3,4]) = False;
member(2, [1,2,3]) = True;
```

are used to learn the recursive statement in the form

```
member(x, y : z) = if x == y then True else if member(x, z) then True else False;
```

from which the simpler form in the definition of `member` can easily be obtained. When checking examples while learning the recursive statement, the base statement is available for use and, in fact, is essential.

Having discussed the form the head of a statement can take, we now turn to the body. Consider a target function $f : \sigma \to \tau$. Suppose the head of the statement to be learned has the form $f(r)$, where r is a term of the form described above, and that the examples have been partitioned appropriately. We consider one such head and class of examples in what follows.

We assume that there are finitely many values of type τ. First we choose one of these values as the *default* value. A good choice for the default could be a value which does not appear on the right hand side of any of the examples, if such a value exists. Otherwise, one of the values is chosen arbitrarily. When τ is o, the usual choice for the default value is *False*. In what follows, we denote the default value by t_0. The role of the default value will become clear shortly.

The learning algorithm below constructs formulas $E_1, \ldots, E_n$, where each E_i is a formula whose free variables all appear in the head $f(r)$. The alphabet used to construct each E_i is that of the background theory, possibly together with the symbol f. If f is included, a recursive definition is possible; otherwise, the definition of f will be non-recursive. Then the induced statement for f has the form:

$f(r) =$

if E_1 then t_1 else

if E_2 then t_2 else
$\vdots$
if E_n then t_n else t_0.

where $t_1, \ldots, t_n$ are (not necessarily distinct) non-default values. The induced definition for f consists of all such statements for all the possibilities for r.

The intention is that each formula E_i in an induced statement "includes" some of the examples (in the corresponding class of examples) which map to t_i. Also E_i should "exclude" all the examples which do not map to t_i. Furthermore, the E_i together should "include" all the examples which do not map to t_0. The role of the default value t_0 is simply to provide a function value for all points not already covered by some $E_1, \ldots, E_n$. Altogether, the E_i provide a general, intensional classification of the examples.

In some applications, there may be insufficient examples to exercise all the values in the codomain of f, that is, there may be a (non-default) t_i with no examples mapping to t_i. In this case, we can simply define E_i to be *False* which has the same effect as omitting the corresponding component
if E_i then t_i else
from the induced statement.

To connect our approach with the usual inductive logic programming approach, consider the special case of learning a target predicate $p : \sigma \to o$, where $False$ is the default value. Then an induced statement takes the form

$p(r)$ = if E_1 then *True* else ... if E_n then *True* else *False*

which simplifies to

$p(r) = E_1 \vee \ldots \vee E_n$.

Each E_i should include some of the examples which map to *True* and should exclude all the examples which map to *False*. The E_i together should include all the examples which map to *True*. We can write this statement in logic programming form as

$p(r) \leftarrow E_1$
$p(r) \leftarrow E_2$
$\vdots$
$p(r) \leftarrow E_n$

from which we can recover the Escher form by using the Closed World Assumption to turn the implication back to an equality.

Note that one can equally choose *True* as the default value. In this case, the induced statement takes the form

$p(r)$ = if E_1 then *False* else ... if E_n then *False* else *True*

which simplifies to

$p(r) = \neg E_1 \wedge \ldots \wedge \neg E_n$.

5 Learning Algorithm

In this section, we discuss the general form of the learning algorithm.

As already pointed out, learning is intrinsically a search problem for which the search space is often huge. For example, when learning a statement $p(r) = E$ for a predicate $p : \sigma \to o$, the algorithm has to search through a large number of possible choices for E to find one which includes the examples mapping to *True* and excludes the examples mapping to *False* (and satisfies the pragma). This problem is already difficult for the usual case when E is a conjunction of atoms and yet we are proposing to add further syntax to the language, so it is likely that great care and ingenuity will be needed to make the task feasible.

The search for a suitable E is essentially a heuristic search problem for which the standard A* algorithm can be employed. We introduce a search space in which individual nodes correspond to formulas. The start node is the formula *True* and the goal nodes are any formulas satisfying the conditions stated above. Various operators take a formula and expand it to a number of successor formulas which then need to be investigated. The choice of which formula to expand next is guided by a heuristic function. The formula with the lowest heuristic value (indicating the highest promise) is chosen. The crucial component of the heuristic function is provided by the examples. The basic idea is that a formula shows more promise if it includes more of the examples it is supposed to include and excludes more of the examples it is supposed to exclude, and the heuristic function should reflect this.

An outline of the learning algorithm appears in Figure 1 below. At Bristol, we have begun to implement a learning system based on this algorithm. Currently, the system is able to learn simple definitions such as the one for `member` above from a few examples.

Input : background theory $\mathcal{B}$, signature $\sigma \to \tau$ of the target function f, and examples $\mathcal{E}$.
Output : definition D for f.

let D_0 be $\emptyset$;
choose a default value t_0 of type τ;
let V be the set of non-default values of type τ;
choose the arguments $r_1, \ldots, r_p$ $(p \geq 1)$ of the heads of the statements in D;
for $j = 1, \ldots, p$ do {
  let $\mathcal{R}_j$ be the set of examples in $\mathcal{E}$ corresponding to r_j;
  let $\mathcal{D}_j$ be the set of examples in $\mathcal{R}_j$ having the form $f(s) = t_0$, for some s;
  let $\mathcal{C}_j$ be $\emptyset$;
  let i be 0;
  while $\mathcal{R}_j \setminus \mathcal{D}_j \neq \emptyset$ do {
    let i be $i + 1$;
    find a $t_i \in V$, a formula E_i, and a non-empty set $\mathcal{N}_j^i \subseteq \mathcal{R}_j \setminus \mathcal{D}_j$ of examples having the form $f(s) = t_i$, for some s, such that
      (i) for all $f(s) = t \in \mathcal{C}_j \cup \mathcal{N}_j^i$, $f(s)$ reduces to t using the program $\mathcal{P}$, and
      (ii) for all $f(s) = t \in \mathcal{R}_j \setminus \mathcal{N}_j^i$, $f(s)$ reduces to t_0 using the program $\mathcal{P}$
    where $\mathcal{P}$ is $\mathcal{B} \cup D_{j-1} \cup \{f(r_j) =$ if E_1 then t_1 else ... if E_i then t_i else $t_0\}$;
    let $\mathcal{C}_j$ be $\mathcal{C}_j \cup \mathcal{N}_j^i$;
    let $\mathcal{R}_j$ be $\mathcal{R}_j \setminus \mathcal{N}_j^i$;
  }
  let D_j be $D_{j-1} \cup \{f(r_j) =$ if E_1 then t_1 else if E_2 then t_2 else ... if E_i then t_i else $t_0\}$;
}
let D be D_p;
return D;

Figure 1: Learning algorithm

It is straightforward to prove the following correctness theorem for the learning algorithm.

Theorem 1 *The definition D for the target function f produced by the learning algorithm covers all the examples in $\mathcal{E}$.*

An obvious source of computational overhead in the learning algorithm is the step where the examples are checked. Instead of checking that all the examples in $\mathcal{C}_j \cup \mathcal{N}_j^i$ are covered, it is natural to want to reduce this to checking just the new examples in $\mathcal{N}_j^i$. After all, the examples in $\mathcal{C}_j$ were checked in the previous iteration! The following example shows that, in general, we cannot avoid checking the examples in $\mathcal{C}_j$ again.

Consider the problem of learning a definition for a predicate p where the examples are

$p(A) = True$

$p(B) = True$
$p(C) = False.$

Choose *False* to be the default value. Suppose during the first iteration the statement

$p(x) =$ if $(x = A) \wedge \neg p(B)$ then *True* else *False*

is induced. This statement, although somewhat peculiar, does cover the first example. Now suppose, in trying to cover the second example (without rechecking the first example), the statement is expanded to

$p(x) =$ if $(x = A) \wedge \neg p(B)$ then *True* else if $x = B$ then *True* else *False*.

Now, unfortunately, the first example is no longer covered since $p(A)$ reduces to *False* using this statement.

This phenomenon of previously covered examples becoming uncovered when the statement being induced is expanded only occurs when there is both recursion and negation appearing in the statement. In fact, for non-recursive definitions, it is clear that one only needs to check the new examples in $\mathcal{N}_j^i$ and the theorem still holds. Similarly, if the definition being learned is a definite clause program, then the theorem also holds for the optimised version of the algorithm.

This example shows that once ILP systems start inducing recursive definitions containing negation, the correctness of the learning algorithm can no longer be taken for granted unless the algorithm explicitly does a great deal of checking.

6 Concluding Remarks

The proposed research addresses some of the main issues of ILP in a novel way. The use of Escher increases the chances of finding natural solutions to problems whose currently proposed solutions are made awkward or unnecessarily complex due to the limitations of first-order theories. Clearly, higher-order learning raises new problems. However, preliminary experiments show that, within reasonable constraints, useful results can be achieved. Our work will address the following issues, which are particularly relevant to current ILP research.

- *Predicate invention and utility.* It has been shown that, under certain circumstances, the invention of new predicates is necessary for successful learning. Yet the problems of predicate invention and utility are difficult to solve and remain open problems in ILP.

  Some proposals have been made in ILP's traditional first-order framework. However, invention and utility are second-order concepts. It is thus our contention that they are best addressed in their natural higher-order setting.

- *ILP systems as programming assistants.* One of the applications of ILP is program synthesis from partial specifications and examples. The more declarative a language, the easier it is to induce programs in that language. Indeed, it has been shown that non-declarative constructs, such as Prolog's cut, are difficult to learn. The strong declarativeness of Escher makes it a particularly suitable candidate for experiments in program synthesis.

References

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Llo95] J.W. Lloyd. Declarative programming in Escher. Technical Report CSTR-95-013, Department of Computer Science, University of Bristol, 1995. Also available at http://www.cs.bris.ac.uk/.

[Llo97] J.W. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1997. To appear.