Instance based function learning

Jan Ramon and Luc De Raedt

Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
{janr,lucdr}@cs.kuleuven.ac.be

Abstract. The principles of instance based function learning are presented. In IBFL one is given a set of positive examples of a functional predicate. These examples are true ground facts that illustrate the input output behaviour of the predicate. The purpose is then to predict the output of the predicate given a new input. Further assumptions are that there is no background theory and that the inputs and outputs of the predicate consist of structured terms. IBFL is a novel technique that addresses this problem and that combines ideas from instance based learning, first order distances and analogical or case based reasoning. We also argue that IBFL is especially useful when there is a need for handling complex and deeply nested terms. Though we present the technique in isolation, it might be more useful as a component of a larger system to deal e.g. with the logic, language and learning challenge.

1 Introduction

Within the field of machine learning, both instance based learning [1] and inductive logic programming (or relational learning) [10, 9] are important subfields. In instance based learning one tries to predict classes of examples by comparing these examples to other examples from a training set. Most often there is only a finite and small set of discrete classes, and the class can be regarded as a discrete attribute of the examples. Instance based learning (and concept-learning) can be regarded as function learning, where the function takes as input the example description and produces as output the value for the class attribute.

In recent work within the field of inductive logic programming classical propositional instance based learning techniques have been upgraded towards a first order framework. A prominent example of this approach is the RIBL system [2]. RIBL (and other approaches along this line) take as input a first order description of an example and predict the value for the corresponding class attribute. In terms of function learning, RIBL maps complex inputs (example descriptions in first order logic) to simple outputs (the value of the class attribute). From the viewpoint of function learning, one may wonder whether it would be possible to extend this framework to learn more complex functions that would map complex inputs onto complex outputs. Within the framework of computational logic, the inputs and outputs would then naturally correspond to deeply structured terms (cf. e.g. [4]). The technique of IBFL, presented in this paper, addresses precisely this problem. IBFL starts from a set of positive examples of a functional

predicate p . The examples are thus of the form $p(in, out)$ where in represents a possibly complex input term and out the corresponding output terms. IBFL is then given a new input terms in' for which it has to compute the corresponding output out' such that $p(in', out')$ holds. IBFL is not given any other information and thus operates with an empty background theory.

This IBFL setting extends the classical first order instance based learning framework. It also provides a framework for studying the issues involved in handling deeply structured terms, which is known to be one of the hard problems in inductive logic programming (cf. e.g. the Project Programme of the ESPRIT IV project on Inductive Logic Programming II). The hardness of this problem is illustrated by the fact that well-known ILP systems are unable to produce good results on the setting mentioned above due to 1) the combinatorics involved in handling deeply structured terms, and 2) the fact that the background theory in IBFL is empty, which makes it necessary to induce recursive hypotheses for defining p . Recursion (and program synthesis) is one of the other known hard problems in ILP. Despite the difficulties involved, many potential applications need to deal with structured terms. One such application is the recent Logic, Language and Learning challenge (cf. [8]) issued by Kazakov, Pulman and Muggleton. We hope that the IBFL framework and its techniques will contribute to a better understanding and possible solutions of these problems. IBFL will therefore also be illustrated on (simple) tasks that are relevant to the LLL challenge as well as to program synthesis.

This paper is organized as follows. In section 2, we review some important aspects of inductive logic programming and instance based learning. In section 3, we state the problem specification. In section 4, we present the base idea of instance based function learning. In section 5, we extend this method with a recursive component that allows to learn the translations of parts of terms. In section 6, we present some experiments, and finally, in section 7, we give some conclusions and possibilities for further work.

2 Preliminaries

A substitution $\theta = \{X_1/Y_1, \dots, X_n/Y_n\}$ is a set of elements X_i/Y_i such that X_i are variables and Y_i are terms. If t is a term and θ is a substitution, then we can apply θ to t : $t\theta$ is the term t in which each occurrence X_i is simultaneously replaced by Y_i . e.g. if $\theta = \{X/f(a), Y/X\}$ and $t = f(X, Y, g(X))$, then $t\theta = f(f(a), X, g(f(a)))$. If an inverse substitution θ^{-1} is applied to a term s , the result is the set of terms t such that $t\theta = s$. e.g. if $\theta = \{X/a, Y/f(a), Z/a\}$ and $t = g(f(f(a)), a)$, $t\theta^{-1} = \{g(f(f(X)), X), g(f(f(X)), Z), g(f(f(Z)), X), g(f(f(Z)), Z), g(f(Y), X), g(f(Y), Z)\}$. We will use the notion of least general generalization [12]: a term t_1 subsumes (or is more general than) a term t_2 iff there is a substitution θ such that $t_1\theta = t_2$. The subsumes relation induces a partial order on the set of terms. The least upper bound under subsumption is called the least general generalization (lgg).

We also use the notion of *position* as defined in [6]. *Positions* are sequences of positive integers (e.g. [2,3,2]). ϵ denotes the empty position, and $\cdot$ the concatenation operation on positions. With t a term or atom the sub-term of t at position u , t/u is defined as follows:

- If t is a term, then $t/\epsilon = t$.
- if $t = f(t_1, \dots, t_n)$, then $t/(i \cdot u) = t_i/u$.

In instance based learning a distance is needed between the examples. As we work with structured terms as examples, we must use a distance between terms such as the distances defined in [11], [7], [13], [14]. In this paper we use the simple distance defined in [11]:

Definition 1 (distance d_{nc} between terms). *If t_1 and t_2 are terms, then*

- if $t_1 = t_2$, $d_{nc}(t_1, t_2) = 0$.
- if $t_1 = p(x_1, \dots, x_n)$ and $t_2 = q(y_1, \dots, y_m)$ with $p \neq q$ or $n \neq m$, then $d_{nc}(t_1, t_2) = 1$.
- if $t_1 = p(x_1, \dots, x_n)$ and $t_2 = p(y_1, \dots, y_n)$, then $d_{nc}(t_1, t_2) = \frac{1}{2n} \sum_{i=1}^n d_{nc}(x_i, y_i)$.

e.g. we can compute $d_{nc}(p(f(a), g(h(c), d)), p(f(b), h(e, d))) = \frac{1}{4}(d_{nc}(f(a), f(b)) + d_{nc}(g(h(c), d), h(e, d))) = \frac{1}{4}(\frac{1}{2}d_{nc}(a, b) + 1) = \frac{1}{4}(\frac{1}{2} \cdot 1 + 1) = \frac{3}{8}$

3 Problem specification

In this section we state the problem we want to solve. The unknown target predicate represents a function. This means that the examples are ground facts about the input-output behaviour of the predicate. In terms of ILP we have the following problem setting:

Given is:

- an unknown functional target predicate p .
- a set of positive examples (ground facts) for p . In the following we will denote the training examples with $E_i = p(E_{i,in}, E_{i,out})$ with $E_{i,in}$ the input term and $E_{i,out}$ the output term.
- an empty background theory.
- an input term N_{in} .

Find:

- an unknown output N_{out} such that $p(N_{in}, N_{out})$.

Suppose e.g. we have an example $p(in(circle, square), in(square, circle))$, and have to find the output of $in(circle, triangle)$ (see figure 1). The program could then answer $in(triangle, circle)$. This is also an example of learning by analogy (cf. [3]).

4 Instance based function learning

In the attribute-value setting, the process of predicting a class for an example with the k -nearest neighbours method can be divided in three important steps. First, the example input is compared using a distance with all the training example inputs and the k nearest examples are selected. Second, from each of these k nearest examples the class information is extracted. Third, from these k classes the most frequent class is selected and used to predict the example.

Predicting a whole term at once is difficult. For this reason we will only predict one functor at a time. Algorithm 1 is the basic algorithm for this.

We start with a variable (a completely uninstantiated term) as predicted output and we repeatedly apply a substitution of the form $Var/f(Var_1, \dots, Var_n)$ to add functors (with variable arguments) until we obtain a ground term. In practice, since it is not always guaranteed that all positions will eventually become ground (by filling them with functors with arity 0 (constants)), a stopping criterion could be used. A good stopping criterion is restricting the maximum number of functors in the predicted output term.

Functors are added using the instance based learning component whose three components we discuss below. We know that algorithm 1 provides a partially instantiated term N_{out}^{tmp} and a position pos such that the sub-term N_{out}^{tmp}/pos at position pos of N_{out}^{tmp} is a variable. Thus the task of this instance based learning component is to (partially) instantiate N_{out}^{tmp}/pos , so we get a better approximation N_{out}^{better} of our prediction N_{out}^* of N_{out} .

First we need to measure the distance between the new example and the training examples. We can use for this the distance d_{nc} given in section 2.

After measuring the distances between the input of the new example and the inputs of all training examples, we can select the k nearest relevant training examples. Whether an example is relevant or not is determined by the possibility of the second step to extract useful information (functors) from that example, as will soon become clear.

In the second step, we must extract information about which functor to predict from the k nearest (relevant) examples. The way this is done is important as this causes the method to be able to transform the structure of terms.

Suppose we must extract information from training example E_i . First the least general generalization under subsumption $G_{in} = \text{lbg}(E_{i,in}, N_{in})$ of the input terms $E_{i,in}$ and N_{in} is computed, together with substitutions θ_e and θ_n such that $G_{in}\theta_e = E_{i,in}$ and $G_{in}\theta_n = N_{in}$ (see also figure 2). Then, we con-

Fig. 1. An example where IBFL could be used to learn by analogy

sider the set $S = E_{i,out}\theta_e^{-1}\theta_n$. Because of the inverse substitution this set can contain several elements. Let then S_1 be the subset of S of all terms t of S such that the functors along the path from the top functor to pos of t and N_{out} are identical. E.g. if $pos = [2, 1, 1]$ and $N_{out} = a(b, c(d(f, g), e))$ and $S = \{a(x, c(d(g, z), e)), a(b, q(d(h, g), e))\}$, then $S_1 = \{a(x, c(d(g, z), e))\}$ (see fig. 3).

Fig. 2. Step 2 of the instance based learning process

Let then $S_2 = \{t/pos | t \in S_1\}$. In the above example with $pos = [2, 1, 1]$, $S_1 = \{a(x, c(d(g, z), e))\}$, $S_2 = \{g\}$. Note that if t/pos is a variable (which will be possible after the extensions of section 5), this does not add functors to S_2 . If S_2 is not empty, then the example is relevant.

So for each relevant example a multi-set of functors is obtained. In the third step then the most frequent occurring functor is predicted.

We illustrate this with a small example. We suppose 1-nearest neighbours is used for simplicity.

Example 1. Suppose we have the following training examples:

`p(swap(a,b),swapped(b,a)).`
`p(dontswap(c,d),notswapped(c,d)).`

We want to know the output of $swap(g, h)$. First set the prediction $N_{out}^* = Var$. We first measure the distances $d_{nc}(swap(g, h), swap(a, b)) = \frac{1}{2}$ and $d_{nc}(swap(g, h),$

Algorithm 1 Base iteration

```

predict_term( $N_{in}$ )
  Let  $N_{out}^* = A\_New\_Variable$ ,
  while not(ground( $N_{out}^*$ )) and not(stoppingcriterion) do
    Let  $Pos$  be a position such that  $V = N_{out}^*/Pos$  is a variable,
    predict_best_functor_at_pos( $N_{in}, N_{out}^*, pos, Functor/Arity$ ),
     $V \leftarrow V\theta$  with  $\theta = \{V/Functor(Newvar_1, \dots, Newvar_{Arity})\}$ 
  return  $N_{out}^*$ .
```

Fig. 3. A and B have equal functors along the path to $[2, 1, 1]$

$dontswap(c, d) = 1$. So $p(\text{swap}(a, b), \text{swapped}(b, a))$ is the nearest example. Then the lgg is $G_{in} = \text{swap}(X, Y)$ and $\theta_e = \{X/a, Y/b\}$ and $\theta_n = \{X/g, Y/h\}$. Next, $\text{swapped}(b, a)\theta_e^{-1}\theta_n = \text{swapped}(Y, X)\theta_n = \text{swapped}(h, g)$. The main functor of N_{out} is predicted to be $\text{swapped}/2$ and we get a more instantiated prediction $N_{out}^* = \text{swapped}(_, _)$. This is not yet ground: In $\text{swapped}(_, _)$ positions $[1]$ and $[2]$ are still variables. For this select $[1]$ as the position to be instantiated and again select the nearest training example, recompute G_{in} , θ_e , θ_n which in this case all obtain the same result. Again $\text{swapped}(b, a)\theta_e^{-1}\theta_n = \text{swapped}(h, g)$, so we predict $N_{out}^*/[1] = h$ and $N_{out}^* = \text{swapped}(h, _)$. Iterating a third time we get $N_{out}^* = \text{swapped}(h, g)$. This is fully instantiated, so we predict $\text{swapped}(h, g)$ as the output term.

5 Recursive IBFL

The method described in the previous section works fine when the main problem is to transform structure. However, we also want to do more complex things. It can happen that too many examples become irrelevant as the predicted output term becomes more and more instantiated. Consider following training examples:

Example 2.

```
p(dog, chien).      p(house, maison).      p(near, pres).
p(cat, chat).       p(school, ecole).       p(in, dans).
p(place(near, school), lieu(pres, ecole)).
p(walks(dog, place(in, school)), court(lieu(dans, ecole), chien)).
p(walks(cat, place(near, school)), court(lieu(pres, ecole), chat)).
p(walks(cat, place(in, house)), court(lieu(dans, maison), chat)).
p(sleeps(cat, place(near, school)), dort(lieu(pres, ecole), chat)).
```

This example concerns a small translation problem. We would like to predict (i.e. translate) the output of $\text{sleeps}(\text{dog}, \text{place}(\text{in}, \text{house}))$. However, if we

apply the method from the previous section, at some point we get $N_{out}^* = dort(lieu(-, -), -)$ and $pos = [1, 2]$. Now there are no good and relevant examples: e.g. for $p(walks(dog, place(in, school)), court(lieu(dans, ecole), chien))$, we get $G = X$, $\theta_e = \{X/walks(dog, place(in, school))\}$, $\theta_n = \{X/sleeps(dog, place(in, house))\}$ and $court(lieu(dans, ecole), chien)\theta_e^{-1}\theta_n = court(lieu(dans, ecole), chien)$, the main functor is here *court* while the main functor of N_{out}^* is *dort*, so we can't use this to predict the first argument of *dort*. On the other hand, for the example $p(sleeps(cat, place(in, school)), dort(lieu(dans, ecole), chat))$, we have $G = sleeps(X, place(in, Y))$, $\theta_e = \{X/cat, Y/school\}$, $\theta_n = \{X/dog, Y/house\}$, and $dort(lieu(dans, ecole), chien)\theta_e^{-1}\theta_n = dort(lieu(dans, ecole), chien)$, and so *ecole* would be predicted. This is not what is intended as we can intuitively see that the result of the prediction should optimally be *dort(lieu(dans, maison), chien)*.

One solution to this problem is to allow the system to learn from the solution of subproblems (subterms). In the above example, this would mean that we let the system use the 'lexicon' that gives the translations for sub-terms.

To achieve this, we will, before using the instance based learning system of the previous section, add information to the input terms of the examples. An example is given in figure 4. This causes the substitutions that are essential in the prediction process to contain as much information as possible. The algorithm translates now all the subterms of the input term before translating the input term. The translation thus proceeds in a bottom-up fashion, starting at the leaves of the tree and gradually working on larger terms.

Fig. 4. Additional information for the input term to improve the substitutions

Example 3. What is the effect on example 2?

We can now translate $sleeps(dog, place(in, house))$: First, the input term $sleeps(dog, place(in, house))$ is extended with info on the sub-terms: we therefore first need to solve two other problems: Given input term *dog*, what is the output term? Given input term $place(in, house)$, what is the output term? We have a training example which gives the translation for *dog*. We translate $place(in, house)$ recursively: we have training examples for $p(in, dans)$ and $p(house, maison)$, so we get $p(place(in, house), lieu(dans, maison))$. So in our original problem, the extended input term is $(sleeps(dog, house), sleeps(\lambda(chien, dog), \lambda(lieu(dans, maison), place(\lambda(dans, in), \lambda(maison, house))))))$. Now the

instance based algorithm is applied. In the first step we get as preliminary prediction $dort(-, -)$. Suppose we now fill in the first argument. $p(sleeps(cat, place(in, house)), dort(lieu(dans, maison), chat))$ is a near and relevant example for this. We extend the input term of this training example. We get $(sleeps(cat, place(in, house)), sleeps(\lambda(chat, cat), \lambda(lieu(dans, ecole), place(\lambda(dans, in), \lambda(maison, house))))))$. We get as generalization of the input terms $G = (sleeps(X, house), sleeps(\lambda(V, X), \lambda(lieu(dans, maison), place(\lambda(dans, in), \lambda(maison, house))))))$ with $\theta_e = \{X/cat, V/chat\}$ and $\theta_n = \{X/dog, V/chien\}$. Finally, applying these same substitutions to $dort(lieu(dans, ecole), chat)$, we get $dort(lieu(dans, ecole), chat)\theta_e^{-1}\theta_n = dort(lieu(dans, W), V)\theta_n = dort(lieu(dans, maison), chien)$ which was the correct translation.

This kind of experiment is difficult for classical ILP systems such as Progol. We tried to run this example with Progol but ran into problems of too large search space due to the exponential number of terms (in the size of the examples) in the bottom clause¹.

6 Experiments

We implemented the method described in the previous sections. In this section, we summarize the result of some experiments.

Experiment 1 *Another application where IBFL could be useful is the learning of parsers. e.g. Given an expression such as $+(2, *(4, 5))$, we would like to be able to execute it (compute the result) on a stack-machine. Therefore we need a program such as $[push(2), push(4), push(5), *, +]$ which computes the expression. The IBFL system can learn to do this conversion. Given the following examples:*

```
p(3+(4*2), [push(3), [push(4), push(2), *], +]).
p(5*6, [push(5), push(6), *]).
p(8+9, [push(8), push(9), +]).
p(11, push(11)).
```

The system can correctly predict

```
p((11+12)*(13+14), [[push(11), push(12), +], [push(13), push(14), +], *]).
```

Then after removing the redundant brackets in the output we obtain the desired program. This is possible independently of the complexity of the expressions to convert, as long as sufficient training examples are given, including simple cases (from which the system can learn most unambiguously).

Experiment 2 *We want to learn the induction step of the predicate reverse (see algorithm 2). We can learn this from partial execution traces. The system was given a set of training examples:*

¹ James Cussens, personal communication


```

p(reverse([c,d,e],[b,a],Result),reverse([d,e],[c,b,a],Result)).
p(reverse([a,b],[ ],Result),reverse([b],[a],Result)).
p(reverse([ ],[z,y,x],Result),Result=[z,y,x]).
p(reverse([r,s,t,u,v],[q,p],Result),
    reverse([s,t,u,v],[r,q,p],Result)).

```

The p predicate has as first (input) argument the predicate to be executed. The second (output) argument represents what must be done for that. There are two kinds of execution steps: If the output term of p is a simple operation (a unification of the output term with the correct result), this can be done at once. This is the case if the first argument in the call of `reverse` is `[]`. If the output term of p is a new call to `reverse`, we can again execute this call by predicting the output of p when this new call is given as input, and so on.

The input of some other steps were given. The system predicted the following outputs correctly:

```

p(reverse([f,g,h,i],[e,d,c],Result),
    reverse([g,h,i],[f,e,d,c],Result)).
p(reverse([x,y,z],[ ],Result),reverse([y,z],[x],Result)).
p(reverse([ ],[z,y,x,w],Result),Result=[z,y,x,w]).

```

The following observations can be made: first, the system could correctly distinguish between the base case (where the first argument is `[]`) and other cases. Second, the only test examples whose output term was predicted incorrectly were the examples with a first argument at least as large as the largest first argument in the training examples. e.g.

```

p(reverse([r,s,t,u,v],[q],Result),reverse([s,t,u,[ ]],[r,q],Result)).

```

One can conclude from this that the system can learn the induction step of the `reverse` predicate, but that further work could make it possible to generalize to larger inputs.

When an induction step is learned, this induction step can be applied several times (i.e. the predicted output can be used as input of the next step). If we do this until the result does not contain any further call to `reverse`, we can compute for all inputs the result of the `reverse` predicate. This means we can learn the `reverse` predicate from partial execution traces..

7 Conclusions

We have presented an approach that extends instance based learning to the learning of complex functions from terms to terms, we have argued that IBFL is

Algorithm 2 Reverse

```

reverse([X|Y],Z,Result) :-
    reverse(Y,[X|Z],Result).
reverse([ ],Z,Result) :- Z=Result.

```

relevant to the problem of handling deeply structured terms and recursion, and we have illustrated the potential use of IBFL in natural language applications and program synthesis.

We expect that the approach can best be used as a component of a larger system that might use IBFL to compute e.g. the output structure of a clause in a program synthesis application. One current limitation of the technique is its inability to incorporate background knowledge. It may be possible to incorporate background knowledge in the process in similar ways as in RIBL and the framework by Flach et al [4]. This framework encodes examples (and background theory) as structured terms and may therefore be well-suited for IBFL.

IBFL is related to the instance based learning work in ILP (see [1]) it can be considered a form of analogical reasoning (see [3],[5],[15]). Indeed, in analogical and case based reasoning one maps a target problem onto a target solution by using similarities with a source problem with known solution. In IBFL the target problem is the new input term, and the source problems and solutions correspond to the examples of the input-output behaviour of the function.

8 Acknowledgements

Jan Ramon is supported by the Flemish Institute for the Promotion of Science and Technological Research in Industry (IWT). Luc De Raedt is supported by the Fund for Scientific Research (FWO), Flanders. This work is supported by the European community Esprit project no. 20237, Inductive Logic Programming 2. We also thank James Cussens for the many helpful mails about *progol*.

References

1. D. W. Aha, D. Kibler, and M. K. Albert. Instance-based learning algorithms. *Machine Learning*, 6(1):37–66, January 1991.
2. W. Emde and D. Wettschereck. Relational instance-based learning. In L. Saitta, editor, *Proceedings of the 13th International Conference on Machine Learning*, pages 122–130. Morgan Kaufmann, 1996.
3. Thomas G. Evans. A program for the solution of a class of geometric-analogy intelligence-test questions. In Marvin L. Minsky, editor, *Semantic Information Processing*, pages 271–353. MIT Press, Cambridge, Massachusetts, 1968.
4. P.A. Flach. Strongly typed inductive concept learning. In D. Page, editor, *Proceedings of the 8th International Conference on Inductive Logic Programming*, volume 1446, pages 185–194. Springer-Verlag, 1998.
5. E. Hirowatari and S. Arikawa. Explanation based generalisation by analogical reasoning. In *Proceedings of the 2nd International Workshop on Inductive Logic Programming*. Institute for New Generation Computer Technology, 1992.
6. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the Association for Computing Machinery*, 27(4):797–821, 1980.
7. A. Hutchinson. Metrics on terms and clauses. In *Proceedings of the 9th European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence, pages 138–145. Springer-Verlag, 1997.

8. D. Kazakov, S. Pulman, and S. Muggleton. The fracas dataset and the ll challenge. Technical report, 1998.
9. N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.
10. S. Muggleton and L. De Raedt. Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.
11. Shan-Hwei Nienhuys-Cheng. Distance between herbrand interpretations: A measure for approximations to a target concept. In *Proceedings of the 7th International Workshop on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1997.
12. G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.
13. J. Ramon and M. Bruynooghe. A framework for defining distances between first-order logic objects. In *Proceedings of the 8th International Conference on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence, pages 271–280. Springer-Verlag, 1998.
14. J. Ramon, M. Bruynooghe, and W. Van Laer. Distance measures between atoms. In *Proceedings of the CompulogNet Area Meeting on 'Computational Logic and Machine Learning'*, pages 35–41, 1998.
15. K. Sadohara and M. Haraguchi. Analogical logic program synthesis from examples. In N. Lavrač and S. Wrobel, editors, *Proceedings of the 8th European Conference on Machine Learning*, volume 912 of *Lecture Notes in Artificial Intelligence*, pages 232–244, Berlin, Heidelberg, New York, 1995. Springer-Verlag.