Towards the definition of learning systems with configurable operators and heuristics *

Fernando Martínez-Plumed, Cèsar Ferri, José Hernández-Orallo, and María José Ramírez-Quintana

DSIC, Universitat Politècnica de València, Camí de Vera s/n, 46022 València, Spain.
{fmartinez,cferri,jorallo,mramirez}@dsic.upv.es

Abstract. The number and performance of machine learning techniques dealing with complex, structured data has considerably increased in the past decades. However, the performance of these systems is usually linked to a transformation of the feature space (possibly including the outputs as well) to a more convenient, flat, representation, which typically leads to incomprehensible patterns in terms of the transformed (hyper-)space. Alternatively, other approaches do stick to the original problem representation but rely on specialised systems with embedded operators dealing with specific kinds of data. This specialisation makes it very difficult to have general systems which are able to deal with different kinds of complex data. In this paper we present and explore a general rule-based learning setting where operators can be defined and customised for each kind of problem. While one particular problem may require generalisation operators, another problem may require operators which add recursive transformations to explore the structure of the data. A right choice of operators can embed transformations on the data but can also determine the way in which rules are generated and transformed, so leading to (apparently) different learning systems. However, this generality requires an adaptive and flexible rethinking of heuristics, with a model-based reinforcement learning approach, to tame the search space.
Keywords: machine learning operators, complex data, heuristics, inductive programming, reinforcement learning, Erlang.

1 Introduction

We know many machine learning systems which can deal with, e.g., the complex structure of a network graph. Other machine learning systems can learn, e.g., to transform or parse sequences into other sequences. Similarly, other systems can deal with time series, with multi-relational data and so on. Many of these systems share some common underlying principles but usually differ on data representation, pattern representation, learning operators and heuristics. Despite all

* This work was supported by the MEC projects CONSOLIDER-INGENIO 26706 and TIN 2010-21062-C02-02, GVA project PROMETEO/2008/051, and the REFRAME project granted by the European Coordinated Research on Long-term Challenges in Information and Communication Sciences & Technologies ERA-Net (CHIST-ERA), and funded by the Ministerio de Economa y Competitividad in Spain. Also, F. Martnez-Plumed is supported by FPI-ME grant BES-2011-045099.

these approaches, there is no general-purpose machine learning system which can deal with *all* of these problems *preserving* the problem representation (although [6] addresses the ambitious task of formulating a general framework for data mining). There are of course several paradigms using, e.g., distances or kernel methods for structured data [13][9] which can be applied to virtually any kind of data, provided we can define similarity functions to compare the individuals. However, this generality comes at the cost of losing the original problem representation and typically losing the recursive character of many data structures.

Other paradigms, such as inductive programming (ILP [22], IFP [16] or IFLP [12]), are able to tackle any kind of data thanks to the expressive power of first-order logic (or term rewriting systems). However, each system has a predefined set of operators (e.g. *lgg* [23], inverse entailment [21], splitting conditions in a decision tree, or others) and an embedded heuristic. Even with the help of background knowledge it is still virtually impossible to deal with, e.g., an XML document, if we do not have the appropriate operators to delve into its structure.

In this paper, we push forward the idea of machine learning systems for which the operators can be modified and finetuned for each problem. Making the user or the problem adapt its own operators is significantly different to the use of feature transformations or specific background knowledge. In fact, it is also significantly more difficult, since operators can be very complex things and usually embed the essence of a machine learning system. A very simple operator, such as *lgg*, requires several lines of code in almost any programming language, if not more. Writing and adapting a system to a new operator is not always an easy task. As a result, having a system which can work with different kinds of operators at the same time is a challenging proposal beyond the frontiers of the state of the art in machine learning.

In addition, machine learning operators are the tools to explore the hypothesis search space. Consequently, some operators are usually associated to some heuristic strategies (e.g., generalisation operators and bottom-up strategies). By giving more freedom to the kind of operators a system can use, we lose the capacity to analyse and define particular heuristics to tame the search space. This means that heuristics must be overhauled, as *decisions* about the operator that must be used at each particular state of the learning process.

We therefore propose a setting where operators can be written or modified by the user. Since operators are defined as functions which transform patterns, we clearly need a language for defining operators which can integrate the representation of the examples, the representation of the patterns and the representation of the operators. We will argue that functional programming languages, with reflection and higher-order primitives, are appropriate for this, and we will choose a powerful and relatively popular programming language in this family, Erlang [1]. A not less important reason for using a functional language is that operators can be understood by the users and properly linked with the data structures used in the examples and background knowledge, so making the specification of new operators easier. The language also sets the general representation of examples as equations, patterns as rules and models as sets of rules.

From here, we devise a flexible architecture which works with populations of rules and programs, which *evolve* as in an evolutionary programming setting or a learning classifier system [14]. Operators are applied to rules and generate new rules, which are combined with existing or new programs. With appropriate operators and using some optimality criteria (based on coverage and simplicity) we will eventually find some good solutions to the learning problem. However, without heuristics, the number of required iterations gets astronomically high. This issue is addressed with a reinforcement learning approach, where the application of an operator over a rule is seen as a decision problem, for which learning also takes place, guided by the optimality criteria which feed a rewarding module.

All this configures an architecture where users can write (or adapt) their operators, according to the problem, data representation and the way the information should be navigated. Data instances, background knowledge, rules, programs and operators are all written in the same language, Erlang. Heuristics are learnt as a result of a decision process where each action is defined as a choice of operator and rule. Interestingly, different problems using the same operators can reuse the heuristics. As a result, the architecture can be seen as a 'system for writing machine learning systems' or to explore new operators.

The paper is organised as follows. Section 2 makes a short account of the many approaches and ideas which are related to this proposal. Section 3 describes the setting, with its main components and architecture, and how all the pieces work together. Section 4 includes some examples which illustrate how operators are defined and how solutions are reached. Section 5 closes the paper.

2 Previous work

The proposal we present in this paper is related to different areas of machine learning: different appropaches to learning from complex data, reinforcement learning, Learning Classifiers Systems, evolutionary techniques, meta-learning, etc. In this section we summarise some of the previous work in these fields and see what we re-use from them and what the differences are in case.

Inductive programming [16], inductive logic programming [22] and some of the related areas such as relational data mining [8] are arguably the oldest attempts to handle complex data. They can be considered *general* machine learning systems, because any problem can be represented, preserving its structure, with the use of the Turing-complete languages underneath: logic, functional or logic-functional. Apart from their expressiveness, the advantage of these approaches is the capability of capturing complex problems in a comprehensible way. This makes ILP especially appropriate for scientific theory formation tasks where the data are structured, the model may be complex, and the comprehensibility of the generated knowledge is essential. Learning systems using higher-order (see, e.g., [18]) were one of the first approaches to deal with complex structures, which were usually flattened in ILP. Despite the power of higher-order function to explore complex structure, this approach has been practically discontinued.

All these systems are based on the use of generalisation operators. For instance, Plotkin's lgg [23] operator works well with a specific-to-general search. The ILP system Progol [21] combines the Inverse Entailment with general-to-specific search through a refinement graph. The Aleph system [24] is based on Mode Direct Inverse Entailment (MDIE). In inductive functional logic programming, the FLIP system [12] includes two different operators: inverse narrowing and a consistent restricted generalisation (CRG) generator. The set of operators configures and delimits the performance of the learning system. A hybrid approach that combines Genetic Algorithms and ILP, which defines unification and anti-unification operators as bitwise operations on binary strings, is presented in [26].

As an evolution of ILP into the fields of (statistical) (multi-)relational learning or related approaches, many systems have appeared to work with rich data representations. In [4], for example, we can find an extensive description of the current and emerging trends in this field where the authors propose to go beyond supervised learning and inference, and consider decision-theoretic planning and reinforcement learning in relational and first-order settings.

Structured Prediction (SP) is one example in this context, where not only the input is complex but also the output. This has led to new and powerful techniques, such as Conditional random fields (CRFs) [17], which use a log-linear probability function to model the conditional probability of an output y given an input x where Markov assumptions are used in order to make inference tractable. Other well-known Global Model is SVM for Interdependent and Structured Output spaces (SVM-ISO, also known as SVM^{struct}) [27].

There have been several approaches applying planning and reinforcement learning to structured machine learning [25]. While the term Relational Reinforcement Learning (RRL) [7, 25] seems to come to mind, it offers state-space representation that is much richer than that used in classical (or propositional) methods, but its goal is not structured data. Other related approaches are, for instance, incremental models [3, 19] which try to solve the combinatorial nature of the very large input/output structured spaces since the structured output is built incrementally. These methods can be applied to a wide variety of techniques such parsing,machine translation,sequence labeling and tree mapping.

Some of these previous approaches use special functions explicitly defined on the individual space, being these functions probabilistic distributions, metrics or kernels. These methods either lack a model (they are instance-based methods) or the model is defined in terms of the transformed space. A recent proposal which has tried to re-integrate the distance-based approach with the pattern-based approach is [11], which works with several structured domains (sets, lists, trees, graphs). In fact, operators can be seen to be consistent with the metric (but not derived from it). This makes it possible to make general systems (e..g, decision trees) which can deal with all these kinds of data as the Newton trees [20]. However, the operators had to be embedded in the system and could not be changed or modified by the user, as we propose in this paper.

Finally, there is an old but related approach, somehow in between genetic algorithms and reinforcement learning. *Learning Classifier Systems* or *LCSs* (rule-based systems) [15] employ two biological metaphors: evolution and learning which are respectively embodied by the genetic algorithm, and a reinforcement learning-like mechanism appropriate for the given problem. Both mechanisms rely on what is referred to as the *environment* of the system (the source of input data). The architecture of our system will resemble in some ways the LCS approach.

Learning to learn is one of the (required) features of our setting and is related to the area of meta-learning [2]. Whereas learning at the base level focuses on accumulating experience on a specific learning task (e.g., credit rating, medical diagnosis, mine-rock discrimination, etc.), learning at the metalevel is concerned with accumulating experience on the performance of multiple applications of a learning system. This may reduce the efforts for designing domain-specific learning algorithms, and lead to more robust and general learning architectures.

3 Setting

Given the inherent complexity of dealing with structured input/output data both in their representation and in their use for inference or learning tasks, we have adopted a functional language for representing the problem and its solution. The advantages of using the same representation language for examples, hypotheses and background knowledge (in this case, rules expressed as unconditional/conditional equations) has been previously shown by the fields of ILP, IFP and IFLP. For this reason, we use Erlang, a functional language with reflection mechanisms which allow us to interact easily with the meta-level representation of the problem to solve. Basically, the key element in our setting is the rule, and the way in which the mechanism of induction is carried out is by applying transformations on the rules to obtain new rules. These transformations are performed by *operators*, which are pieces of code that take a set of rules as input and return a set of (probably new) rules.

As we have mentioned, one of the goals is to allow the user to configure the operators that will be used to solve the problem. Depending on the operators that the user provides to the system, it could well behave as a decision tree (if we implement operators that apply some conditions on the rules), or as a bottom-up concept covering algorithm (if we provide generalisation operators). That is, a system may behave very differently by changing the operators. However, how does a system decide which operator apply to which rule? We set that the system must receive feedback in form of a numerical reward in a similar way as reinforcement learning.

3.1 Notation

The evidence E of the problem to solve consists of the positive examples E^+ and the (possibly empty) set of negative examples E^-. An instance e is an

unconditional equation of the way $l \to r$ where l is called the left-hand side (lhs) of e, r is the right-hand side (rhs) of e and r is a term in normal form (it cannot be reduced). A rule $g \in \mathcal{R}$ (where $\mathcal{R}$ denotes the space of all (conditional) rules) is a conditional rule of the form l [when C] $\to r$ where C is the condition (a sequence of guards), and l and r are the left-hand side and the right-hand side of g, respectively. If C is empty, g is said to be an unconditional rule. Background knowledge B is a set of rules, possibly empty.

The system works on two sets: a set of individuals R (rules in $\mathcal{R}$) and a set of communities P (programs in $2^{\mathcal{R}}$). Note that examples are also unconditional rules. A program p is a set of rules. An *operator* o is a function that transforms a rule into a new rule, that is $o : \mathcal{R} \to \mathcal{R}$. Analogously, a *combiner* c is a function $c : 2^{\mathcal{R}} \to 2^{\mathcal{R}}$ that transforms programs into new programs. A program p is a solution to the learning problem if it covers all positive examples $B \cup p \models E^+$ (posterior sufficiency or completeness) and does not cover any negative examples $B \cup p \models E^-$ (posterior satisfactibility or consistency). Our system has the aim of obtaining complete solutions, but their consistency is not a mandatory property, so approximate solutions are possible. As usual, the coverage relation can also be defined in terms of the operational mechanism of the functional language. An example $l \to r$ is covered by a program p if the normal forms of l and r computed with respect to p are equal. The function $cover : 2^{\mathcal{R}} \to \mathbb{N}$ calculates the coverage of a program $p \in 2^{\mathcal{R}}$ and it is defined as $cover(p) = card(\{e \in E^+ : B \cup p \models e\})$, where $card(S)$ denotes the cardinality of a set S.

3.2 General Architecture

Figure 1 illustrates the proposed architecture. In this section we give a description of the main components of the system and their interaction.

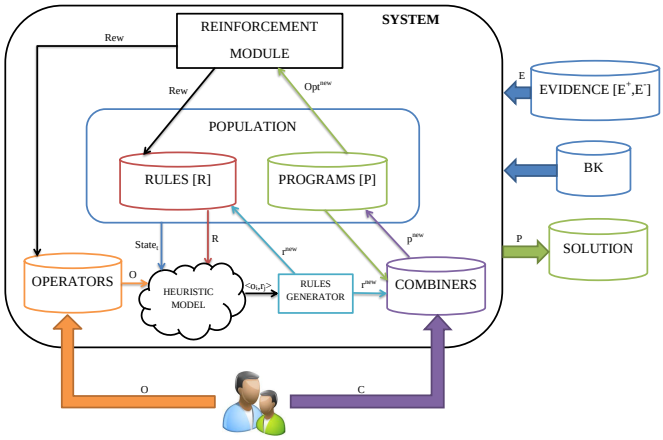

Fig. 1: Prototype System Architecture.

The inputs of our setting are: the positive and negative evidence (a set of equations or observations about the target function) and, if available, the background

knowledge. Additionally, the user can optionally provide the system with the following information: new learning operators that are added to the current set of operators O, and new combiners that are added to the current set of combiners C. This capability of adapting the set of learning operators as well as the set of combiners is one of the characteristics of this setting. Both are part of the meta-level facility that the system offers to the users.

Rule and Program Repositories Initially, the set of rules R is populated with the positive evidence E^+ and the set of programs P is populated with as many unitary programs as rules there are in R. In addition, for each initial program $p \in P$, we assign an optimality value $Opt(p) = cover(p)$. Both repositories are updated at each step of the algorithm. First, the *Rule Generator* process builds new rules (r^{new}), which are added to R. By applying the combiners, (r^{new}) is mixed with the programs in P in order to generate a new program p^{new}, which is added to P.

System Operators The definition of customised operators is one of the key concepts of our proposal. The main idea is that, when the user wants to deal with a new problem, she can define her/his own set of *operators*, especially suited for the data structures of the problem. This feature allows our system to adapt to the problem. More precisely, an *operator* is defined as a function which is applied to a rule in order to generate new rules. For instance, imagine that we want to define an operator to deal with lists. Given a rule $f(X) \rightarrow Y$ where the input attribute X is a list, the operator can extract the head of X and return it as the rhs of the new rule. So, the operator could be defined as: $takeHead(f(X) \rightarrow Y)$ $[when\ X\ is\ a\ List] \rightarrow (f(X) \rightarrow head(X))$.

System Combiners Combiners evolve the population of programs. Here, we only show two simple combiners (although other possibilities are considered): *addition*, which adds the program that results from joining the new rule r^{new} generated by the *Rule Generator* with the best program (in terms of optimality); and *union* which joins the two best programs (also in terms of optimality) in P.

Other processes The *Reinforcement Module* is used to guide the *Rule Generator* in each step of the algorithm. Following the reinforcement learning approach, we define the system state S as the set composed by R and P. An action A is a tuple $< r_i, o_i >$ where r_i is a rule and o_i is an operator. Given an state S, an action A is chosen by the *Heuristic Model* and sent to the *Rule Generator*. This creates new rules (and programs), which causes the system to move to a new state. Initially, the *Heuristic Model* does not have enough evidence and the choice is random, but after a few iterations, the model is learnt by using a machine learning technique (for instance, a decision tree). This model is trained to predict the reward after a given action A. With this model, we just choose the action which maximises the estimated reward.

Rewards are generated as follows. From the optimality Opt^{new} of the new program p^{new} generated by the combiner, the *Reinforcement Module* calculates a reward *Rew*. *Rew* is used to update the optimality of the action $A =< r_i, o_i >$ what influences their subsequent selection by the heuristic model.

4 Examples

Following the setting described in the previous section, we developed a first prototype, which included the overall architecture, the definition of operators and a simplified version of the RL-based heuristics. In this section, we describe three different examples where we illustrate how operators are defined and used to iteratively approach the solution. We also use these examples to bring out some difficulties that need to be addressed.

4.1 Sequence processing

Let us start with a toy example of the kind used in structured prediction, where not only the input is structured but also the output. Consider the problem of learning a transformation over the words formed by a given alphabet. More precisely, suppose we have a set of instances where both the input and output are lists (i.e., strings). Consider the very particular case where we have a small alphabet of a non-empty finite set of symbols $\Sigma = \{a, t, c, g, u\}$ and the transformation just replaces t with u. Instances would look like this: $trans([t, c, g, a, t]) \rightarrow [u, c, g, a, u]$.

The first thing we need to define is the basic *replacement functions* for the symbols in the alphabet. This is done in the background knowledge, with functions like: $f_{at}(a) \rightarrow t$; $f_{cg}(c) \rightarrow g; ...$ Typically, all the combinations can be defined or only some of them if some replacements are not possible.

According to the structure of the example, a string, we need a way to navigate the structure and apply local or global changes. In order to do this we need to define appropriate operators. The first operator, $applyMap$ is a mechanism to convert a rule into another rule which introduces the higher-order function *map*, which applies a parametrised function to the whole list. The definition of this operator is written in Erlang, but it can be informally defined as follows: $applyMap(trans(X) \rightarrow Y) \Rightarrow trans(X) \rightarrow map(V_F, X)$, where X and Y stand for any list and V_F is a function variable (a higher-order variable). Notice that $\rightarrow$ represents the rewriting symbol of equations (rules), while $\Rightarrow$ represents the transformation performed by the operator from rule to rule. In order to introduce a replacement function, we need more operators, such as $addBK_f$, which fill the gap V_F by introducing the function f from the BK. Note that at this moment it seems a matter of taste whether we define one operator for each replacement function or a single stochastic operator for all of them, but the difference is important for heuristics. An example of one of each of these operators is: $addBK_f(trans(X) \rightarrow map(V_F, X)) \Rightarrow trans(X) \rightarrow map(f, X)$. Finally, we need a way of generalising input (and output) strings. This is performed by the

genPat operator: $genPat(trans(X) \rightarrow Y) \;\Rightarrow\; trans(V_S) \rightarrow Y$, where V_S is a string variable.

For this toy example there is a simple sequence of operator applications which turns a simple example into a general solution. For instance, given the instance $trans([t,c,g,a,t]) \rightarrow [u,c,g,a,u]$, we have this sequence.

$$\begin{aligned} genPat(trans([t,c,g,a,t]) \rightarrow [u,c,g,a,u]) &\;\Rightarrow\; trans(V_S) \rightarrow [u,c,g,a,u] \\ applyMap(trans(V_S) \rightarrow [u,c,g,a,u]) &\;\Rightarrow\; trans(V_S) \rightarrow map(V_F, V_S) \\ addBK_{f_{tu}}(trans(V_S) \rightarrow map(V_F, V_S)) &\;\Rightarrow\; trans(V_S) \rightarrow map(f_{tu}, V_S) \end{aligned}$$

This latter equation $trans(V_S) \rightarrow map(f_{tu}, V_S)$ is the solution for this toy example. Given the simplicity and the relatively small number of operators, the effect of the coverage mechanisms and the heuristics is not critical, and the prototype solves this problem in a few seconds.

4.2 Bunches of keys

We will continue with a more complex problem, a well-known multi-instance classification problem. Consider the problem of determining whether a key in a bunch of keys can open a door [18]. More precisely, for each bunch of keys either no key opens the door or there is at least one key which opens the door. Each instance is given by a bunch of keys, where each key has several features, so there is a two-level structure (sets of lists). While this is a prototypical multiple-instance problem, it is similar to a number of important practical problems, e.g. drug activity prediction [5].

We model a bunch of keys as a set of keys. Each key, in turn, is modeled as a list capturing four of its properties: the company that makes it (*Abloy, Chubb, Rubo, Yale*), its number of prongs (an integer), its length (*Short, Medium, Long*) and its width (*Narrow, Normal, Broad*). A training example (a bunch with two keys which does open the door) may look like this: $opens([[abloy, 3, medium, narrow], [chubb, 6, medium, normal]]) = \top$.

Given a set of such examples, we want to learn the function *opens : Bunch* $\rightarrow \Omega$.. $\Omega = \{\top, \bot\}$. For this, we need a function *setExists(Key,Bunch)* which evaluates ($\top$ or $\bot$) whether there exists a *Key* in a *Bunch*. This function will belong to the background knowledge. We need also to provide the system with a set of operators. We again need an operator which incorporates conditions on the right hand side of a rule: $addBK(opens(X) = \top) \;\Rightarrow\; opens(X) \rightarrow setExists([], X)$.

This incorporates an empty list of conditions. Now we need operators to add conditions. We will have one operator for each attribute value. For instance, the operator for inserting a condition for keys with *abloy* is: $KCond(opens(X) \rightarrow setExists(C, X)) \;\Rightarrow\; opens(X) \rightarrow setExists([abloy|C], X))$.

Finally, we need a generalisation operator which introduces a variable instead of a list: $genPat(opens(X) = Y) \;\Rightarrow\; opens(V_L) \rightarrow Y$.

If the prototype and operators are provided, given the original evidence for this example (five $\top$ instances and four $\bot$ instances), it will return the following definition: $opens(X) \rightarrow setExists([abloy, medium], X)$, which means that *a*

bunch of keys opens the door if and only if it contains an abloy key of medium length, which is the proposed solution for this classical example.

4.3 Web categorisation

The last example corresponds to a *web* classification problem with a higher level of difficulty. It was originally proposed in [10]. The evidence of the problem is modelled with 3 parameters described as follows: *Structure* (the graph of links between pages is represented as ordered pairs where each node encodes a linked page), *Content* (the content of the web page is represented as a set of attributes with the keywords, the title, etc.), and Use: (the information derived from connections to a web server which is encoded by means of a numerical attribute with the daily number of connections).

The goal of the problem is to categorise which web pages are about sports. A training example looks like this: $sportsWeb(Structure, Content, Connections) \rightarrow \top$ where the *Structure* attribute may be for instance $[\{[olympics, games], [swim]\}, \{[swim], [win]\}, \{[win], [medal]\}]$ and is interpreted in the following way: the first component of the list stands for the current web page with keywords "olympics" and "games". This page links to another page which has "swim" as its only keyword. There are other two connections. The *Content* may be $[\{olympics, 30\}, \{held, 10\}, \{summer, 40\}]$, which represents the frequency (number of occurrences) of the most relevant words in the web page. Finally the *Connections*, which is just an integer attribute which represents the number of connections.

Given the structure of the data, we need to add functions to the background knowledge to navigate this structure. We define *graphExists(Edge,Graph)* which checks whether an edge is in a graph, and *setExists(Key,List)* which tests whether the keyword Key belongs to the list. Again, we also need to provide the system with a set of operators. As in previous cases, we can reuse a generic operator to select some function from the background knowledge (one for each function) in order to replace the right hand side of the rules: $addBK_{graph}(sportsWeb(S, C, U) \rightarrow \top) \Rightarrow sportsWeb(S, C, U) \rightarrow graphExists(\{[], []\}, S)$, which introduces an empty condition about a connection between pages. We can similarly define an operator for introducing a condition over the sets.

Another useful operator takes some type constants and add it to the condition of the *setExists* function (first attribute) and another operator which generate a node and add it as a node to search in the graph attribute of the function *graphExists*: $linkl_{football}(sportsWeb(S, C, U) \rightarrow graphExists(\{X, Y\}, S)) \Rightarrow sportsWeb(S, C, U) \rightarrow graphExists(\{[football|X], Y\}, S)$. Note that this operator is parametrised for the different attribute values. Finally, we need a generalisation operator for each input pattern of the rules: $genPat_1(sportsWeb(S, C, U) \rightarrow \top) \Rightarrow sportsWeb(V_S, C, U) \rightarrow \top$. There are also some other operators to generalise the second and third arguments.

From here, the prototype runs on the examples and when the stopping criterion is met, a list of the best programs sorted by optimality is returned. For instance, our system found the following correct program which defines the

sportsWeb function:

$$\{sportsWeb(V_S, V_C, V_U) \rightarrow graphExists(\{[final], [match]\}, V_S).$$
$$sportsWeb(V_S, V_C, V_U) \rightarrow setExists([\{athens\}]], V_C).$$
$$sportsWeb(V_S, V_C, V_U) \rightarrow setExists([\{europe\}]], V_C). \ \}$$

which means that *if the word 'athens' or 'europe' appears in Content, and Structure contains the link* $\{[final], [match]\}$ *then this is a sport web page.*

5 Conclusions and future work

The increasing interest in learning from complex data has led to a more integrated view of this area, where the same (or similar) techniques are used for a wide range of problems using different data and pattern representations. This general view has not been accompanied by general systems, at least if the original data representation and structure is maintained. In fact, the most general approach can still be found in ILP or the more general area of inductive programming, where no recent breakthrough has taken place in how these systems can be further generalised.

In this paper, we have proposed that more general systems can be constructed by not only giving power to data and background knowledge representation but also to a flexible operator redefinition and the reuse of heuristics across problems and systems. Generality clearly entails a computational cost. In order to address this issue we rely on the definition of customised operators, depending on the data structures and problem at hand. Since this has to be done by the user, we need a language for expressing operators. A second component is heuristics, since the use of different operators precludes the system to use specialised heuristics for each of them. We have proposed this as a decision process, where operators are actions to be taken, and this is also seen as a *reinforcement* learning problem.

We have included some illustrative examples with a first prototype implementing the general architecture, and we have seen where the flexibility stands out but also where the (computational) problems arise. Our immediate future work is focussed on transforming the prototype into a learning system, including all the issues in the architecture we have presented in this paper. For instance, we need to further develop and refine the heuristics module of the system, with a better description of the state, better reinforcement learning models which could eliminate many useless explorations of the search space.

Overall, we are conscious that our approach entails some risks, since a general system which can be instantiated to behave virtually like any other system by a proper choice of operators is an ambitious goal. We hope that, even in an unsuccessful scenario, we can learn and discover some new properties, limitations and principles that can be used in the future.

References

1. J. Armstrong. A history of erlang. In *Proceedings of the third ACM SIGPLAN conf. on History of programming languages*, HOPL III, pages 1–26. ACM, 2007.

2. P. Brazdil and Giraud-Carrier. Metalearning: Concepts and systems. In *Metalearning*, Cognitive Technologies, pages 1–10. Springer Berlin Heidelberg, 2009.
3. H. Daumé III and J. Langford. Search-based structured prediction. 2009.
4. T. Dietterich, P. Domingos, L. Getoor, S. Muggleton, and P. Tadepalli. Structured machine learning: the next ten years. *Machine Learning*, 73:3–23, 2008.
5. T. G. Dietterich, R. Lathrop, and T. Lozano-Perezl. Solving the multiple-instance problem with axis-parallel rectangles. *Artificial Intelligence*, 89:31–71, 1997.
6. Sašo Džeroski. Towards a general framework for data mining. KDID'06, pages 259–300, Berlin, Heidelberg, 2007. Springer-Verlag.
7. S. Dzeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43:7–52, 2001. 10.1023/A:1007694015589.
8. S. Dzeroski and N. Lavrac, editors. *Relational Data Mining*. Springer-Verlag, 2001.
9. V. Estruch, C. Ferri, J. Hernández-Orallo, and M. J. Ramírez-Quintana. Similarity functions for structured data. an application to decision trees. *Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial*, 10(29):109–121, 2006.
10. V. Estruch, C. Ferri, J. Hernández-Orallo, and M. J. Ramírez-Quintana. Web categorisation using distance-based decision trees. *ENTCS*, 157(2):35–40, 2006.
11. V. Estruch, C. Ferri, J. Hernandez-Orallo, and M.J. Ramirez-Quintana. Bridging the Gap between Distance and Generalisation. *Computational Intelligence*, 2012.
12. C. Ferri, J. Hernández-Orallo, and M.J. Ramírez-Quintana. Incremental learning of functional logic programs. In H. Kuchen and K. Ueda, editors, *FLOPS*, volume 2024 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2001.
13. T. Gärtner. *Kernels for Structured Data*. PhD thesis, Universitat Bonn, 2005.
14. J. Holland and Booker. What is a learning classifier system? In *Learning Classifier Systems*, volume 1813 of *LNCS*, pages 3–32. 2000.
15. J. H. Holmes, P. Lanzi, and W. Stolzmann. Learning classifier systems: New models, successful applications. *Information Processing Letters*, 2002.
16. E. Kitzelmann. Inductive programming: A survey of program synthesis techniques. In *3rd Workshop AAIP*, volume 5812 of *LNCS*, 2010.
17. J. Lafferty and A McCallum. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. ICML '01, pages 282–289, 2001.
18. J.W. Lloyd. Knowledge representation, computation, and learning in higher-order logic. 2001.
19. F. Maes, L. Denoyer, and P. Gallinari. Structured prediction with reinforcement learning. *Machine Learning Journal*, 77(2-3):271–301, 2009.
20. F. Martínez-Plumed, V. Estruch, C. Ferri, J. Hernández-Orallo, and M. J. Ramírez-Quintana. Newton trees. In *Australasian Conference on Artificial Intelligence*, volume 6464 of *LNCS*, pages 174–183, 2010.
21. S. Muggleton. Inverse entailment and progol. *New Generation Computing*, 1995.
22. S. H. Muggleton. Inductive logic programming: Issues, results, and the challenge of learning language in logic. *Artificial Intelligence*, 114(1–2):283–296, 1999.
23. G. Plotkin. A note on inductive generalization. *Machine Intelligence*, 5, 1970.
24. A. Srinivasan. *The Aleph Manual*, 2004.
25. P. Tadepalli, R. Givan, and K. Driessens. Relational reinforcement learning: An overview. In *In Proc. of the Workshop on Relational Reinforcement Learning*, 2004.
26. Alireza Tamaddoni-Nezhad and Stephen Muggleton. A genetic algorithms approach to ilp. In *Proc. of the 12th Int. Conf. on Inductive logic programming*, ILP'02, pages 285–300, Berlin, Heidelberg, 2003. Springer-Verlag.
27. I. Tsochantaridis, T. Hofmann, T. Joachims, and Y. Altun. Support vector machine learning for interdependent and structured output spaces. In *ICML*, 2004.