Learning with Configurable Operators and RL-Based 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. In this paper, we push forward the idea of machine learning systems for which the operators can be modified and finetuned for each problem. This allows us to propose a learning paradigm where users can write (or adapt) their operators, according to the problem, data representation and the way the information should be navigated. To achieve this goal, data instances, background knowledge, rules, programs and *operators* are all written in the same functional language, Erlang. Since changing operators affect how the search space needs to be explored, heuristics are learnt as a result of a decision process based on reinforcement learning where each action is defined as a choice of operator and rule. As a result, the architecture can be seen as a 'system for writing machine learning systems' or to explore new operators.

Keywords: machine learning operators, complex data, heuristics, inductive programming, reinforcement learning, Erlang.

1 Introduction

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 that are only able to deal with specific types of data.

Despite all these approaches and the vindication of more general frameworks for data mining [6], there is no general-purpose machine learning system which can deal with *all* of these problems *preserving* the problem representation. There

[⋆] 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 Economía y Competitividad in Spain. Also, F. Martínez-Plumed is supported by FPI-ME grant BES-2011-045099.

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 [23], 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* [24], inverse entailment [22], 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 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. 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 tools to explore the hypothesis 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, patterns and 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 (RL) 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. 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 introduces how operators are expressed and applied. Section 4 describes the RL-based heuristics used to guide the learning process. Section 5 includes some examples which illustrate how operators are defined and how solutions are reached. Section 6 closes the paper.

2 Previous Work

The system we present in this paper is related to different areas of machine learning: learning from complex data, reinforcement learning, Learning Classifiers Systems, evolutionary techniques, meta-learning, etc. In this section we summarise some of the previous works in these fields somehow related to our proposal.

Inductive programming [16], inductive logic programming (ILP) [23] 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. ILP, for instance, has been found 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., [19]) were one of the first approaches to deal with complex structures, which were usually flattened in ILP. Despite the power of higher-order functions to explore complex structure, this approach has never become mainstream.

All these systems are based on the use of different fixed operators. For instance, Plotkin's lgg [24] operator works well with a specific-to-general search.

The ILP system Progol [22] combines the Inverse Entailment with general-to-specific search through a refinement graph. The Aleph system [26] 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. In any case, the set of operators configures and delimits the performance of each learning system. Also, rules that are learned on a first stage can be reused as background knowledge for subsequent stages (incremental learning). Hybrid approaches that combine Genetic Algorithms and ILP have also been introduced as in [29].

As an evolution of ILP into the fields of (statistical) (multi-)relational learning or related approaches, many systems have been developed to work with rich data representations. In [4], for example, we can find an extensive description of the current and emerging trends in the so-called 'structured machine learning' 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 of learning from complex data 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) [18], 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}) as a SP evolution of [13] (Kernels) or [9] (distances). [30]. Also, *hierarchical classification* can be viewed as a case of SP where taxonomies and hierarchies are associated with the output [17].

Some of these previous approaches use special functions (probabilistic distributions, metrics or kernels) explicitly defined on the individual space. 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], (leading, e.g., to Newton trees [21]).

There have been several approaches applying planning and reinforcement learning to structured machine learning [28]. While the term Relational Reinforcement Learning (RRL) [7,28] 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 [3,20] 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 as parsing, machine translation, sequence labelling and tree mapping.

Finally, there is an approach, somewhat in between genetic algorithms and reinforcement learning, known as *Learning Classifier Systems* (*LCSs*) [15]. LCSs 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]. Learning at the metalevel is concerned with accumulating experience on the performance of multiple applications of a learning system. A more integrated approach resembling meta-learning and incremental learning is [25], where the authors present the *Optimal Ordered Problem Solver* (OOPS), an optimally fast way of incrementally solving each task in the sequence by reusing successful code from previous tasks.

3 Configuring Rule Operators

After this review of related work, we still perceive a lack of flexibility in the way in which different problems can be handled, especially when structured learning is required. As we have mentioned in Section 1, in this paper we set the goal of constructing a system which can be configured with different (possibly user-defined) operators, and where the heuristics are also learned from previous applications of operators for the same or similar problems. As a long-term goal, this can be roughly seen as a general system for designing customised systems for applications with complex data.

In order to achieve the above-mentioned goals, we need to use configurable operators, instead of hard-wired operators. Changing hard-wired operators requires the modification of dozens of lines of code and usually entails a re-writing (or complete overhauling) of heuristics. Instead, in our approach the heuristics will be substituted by a reinforcement learning approach, which will determine which pair of operator and rule will be chosen at each state of the system.

Additionally, we will represent operators in the same language already used for examples, models and background knowledge. The advantages of using the same representation language (in this case, rules expressed as unconditional / conditional equations) has been previously shown by the fields of ILP, IFP and IFLP (except for operators). Hence, we look for a flexible language, with powerful features for defining operators and able to represent all other elements (theories and examples) in an understandable way. For this reason we use Erlang, a functional language with reflection mechanisms which allows us to interact easily with the meta-level representation of how rules and programs are transformed by operators.

3.1 Notation

Let Σ be a set of *function symbols* together with their arity and $\mathcal{X}$ a countably set of *variables*, then $\mathcal{T}(\Sigma, \mathcal{X})$ denotes the set of *terms* built from Σ and $\mathcal{X}$. The set of variables occurring in a term t is denoted *Var(t)*. A term t is a *ground term* if $Var(t) = \varnothing$.

An equation is an expression of the form $l = r$ where l and r are terms. l is called the left hand side (*lhs*) of the equation and r is the right hand side (*rhs*). $\mathcal{R}$ denotes the space of all (conditional) functional rules ρ of the way l [when G] $\rightarrow T, r$ where l and r are the lhs and the rhs of ρ (respectively), $G = \{g_1, g_2, \ldots g_m \mid m \geq 0\}$) is a set of conditions or Boolean expressions called guards, and $T = b_1, \ldots, b_n$, the tail of ρ, is a sequence of equations. If $G = \varnothing$, then ρ is said to be an unconditional rule. Let $\mathcal{P} = 2^{\mathcal{R}}$ be the space of all possible functional programs formed by sets of rules $\rho \in \mathcal{R}$. Given a program $p \in \mathcal{P}$, we say that term t reduces to term s with respect to p, $t \rightarrow_p s$, if there exists a rule l [when G] $\rightarrow T, r \in p$ such that a subterm of t at occurrence u matches l with substitution θ, all conditions $h_i\theta$ holds, for each equation $b_{i_l} = b_{i_r} \in T$, $b_{i_l}\theta$ and $b_{i_r}\theta$ have the same normal form (that is, $b_{i_l}\theta \rightarrow_p^* b$, and $b_{i_r}\theta \rightarrow_p^* b$ and b can not be further reduced) and s is obtained by replacing in t the subterm at occurrence u by $r\theta$.

An example e is a rule without condition nor tail, that is e is of the form $l \rightarrow r$, being r in normal form and both l and r are ground. We say that an example $l \rightarrow r$ is covered by a program p (denoted by $p \models \{l \rightarrow r\}$) if l and r have the same normal form with respect to p. A functional program $p \in \mathcal{P}$ is a solution of a learning problem defined by a set of positive examples E^+, a (possibly empty) set of negative examples E^- and a background theory B if it covers all positive examples, $B \cup p \models E^+$ (posterior sufficiency or completeness), and does not cover any negative example, $B \cup p \not\models E^-$ (posterior satisfiability or consistency). Our system has the aim of obtaining complete solutions, but their consistency is not a mandatory property, so approximate solutions are possible. The function $Cov^+ : 2^{\mathcal{R}} \rightarrow \mathbb{N}$ calculates the positive coverage of a program $p \in 2^{\mathcal{R}}$ and it is defined as $Cov^+(p) = Card(\{e \in E^+ : B \cup p \models e\})$, where $Card(S)$ denotes the cardinality of the set S. Additionally, the function $Cov^- : 2^{\mathcal{R}} \rightarrow \mathbb{N}$ calculates the negative coverage of a program $p \in 2^{\mathcal{R}}$ and it is defined as $Cov^-(p) = Card(\{e \in E^- : B \cup p \models e\})$.

As we can see in Figure 1, our system works with two sets: a set of rules $R \subseteq \mathcal{R}$ and a set of programs $P \subseteq \mathcal{P}$, where each program $p \in P$ is composed by rules belonging to R. 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 there are rules in R.

3.2 Operators

The definition of customised operators is one of the key concepts of our proposal. The idea is to transform the set of rules R using a set of *operators* O (provided by the user or existing in the system). An operator $o \in \mathcal{O}$ is then a function $o : 2^{\mathcal{R}} \rightarrow 2^{\mathcal{R}}$ where $O \in \mathcal{O}$ will be the set of operators defined by the user.

An operator can be seen as a piece of code (as complex as the user may wants) which performs modifications over the *lhs* or *rhs* of a rule and which is written in the same functional language as the system (Erlang) to take advantage of its high-order and reflection capabilities. The main idea is that, when the user wants to deal with a new problem, he/she can define his/her own set of *operators*, especially suited for the data structures of the problem. This feature allows our system to adapt to the problem at hand.

Depending on the operators the user provides to the system, it could well behave as a decision tree or, more precisely, as a coverage-based rule learning system (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, the system may behave very differently by changing the operators.

Let us see an example. Given a rule $FName(Arguments) \rightarrow RHS$, where $Arguments$ is a list, imagine that we want to define an operator for obtaining the head of $Arguments$ and return it as the rhs of a new rule. This operator could be defined as:

$$takeHead(FName(Arguments) \rightarrow RHS)\ [when\ Arguments\ is\ a\ List] \\ \Rightarrow (FName(Arguments) \rightarrow head(Arguments)).$$

where $\Rightarrow$ represents the rule transformation relation defined by the operators. The codification in Erlang could be as follows:

```
Operator_takeHead(Rule) ->
(1)    {function,_,FName,_,{clause,_,Arguments,Guards,RHS}} = Rule,
(2)    {cons,_,L1,L2} = Arguments,
(3)    {function,_,FName,_,{clause,_,Arguments,Guards,L1}}.
```

where identifiers with a capital letter followed by any combination of uppercase and lowercase letters and underscores are Erlang variables, and other static (or constants) literals are Erlang atoms. In line 1, the `Rule` is parsed and transformed into a valid Erlang *abstract syntax tree* (AST) in order to easily access to its components: the Erlang *forms* `FName`, `Arguments`, `Guards` and `RHS`. Next, the operator decomposes `Arguments` into the Erlang meta-expression for lists (line 2), and finally, line 3 returns the new AST constructed by replacing the `RHS` part by `L1` in the AST obtained in line 1. For simplicity, we have omitted some further code for checking the arity and type of `Arguments`.

Our system also has a special kind of operators, called *combiners*, that only apply to programs. The *Program Generator* module (Figure 1) applies a combiner to the last rule ρ' generated by the *Rule Generator* module and the population of programs P. Thus, a combiner $c \in C$ can be formally described as a function $c : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P}$ that transforms programs into programs.

By default, our system provides two simple combiners (although other possibilities are considered): *addition*, joins the new rule ρ' generated with the best program (in terms of optimality) to the population P; and *union* which joins the two best programs (also in terms of optimality) in P.

4 LR-Based Heuristics

The freedom given to the user concerning the definition of their own operators implies the impossibility of defining specific heuristics to explore 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. For this, we have developed a model-based reinforcement learning approach, where the application of an operator over a rule is seen as a decision problem, for which

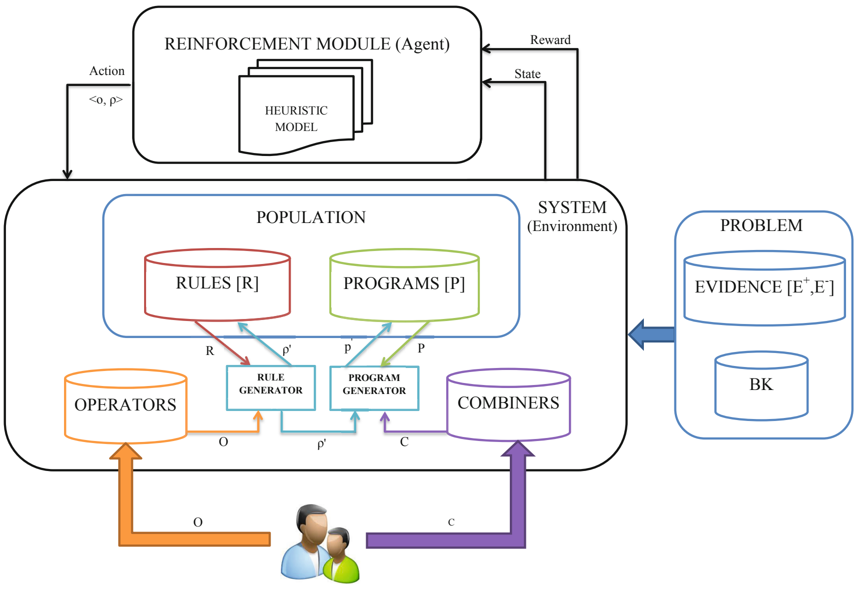

Fig. 1. Prototype System Architecture

learning also takes place, guided by the optimality criteria which feed a rewarding module. Below we will describe our approach.

4.1 State of the System

To guide the learning process we need a picture of the system in each step of the process (before and after applying an action) in terms of the quality of the set of rules and programs generated until now. Formally, we define a state at each iteration t of the system as a tuple $\sigma_t = \langle R, P \rangle$ which represent the population of rules R and programs P in t. The probable infinite number of states makes the abstraction of states necessary. How to do this? As we want to find a good solution to the learning problem, we describe each state σ_t by a tuple of features $s_t = \langle \phi_1, \phi_2, \phi_3, \phi_4, \phi_5 \rangle$ from which to extract relevant information in t:

1. *Global optimality* (ϕ_1): This feature shows the average optimality of all programs in P_t. In turn, the optimality of each program p consists of four factors:
   – *Positive Coverage* measures the proportion of positive examples covered by the program:
   $$PosCov(p) = \frac{Cov^+(p)}{Card(E^+)} \tag{1}$$
   – *Negative Coverage* measures the proportion of negative examples covered by the program:
   $$NegCov(p) = \frac{Cov^-(p)}{Card(E^-)} \tag{2}$$

– *Program Length Ratio* measures the cardinality of p w.r.t. the cardinality of the positive evidence:

$$ProgLength(p) = \frac{Card(p)}{Card(E^+)} \tag{3}$$

– *Applied Operators Ratio*, the idea is to penalise programs which have used a large number of operators:

$$OpersRate(p) = \frac{\sum_{\rho \in p} Card(PrevOpers(\rho))}{Card(O) \cdot Card(p)} \tag{4}$$

where $PrevOpers(\rho)$ is the list of previous operators applied to obtain the rule ρ. The optimality of a program p is computed by weighting the four factors according to its importance, in a way inspired by the *MDL/MML principle* [31]:

$$Opt(p) = w_1 \cdot PosCov(p) - w_2 \cdot NegCov(p) \tag{5}$$
$$-w_3 \cdot ProgLength(p) - w_4 \cdot OpersRate(p)$$

by default, $w_1 = 0.4$, $w_2 = 0.3$, $w_3 = 0.1$ and $w_4 = 0.2$.
Finally, the*Global optimality* factor is then calculated as the average of the optimalities of all programs in the system:

$$OptGlobal(P_t) = \frac{1}{Card(P_t)} \sum_{p \in P_t} Opt(p) \tag{6}$$

2. *Average Size of Rules* (ϕ_2): measures the average size of all the rules in R_t. In particular, we compute the size of a rule ρ as in [12]:

$$Size(\rho) = 1 + n_v/2 + n_c + n_f \tag{7}$$

with n_v, n_c and n_f being, respectively, the number of variables, constants and functors of only the rhs of r.
3. *Average Size of programs* (ϕ_3): measures the average cardinality of all the programs in P_t in terms of the number of rules.
4. *Best Rule Optimality* (ϕ_4): is the optimality of the best rule (as unitary program) generated until now.
5. *Best Program Optimality* (ϕ_5): is the optimality of the best program generated until now.

4.2 Decisions

For each iteration of the system, we have to select the rule and operator to produce new rules. Depending on the problem to solve, the number of required iterations to learn a problem could be astronomically high. To address this issue we need a particular heuristic to tame the search space and make good decisions about the choice of rule and operator, in which the application of an operator to a rule is seen as a decision problem.

For that, we model the decision process as a typical reinforcement learning task. Formally, our decision problem is a four-tuple $\langle \mathcal{S}, \mathcal{A}, \tau, \omega \rangle$ where: $\mathcal{S}$ is an infinite state space; $\mathcal{A}$ is a finite actions space ($\mathcal{A} = \mathcal{O} \times \mathcal{R}$); $\tau : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ is a transition function between states and $\omega : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is the reward function. These components are defined below:

– **States.** Each state is described by five features as we have seen in section 4.1.
– **Actions.** An action is a tuple $\langle o, \rho \rangle$ with $\rho \in R$ and $o \in O$ that represents the operator o to be applied to the rule.
– **Transitions.** Transitions are deterministic. A transition τ evolves the current sets of rules and programs by applying the operators selected (together with the rule) and the combiners.
– **Rewards.** The optimality criteria seen above is used to feed the rewards. In particular, we use the result returned by equation (5) as reward.

With all these elements, the aim of our decision process is to find a policy $\pi : S \to A$ that maximises

$$V^{\pi}(s_t) = \sum_{i=0}^{\infty} \gamma^i w_{t+i} \tag{8}$$

for all s_t, where $\gamma \in [0, 1]$ is the *discount parameter* which determines the importance of the future rewards ($\gamma = 0$ only considers current rewards, while $\gamma = 1$ strives for a long-term high reward).

At each point in time, the reinforcement learning policy can be in one of the states s_t of S and selects an action $a_t = \pi(s_t) \in A$ to execute. Executing such action a_t in s_t will change the state into $s_{t+1} = \tau(s_t, a_t)$, and the policy receives a reward $w_t = \omega(s_t, a_t)$. The policy does not know the effects of the actions, i.e. τ and ω are not known by the policy and need to be learned. This is the typical formulation of reinforcement learning [27] but using features to represent the states.

In our setting, for the reinforcement learning module, we use a hybrid between model-free value-function methods (which search for action that maximises values) and model-based methods (which generalise τ and ω) [27]. Our approach uses the *state-value* function ($Q(s, a)$, which returns *q values*) generalising it with a regression model, actually a Linear Regression, where $s \in \mathcal{S}$, $a \in \mathcal{A}$, and finally, the quality values $q \in \mathbb{R}$.

A model $M : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ calculates the optimality or q-value for each state and action. By using $a_t = \arg\max_{a \in \mathcal{A}} \{M(s_t, a_i)\}$ we get the best action for state s_t. Once we have the action, it is carried out to obtain a new state $s_{t+1} = \tau(s_t, a_t)$.

In order to train the model we need to provide different states and actions as an input, and quality values as output. We use *q values* for the state-value function as in Q-learning [32], so to train the model we use a matrix $Q = |S| \times |O| \times |R|$ where S is the set of states reached so far, O is the set of operators and R is the set of rules generated. Both sets, S and R, grow in each step of the system (the number of operators is constant), therefore, the matrix also grows in terms of the number of rows (states) and columns (actions). In Table 1 we can see an example of a Q-matrix that can be used to train our model. Before the system starts, this matrix is initialised with one row (state s_0) with q values equal to 1 for every action (combinations of operators and rules). In this way, the model trained with this matrix will have the same probability of selection for all possible actions at the initial steps of the algorithm.

Table 1. Q-matrix example

state (s)					**action (a)**		**q**
ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	o	ρ	
1.223	1.473	3.431	1.88	1.99	2	12	**0.78**
1.301	1.511	3.431	1.88	1.99	5	27	**0.65**
...							

Once the system has started, at each step, the Q-matrix is updated (as we will see below) and the model can be retrained periodically.

To update each q value in the Q-matrix at each step we use the following formula, as in Q-learning:

$$Q[s_t, a_t] \leftarrow Q[s_t, a_t] + \alpha \times \left[w_{t+1} + \gamma \max_{a_{t+1}} M(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \quad (9)$$

where the max future value is obtained by the model instead of a Q-matrix. α ($\alpha \in [0,1]$) is the *learning rate* which determines to what extent the newly acquired information will override the old information ($\alpha = 0$ makes the agent not to learn anything, while $\alpha = 1$ makes the agent consider only the most recent information); and $\gamma \in [0,1]$ is the discount parameter. By default, $\alpha = 0.5$ and $\gamma = 0.5$.

Using our *Reinforcement Learning* approach, populations of rules and programs are updated at each step of the algorithm. First, the *Rule Generator* process (Figure 1) gets the operator o and the rule ρ returned as an action $a = \langle o, \rho \rangle$ by the *Reinforcement Learning Module* (policy). This process applies the operator over the rule obtaining a new rule ρ' (if the operator is not suitable for the rule selected, the process returns the same rule) which is added to R. The way in which the set programs is evolved is by the *Program Generator* process. This takes the new rule generated ρ' (if appropriate) as input, the set of programs P and the set of combiners C and generates a new program p' (which is added to P) applying the combiners over the previous inputs.

4.3 Stopping Criterion

The process is limited to a maximum number of iterations which is also determined by the user or when the prototype founds the best solution (a program which covers all the positive evidence and and does not cover any negative examples) with a minimum optimality value, whichever comes first.

5 Examples

In this section, we describe three different examples where we illustrate how operators are defined and used to iteratively approach the solution. We also show more details about our system[1] and how it solves these problems[2].

[1] Available at `http://users.dsic.upv.es/~flip/SystemMetaRL.rar`

[2] Available at `http://users.dsic.upv.es/~flip/SystemMetaRLProblems.rar`

5.1 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 data structure of examples (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).

In order to introduce a replacement function, we need more operators, such as $addBK_f$, which fills 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 system solves this problem (with five positive and five negative examples) in 9.58 seconds using 58 iterations.

5.2 Bunch 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 [19]. 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: $opens([[abloy, 3, medium, narrow], [chubb, 6, medium, normal]]) = true$.

Given a set of such examples, we want to learn the function *opens : Bunch* $\rightarrow$ $\{$*True,False*$\}$.For this, we need a function *setExists(Key,Bunch)* which evaluates ($True$ or $False$) whether there exists a *Key* in a *Bunch.* This function will belong to the background knowledge. We also need 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: $addBK(opens(X) = True) \Rightarrow opens(X) \rightarrow setExists([], X)$.

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 system and operators are provided, given the original evidence for this example (five $True$ instances and four $False$ 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. The system solves this problem in 17.88 seconds using 60 iterations.

5.3 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 *Connections* (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: $sportsWeb(Structure, Content, Connections) \rightarrow$

true 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, *Connections* 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: $addBK_{graph}(sportsWeb(S, C, U) \rightarrow True) \;\Rightarrow\; sportsWeb(S, C, U) \rightarrow graphExists(\{[], []\}, S)$, 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 adds them to the condition of the *setExists* function (first attribute) and another operator which generate a node and adds it as a node to search in the graph attribute of the function *graphExists*:

$$\begin{aligned}&linkl_{football}(sportsWeb(S, C, U) \rightarrow graphExists(\{X, Y\}, S))\\ \Rightarrow\;& sportsWeb(S, C, U) \rightarrow graphExists(\{[football|X], Y\}, S).\end{aligned}$$

Note that this operator is parametrised for the different attribute values. Finally, we need a generalisation operator for each input pattern of the rules: $genPat_1(sportsWeb(S, C, U) \rightarrow True) \;\Rightarrow\; sportsWeb(V_S, C, U) \rightarrow True$. There are also some other operators to generalise the second and third arguments.

Our system found the following correct program which defines the *sportsWeb* function:

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

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.* The system solves this problem (with seven positive examples and 2 negative examples) in 19.02 seconds using 42 iterations.

6 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 which otherwise need to be modified when the original data representation and structure changes. In fact, the most general approach can still be found in ILP (or the more general area of inductive programming). However, each system is still specific to a set of embedded operators and heuristics.

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. This carries 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. This can be done by the user, using a language for expressing operators. A generalised operator choice entails generalised heuristics, since the use of different operators precludes the system to use specialised heuristics for each of them. The choice of the wight pair of operator and rule has been reframed as a decision process, as a *reinforcement learning* problem.

We have included some illustrative examples with a first system implementing the general architecture, and we have seen where the flexibility stands out. Our immediate future work is focused on the reuse of operators and heuristics (RL models) across different problems.

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 think that for cocomplex problems that cannot be solved by the system with its predefined operators, the system can be used to investigate which operators are more suitable. In more general terms, this can be used as a system testbed, where we can learn and discover some new properties, limitations and principles for more general machine learning systems that can be used in the future.

References

1. Armstrong, J.: A history of erlang. In: Proceedings of the Third ACM SIGPLAN Conf. on History of Programming Languages, HOPL III, pp. 1–26. ACM (2007)
2. Brazdil, P., Giraud-Carrier: Metalearning: Concepts and systems. In: Metalearning. Cognitive Technologies, pp. 1–10. Springer, Heidelberg (2009)
3. Daumé III, H., Langford, J.: Search-based structured prediction (2009)
4. Dietterich, T., Domingos, P., Getoor, L., Muggleton, S., Tadepalli, P.: Structured machine learning: the next ten years. Machine Learning 73, 3–23 (2008)
5. Dietterich, T.G., Lathrop, R., Lozano-Perez, T.: Solving the multiple-instance problem with axis-parallel rectangles. Artificial Intelligence 89, 31–71 (1997)
6. Džeroski, S.: Towards a general framework for data mining. In: Džeroski, S., Struyf, J. (eds.) KDID 2006. LNCS, vol. 4747, pp. 259–300. Springer, Heidelberg (2007)
7. Dzeroski, S., De Raedt, L., Driessens, K.: Relational reinforcement learning. Machine Learning 43, 7–52 (2001), 10.1023/A:1007694015589
8. Dzeroski, S., Lavrac, N. (eds.): Relational Data Mining. Springer (2001)
9. Estruch, V., Ferri, C., Hernández-Orallo, J., Ramírez-Quintana, M.J.: Similarity functions for structured data. an application to decision trees. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial 10(29), 109–121 (2006)

10. Estruch, V., Ferri, C., Hernández-Orallo, J., Ramírez-Quintana, M.J.: Web categorisation using distance-based decision trees. ENTCS 157(2), 35–40 (2006)
11. Estruch, V., Ferri, C., Hernández-Orallo, J., Ramírez-Quintana, M.J.: Bridging the Gap between Distance and Generalisation. Computational Intelligence (2012)
12. Ferri-Ramírez, C., Hernández-Orallo, J., Ramírez-Quintana, M.J.: Incremental learning of functional logic programs. In: Kuchen, H., Ueda, K. (eds.) FLOPS 2001. LNCS, vol. 2024, pp. 233–247. Springer, Heidelberg (2001)
13. Gärtner, T.: Kernels for Structured Data. PhD thesis, Universitat Bonn (2005)
14. Holland, J.H., Booker, L.B., Colombetti, M., Dorigo, M., Goldberg, D.E., Forrest, S., Riolo, R.L., Smith, R.E., Lanzi, P.L., Stolzmann, W., Wilson, S.W.: What is a learning classifier system? In: Lanzi, P.L., Stolzmann, W., Wilson, S.W. (eds.) IWLCS 1999. LNCS (LNAI), vol. 1813, pp. 3–32. Springer, Heidelberg (2000)
15. Holmes, J.H., Lanzi, P., Stolzmann, W.: Learning classifier systems: New models, successful applications. Information Processing Letters (2002)
16. Kitzelmann, E.: Inductive programming: A survey of program synthesis techniques. In: Schmid, U., Kitzelmann, E., Plasmeijer, R. (eds.) AAIP 2009. LNCS, vol. 5812, pp. 50–73. Springer, Heidelberg (2010)
17. Koller, D., Sahami, M.: Hierarchically classifying documents using very few words. In: Proceedings of the Fourteenth International Conference on Machine Learning, ICML 1997, pp. 170–178. Morgan Kaufmann Publishers Inc., San Francisco (1997)
18. Lafferty, J., McCallum, A.: Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In: ICML 2001, pp. 282–289 (2001)
19. Lloyd, J.W.: Knowledge representation, computation, and learning in higher-order logic (2001)
20. Maes, F., Denoyer, L., Gallinari, P.: Structured prediction with reinforcement learning. Machine Learning Journal 77(2-3), 271–301 (2009)
21. Martínez-Plumed, F., Estruch, V., Ferri, C., Hernández-Orallo, J., Ramírez-Quintana, M.J.: Newton trees. In: Li, J. (ed.) AI 2010. LNCS, vol. 6464, pp. 174–183. Springer, Heidelberg (2010)
22. Muggleton, S.: Inverse entailment and Progol. New Generation Computing (1995)
23. Muggleton, S.H.: Inductive logic programming: Issues, results, and the challenge of learning language in logic. Artificial Intelligence 114(1-2), 283–296 (1999)
24. Plotkin, G.: A note on inductive generalization. Machine Intelligence 5 (1970)
25. Schmidhuber, J.: Optimal ordered problem solver. Maching Learning 54(3), 211–254 (2004)
26. Srinivasan, A.: The Aleph Manual (2004)
27. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (1998)
28. Tadepalli, P., Givan, R., Driessens, K.: Relational reinforcement learning: An overview. In: Proc. of the Workshop on Relational Reinforcement Learning (2004)
29. Tamaddoni-Nezhad, A., Muggleton, S.: A genetic algorithms approach to ILP. In: Matwin, S., Sammut, C. (eds.) ILP 2002. LNCS (LNAI), vol. 2583, pp. 285–300. Springer, Heidelberg (2003)
30. Tsochantaridis, I., Hofmann, T., Joachims, T., Altun, Y.: Support vector machine learning for interdependent and structured output spaces. In: ICML (2004)
31. Wallace, C.S., Dowe, D.L.: Refinements of MDL and MML coding. Comput. J. 42(4), 330–337 (1999)
32. Watkins, C., Dayan, P.: Q-learning. Machine Learning 8, 279–292 (1992)