AND/OR Trees for the Learning of Functional Logic Programs

C. Ferri-Ramírez J. Hernández-Orallo M.J. Ramírez-Quintana

DSIC, UPV, Camino de Vera s/n, 46020 Valencia, Spain.

{cferri,jorallo,mramirez}@dsic.upv.es

Abstract

In this paper, we present the implementation of a method for generating very expressive decision trees over a functional logic language. The method is based on several partitions which allow us the induction of nested and recursive functions and the use of background knowledge. The construction of the decision tree is guided according to the MDL principle. Hence, the search space is covered in a short-to-long fashion. Moreover, a multiple decision tree is obtained rather than a single tree. We show that the method is suitable for classification problems.

Keywords: Decision Trees, Inductive Functional Logic Programming (IFLP), Inductive Logic Programming (ILP), Minimum Description Length (MDL).

1 Introduction

Given a concept t, a decision tree maps a set of examples of t into leaves, such that all the nodes on one level define a partition of the examples, and the children of a node define a partition of the examples covered by that node. In general, each path from the root to a leaf corresponds to a conjunction of conditions, whereas a decision tree represents a disjunction of such conjunctions. Hence, trees can also be represented as sets of conditional rules to improve their readability. Decision trees have been used as classifiers. Most systems (CART [1], ID3 [15], C4.5 [17], FOIL [16]) that induce decision trees construct the tree from the root to the leaves, given a set of cases (examples). Each case is described by a vector of attribute values, which represents a mapping from attribute values to classes. The attributes can be continuous or discrete. Systems for inducing decision trees are usually based on the generation of partitions or splits. Generally, there are two kinds of splits: splits for discrete values and splits for continuous values.

This work has been partially supported by CICYT under grant TIC 98-0445-C03-C1 and Generalitat Valenciana under grant GV00-092-14.

In the Functional Logic Programming (FLP) paradigm, conditional programs are sets of rules, and, hence, they can also be represented as trees. The context of this work is the generation of a conditional FLP program (a hypothesis) from examples (an evidence). Let us consider a simple example of the induction of the *member* function.

Example 1 *Given the evidence:*

$$E^+ = \left\{ \begin{array}{l} e_1 : member(a,\lambda) = false \\ e_2 : member(b, ins(\lambda,a)) = false \\ e_3 : member(c,\lambda) = false \\ e_4 : member(c, ins(\lambda,b)) = false \\ e_5 : member(a, ins(ins(\lambda,b),d)) = false \\ e_6 : member(a, ins(ins(\lambda,b),a)) = true \\ e_7 : member(b, ins(ins(\lambda,b),a)) = true \\ e_8 : member(c, ins(ins(ins(\lambda,b),a),c)) = true \\ e_9 : member(a, ins(ins(ins(\lambda,b),a),b)) = true \\ e_{10} : member(c, ins(\lambda,c)) = true \end{array} \right\}$$

The following program P can be constructed

$$\begin{array}{ll} (i) & member(X,\lambda) = false \\ (ii) & member(X, ins(Z,X)) = true \\ (iii) & member(X, ins(L,W)) = member(X,L) \Leftarrow W \neq X \end{array}$$

Note that the last rule has a disequality in its condition. There are extensions of FLP which are able to handle disequalities as constraints [7, 9]. The following table shows the number of examples that each rule of P covers directly (i.e. examples that can be proven with only this rule) and the number of examples for which that rule is required (i.e. examples that can be proven with this rule jointly with others), respectively.

Rule	Direct Cover	Required
i	2	5
ii	3	5
iii	5	5

Table 1: Number of examples that a rule covers directly and indirectly (it is required).

In the following figure, we show the tree corresponding to the previous program.

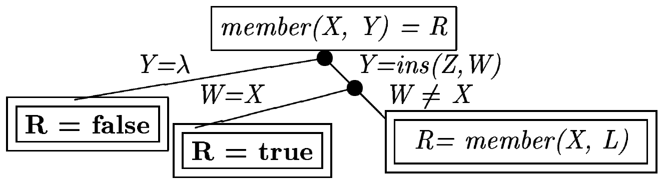


Figure 0: Decision tree for the member problem.

We can represent the tree as a set of program rules as follows:

$$\begin{array}{l} member(X,Y) = R \Leftarrow Y = \lambda, R = false \\ member(X,Y) = R \Leftarrow Y = ins(Z,W), W = X, R = true \\ member(X,Y) = R \Leftarrow Y = ins(Z,W), W \neq X, R = member(X,L) \end{array}$$

For the sake of readability, this program can be easily written as the above program P. Note that in the generation of this tree, the function result has been left as a variable (R) in order to include recursion.

In this paper we introduce a method to construct a decision tree given an evidence, which will result in a (conditional) FLP program. We extend the scope of traditional decision trees taking into consideration the constructor-based data types of the attributes of the function to be learned. Therefore, we include these constructor symbols of the data types as another split criterion. We call this approach *Constructor-based Decision-Tree Learning* (CDTL) since the idea is to use the type information of the functions to generate the trees. For the learning of a target function f, the training sample is composed of ground equations whose left-hand sides are terms of the form $f(\ldots)$ and whose right-hand sides are the function result (the *class* in decision-tree terminology). The root node of the tree is an equation of the form $f(X_1, \ldots, X_{n-1}) = X_n$. Likewise, since we consider functional logic languages, the result of the function (X_n) is also considered as another discrete attribute to be tested in our method. It allows for the induction of nested and recursive functions as well as the use of background knowledge in the definition of the target function. This approach extends the framework for the induction of functional logic languages defined in [6] making it conditional and capable of inducing functions with high arity more efficiently.

The proposed decision-tree algorithm follows a short-to-long search. The split criterion is based on the Minimum Description Length (MDL) principle [18]. It differs from other approaches whose quality criteria are based on discrimination, as in [1, 17]. It also differs in the way the search space (an AND/OR tree) is traversed, producing an increasing number of solutions for increasing provided time.

The paper is organised as follows. In Section 2, we introduce some basic notions about functions and the Inductive Functional Logic Programming (IFLP) framework. Section 3 defines the set of partitions and establishes the criterion used for generating the tree. The algorithm and its implementation is presented in Section 4. Section 5 includes some experimental results. Finally, Section 6 presents the conclusions.

2 Preliminaries and Notation

We briefly review some basic concepts about equations, $\mathcal{E}$-unification and the IFLP framework. Let S be a set (of *sorts*[1], also called *types*). An S-sorted signature Σ is an $S^* \times S$-sorted family $\langle \Sigma_{w,s} | w \in S^*, s \in S \rangle$. $f \in \Sigma_{w,s}$ is a function symbol of arity w and type s; the arity of a function symbol expresses which data sorts it

[1]A sort is a name for a set of objects.

expects to see as input and in what order, and the s expresses the type of data it returns. Also, we consider Σ as the disjoint union $\Sigma = \mathcal{C} \uplus \mathcal{F}$ of symbols $c \in \mathcal{C}$, called *constructors*, and symbols $f \in \mathcal{F}$, called *defined functions*. Let $\mathcal{X}$ be a countably infinite set of *variables*. Then $\mathcal{T}(\Sigma, \mathcal{X})$ denotes the set of *terms* built from Σ and $\mathcal{X}$, and $\mathcal{T}(\mathcal{C}, \mathcal{X})$ is the set of constructor terms. The set of variables occurring in a term t is denoted $Var(t)$. A term t is a *ground term* if $Var(t) = \emptyset$. A *substitution* is defined as a mapping from the set of variables $\mathcal{X}$ into the set of terms $\mathcal{T}(\Sigma, \mathcal{X})$. 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). An equational theory $\mathcal{E}$ (which we call *program*) is a finite set of equational clauses of the form $l = r \Leftarrow e_1, \ldots, e_n$ with $n \geq 0$ where e_i is an equation, $1 \leq i \leq n$. The theory (and the clauses) are called *conditional* if $n > 0$ and *unconditional* if $n = 0$. In what follows, a program P defines a signature Σ_P which is composed by the function symbols that appear in P.

IFLP [6] can be defined as the functional (or equational) extension of ILP. The goal is the inference of a theory (a functional logic program P) from evidence (a set of positive and optionally negative ground equations E) using a background knowledge theory (a functional logic program B). The rhs of the equations in E are normalised w.r.t. the background theory B and to the theory P, which is meant to be discovered (target hypothesis). In [6], an approximation to the induction of unconditional functional logic programs has been presented based on two operators: *Consistent Restricted Generalisation* (CRG) and *Inverse Narrowing*. Informally, a CRG of an equation e is another equation which generalises e without introducing extra variables on the rhs and which is consistent with the evidence.

3 Constructor-Based Decision-Tree Learning

In this section, we define the kinds of splits that we consider for the construction of decision trees. As stated before, our method exploits each data type in a different fashion. Hence, first of all, let us define the possible types we are going to deal with in our approach. Table 2 shows these types.

Kind	Description	Type Example	Attribute Ex.	Order
UDF	Unordered Discrete Finite	$\{red, green, blue\}$	*green*	-
UDI	Unordered Discrete Infinite	*Lists*	$cons(\lambda, a)$	-
ODF	Ordered Discrete Finite	$\{low, med, high\}$	*low*	$<$
ODI	Ordered Discrete Infinite	*Naturals*	4	$<$
C	(Ordered) Continuous (Infinite)	[0.0 ... 360.0]	47.34	$<$
U	Undefined Type	-	-	-
R	Restricted (Dummy) Type [2]	Elements of a List	a	-

Table 2: Kind of types allowed.

For the unordered types, there is no need to know or to give the exact type of an attribute, because the possible values can be obtained from the evidence.

[2]This type is useful for avoiding partitions on types which are irrelevant.

Next, in Table 3, we define the possible partitions (splits) that can be made according to the types (consider an equation $f(X_1,..,X_{n-1}) = X_n$ and $1 \leq i \leq n$). Note that, in IFLP, the attribute tests are expressed as equations.

#	Partition on Attribute X_i (Split)	Kinds of Types Applicable
1	$X_i = a_1 \mid X_i = a_2 \mid \ldots \mid X_i = a_k$	Finite (udf,odf)
2	$X_i = c_0 \mid \ldots \mid X_i = c_k(Y_1, \ldots, Y_{k_m})$	Constructor based (udi) and (u)
3	$X_i < t \mid X_i \geq t$[3] where t is a threshold	Ordered (odf,odi) and Continuous (c)
4	$X_i = Y$ where $Y \in \{X_1, \ldots, X_n\}$ and $Y \neq X_i$[4]	Discrete (udf,udi,odf,odi) and (u,r)
5	$X_i = a \mid X_i \neq a$	udf,odf,odi,c,u
6	$a_1 = f(Y_1, \ldots, Y_n) \mid \ldots \mid a_n = f(Y_1, \ldots, Y_n)$ where $\exists! Y_i \in \{X_1, \ldots, X_n\}$	all
7	$X_i = f(Y_1, \ldots, Y_n)$	all
8	$a_1 = g(Y_1, \ldots, Y_n) \mid \ldots \mid a_n = g(Y_1, \ldots, Y_n) \ldots$ where $\exists! Y_i \in \{X_1, \ldots, X_n\}$ and $g \in \Sigma_B$	all
9	$X_i = g(Y_1, \ldots, Y_n)$ where $g \in \Sigma_B$	all

Table 3: Splits allowed.

All partitions, except 4, 7 and 9, split the example set into disjoint subsets. Note that partitions 4, 7 and 9 have only one child to avoid non-disjoint partitions. The 9 kinds of partitions entail an extension of other systems such as ID3, FOIL and our CRG method, as is shown in Table 4.

#	ID3	FOIL	CRG	CDTL
1	×	×	×	×
2	-	-	×	×
3	×	×	-	×
4	-	×	×	×
5	-	-	-	×
6	-	×	-	×
7	-	-	-	×
8	-	×	-	×
9	-	-	-	×

Table 4: Comparison between splits allowed by CDTL w.r.t. ID3, FOIL, and CRG methods.

With the previous partitions the adaptation of classical split criteria, such as those used by C4.5 / FOIL or CART, would not be suitable, because these measurements are devised to reward partitions which correctly discriminate the class of the result, be it a predicate or a function. However, this may be misleading for recursive functions where this recursive call appears directly on the rhs of a rule; for instance, the lhs of the rule $sum(X, s(Y)) = s(sum(X,Y))$ does not make any bias on the distribution of the result of the function sum. Consequently, a criterion based on this discrimination or on a distribution change would never select the partitions which are needed to generate the previous rule.

[3] $X_i \geq t$ represents a constraint which can be handled by a constraint solver which can solve linear real inequalities ([10]).

[4] There are extensions of the functional logic programming which are able to handle disequalities as constraints ([7, 9]).

As we have stated in the introduction, one proper way to order the search space is by the description length of the hypothesis. By definition, a top-down construction of a decision tree is short-to-long, since it adds conditions and after a partition is made the tree is larger to describe. However, this is not sufficient. The idea is to devise a split criterion such that partitions that presumably lead to shorter trees should be selected first.

There exists a suitable paradigm for performing this search: the MDL principle. If we assume $P(h) = 2^{-K(h)}$ where $K(\cdot)$ is the descriptional (Kolmogorov) complexity of a hypothesis h, and $P(E|h) = 2^{-K(E|h)}$ with E being the evidence, we can obtain the so-called maximum a posteriori (MAP) hypothesis as follows [8].

$$h_{MAP} = argmax_{h \in H} P(h|E) = argmin_{h \in H}(K(h) + K(E|h))$$

This last expression is the MDL principle, which means that the best hypothesis is the one which minimises the sum of the description of the hypothesis and the description of the evidence using the hypothesis.

Initially, when there is only the root of the tree, the hypothesis is empty and the length of its description ($K(\emptyset)$) is almost zero, while the description of the data ($K(E|\emptyset)$) is large. At the end of the construction of the decision tree, the description of the hypothesis $K(h)$ may be large, and each branch constitutes a rule of the program, while the description of the data by using the tree, i.e. $K(E|h)$, will have been reduced considerably. If the resulting tree is good, the term $K(h) + K(E|h)$ is smaller than initially.

The construction of the tree is made in the following greedy way: we select the split with less cost from all possible splits (*split criterion*) and the other nodes remain *suspended*; the generation of the tree stops when all nodes are closed (i.e. the class is consistent with all the examples that fall into that node) (*stop criterion*); for the generation of more solutions, from all the *suspended nodes* (nodes which can be split), we select the node with less relative MDL cost of coding the whole set of examples that fall into it (*node activation criterion*). Hence, the result of this process is a multi-tree rather than a tree. Finally, we also define a *solution tree selection criterion*, in our case, the shortest tree, i.e., the one which minimises $K(h)$.

4 CDTL Algorithm

The approach described in the previous sections has been implemented in a machine learning system for the induction of FLP programs. This application reuses some parts (narrowing solver and parser) of our previous work on the IFLP framework, the FLIP system [2][5].

The initial purpose of this work was to overcome the limitations of the FLIP system. This implementation, based on the IFLP framework described in [5, 6], allows for the learning of functional logic programs from positive and negative evidence, and (optionally) from background knowledge. Although the experimental results were promised, this framework is limited by the use of the CRG generalisation operator because it produces a huge amount of equations from a simple example. For this

[5]This software is public available at [3].

reason, its application for the learning of concepts with a large number of arguments or examples is not suitable. A first attempt to improve the behavior of the system was the extension of the IFLP algorithm in order to make it incremental [4]. In this way, the new version of the FLIP system was able to deal with problems whose evidence was composed by a (considerably) large number of examples. Nevertheless, one of the most interesting fields of application of machine learning is data mining, and (unfortunately) the use of the previous IFLP framework is difficult due to the size of this kind of problems.

Considering these limitations we decided to redesign the IFLP framework by using decision trees as the basis of the learning mechanism. As we have described in the previous section, we have used the MDL principle to guide several phases of the tree generation. For the implementation of the framework we use a tree based on AND/OR graphs [12, 14], a well known structure from Artificial Intelligence. In this tree, the OR-nodes represent the possible splits to be applied at this point (Table 3), whereas the AND-nodes are filled with the branches obtained by the application of the partition defined by its parent OR-node. Once an OR-node is selected to be split (the active node), the set of OR-nodes at the same level are labelled as suspended nodes. Figure 1 illustrates a piece of the AND/OR tree for the *playtennis* example [11], a classical classification problem where the system have to learn if it is possible to play a tennis match depending on weather conditions.

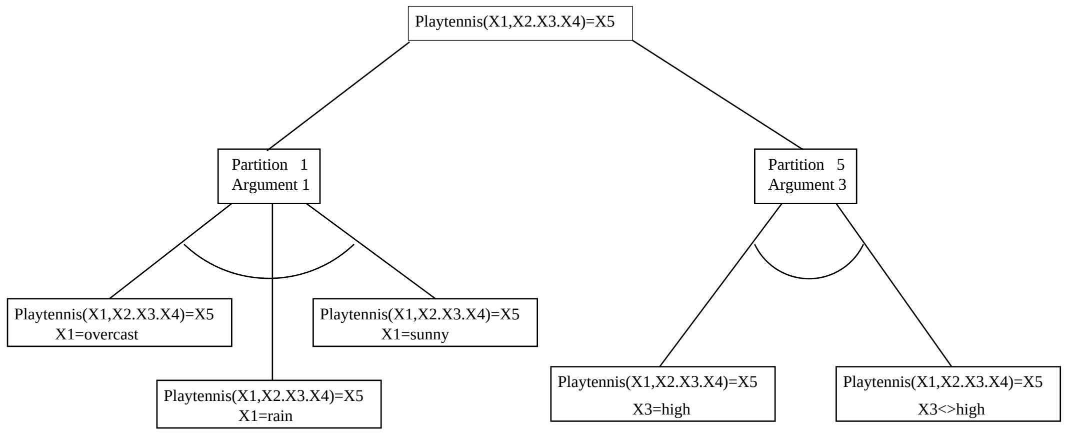

Playtennis(X1,X2.X3.X4)=X5
Partition 1
Argument 1
Partition 5
Argument 3
Playtennis(X1,X2.X3.X4)=X5
X1=overcast
Playtennis(X1,X2.X3.X4)=X5
X1=sunny
Playtennis(X1,X2.X3.X4)=X5
X1=rain
Playtennis(X1,X2.X3.X4)=X5
X3=high
Playtennis(X1,X2.X3.X4)=X5
X3<>high

Figure 1: Section of the AND/OR tree for the *playtennis* example.

The CDTL algorithm is based on a recursive function that develops just the best suspended OR-node for each new tree. To produce more than just one tree, we store the suspended OR-nodes in a list:

```
Procedure GenerateAOtree
Input: atree an AND-node
Output: an AND/OR tree with a solution

    begin
    if $leaf(atree)$
        begin
        $propagate\_costs(atree.father)$
         exit
        end
```

```
$listotree = generate\_partitions(atree)$//build the partitions
$Compute\_cost(listotree)$ //compute the cost of each partition
$ORnode = extract\_best(listotree)$//select the best OR-node suspended
$ORnode.active = true$ //set the selected OR-node active
$suspended\_list.store\_suspended(listotress)$ //store the suspended nodes
$childrenA = ORnode.children$
while $childrenA \neq \emptyset$
  $childA = extract(childrenA)$
  $GenerateAOtree(childA)$
endwhile
end
```

When the first-solution tree is completed, the search of the next solution is restarted exploring the tree from the best suspended OR-node according to the node activation criterion, which selects the suspended node with lowest ratio between the information cost of this node and the information cost of the best node at the same level. This process can be repeated as many times as desired until the complete search space is traversed (the whole AND/OR tree is developed). Nevertheless, it is suitable to limit the number of OR-nodes to exploit through a parameter *Numtree*, because the complete generation of the tree could be very expensive for complex problems.

Therefore, the multi-tree algorithm is the following:

Procedure **CDTL**

```
begin
$init(atree)$//init the tree with an equation with all the arguments open
$GenerateAOtree(atree)$ //call the recursive function
while $Numtree > 0$
  $ORnode = extract\_best(suspended\_list)$//extract the best OR-node
  $childrenA = ORnode.children$
  while $childrenA \neq \emptyset$
    $childA = extract(childrenA)$
    $GenerateAOtree(childA)$
endwhile
  $Numtree = Numtree - 1$
endwhile
$best\_order\_search(atree)$//show the solution
end
```

of OR-nodes The extraction of the solution program must be done by selecting the best tree according to the selection tree criterion, but this process requires that the information of the costs of the nodes hanging from each active OR-node has been updated. This is the reason for the function *propagate_cost*: when a leaf is found, we must propagate cost information to the ancestor nodes.

The *Numtree* value is specified by the user. However, it could be interesting to establish heuristics that could set automatically this value depending on the problem to be learned (i.e. number and size of the function arguments, size of the evidence). Another possibility could be to establish a stop criterion based on the cost of the generated trees. For example, the search could be interrupted when the new trees do not improve the cost of the current best tree in several consequent solutions.

The algorithm presented so far may resemble an A* or an AO* algorithm [12]. These algorithms are guided by an optimistic estimate (a function $h(x)$) of the cost of the rest of the tree to be constructed. If this estimate plus the cost that is carried so far ($g(x)$) is greater than the best $g(x)+h(x)$ of another branch, the search jumps to the other branch and retakes the search. A* and AO* ensure the best solution provided the heuristic function is well-defined. However, this method performs a lot of jumps from side to side in the tree and usually explores a great number of nodes.

In our case, we explore a very limited number of nodes, and we ensure that a solution is output quickly, and the rest of time is devoted to explore the AND/OR tree for better solutions. This allows a more appropriate handling of resources and better response time of the algorithm.

4.1 Example

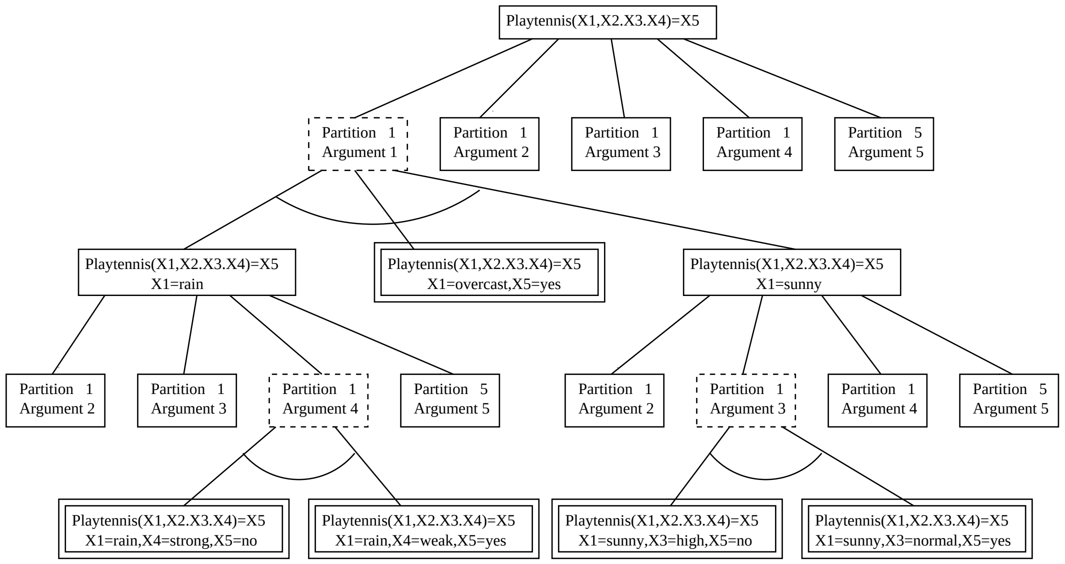
Let us illustrate the previous procedure with the most classical example of decision trees [11]. The purpose is to classify the days in which the weather for playing tennis is good or bad. The evidence is:

$$E = \left\{ \begin{array}{l} e_1 : \mathit{playtennis}(\mathit{overcast}, \mathit{hot}, \mathit{high}, \mathit{weak}) = \mathit{yes} \\ e_2 : \mathit{playtennis}(\mathit{rain}, \mathit{mild}, \mathit{high}, \mathit{weak}) = \mathit{yes} \\ e_3 : \mathit{playtennis}(\mathit{rain}, \mathit{cool}, \mathit{normal}, \mathit{weak}) = \mathit{yes} \\ e_4 : \mathit{playtennis}(\mathit{overcast}, \mathit{cool}, \mathit{normal}, \mathit{strong}) = \mathit{yes} \\ e_5 : \mathit{playtennis}(\mathit{sunny}, \mathit{cool}, \mathit{normal}, \mathit{weak}) = \mathit{yes} \\ e_6 : \mathit{playtennis}(\mathit{rain}, \mathit{mild}, \mathit{normal}, \mathit{weak}) = \mathit{yes} \\ e_7 : \mathit{playtennis}(\mathit{sunny}, \mathit{mild}, \mathit{normal}, \mathit{strong}) = \mathit{yes} \\ e_8 : \mathit{playtennis}(\mathit{overcast}, \mathit{mild}, \mathit{high}, \mathit{strong}) = \mathit{yes} \\ e_9 : \mathit{playtennis}(\mathit{overcast}, \mathit{hot}, \mathit{normal}, \mathit{weak}) = \mathit{yes} \\ e_{10} : \mathit{playtennis}(\mathit{sunny}, \mathit{hot}, \mathit{high}, \mathit{weak}) = \mathit{no} \\ e_{11} : \mathit{playtennis}(\mathit{sunny}, \mathit{hot}, \mathit{high}, \mathit{strong}) = \mathit{no} \\ e_{12} : \mathit{playtennis}(\mathit{rain}, \mathit{cool}, \mathit{normal}, \mathit{strong}) = \mathit{no} \\ e_{13} : \mathit{playtennis}(\mathit{sunny}, \mathit{mild}, \mathit{high}, \mathit{weak}) = \mathit{no} \\ e_{14} : \mathit{playtennis}(\mathit{rain}, \mathit{mild}, \mathit{high}, \mathit{strong}) = \mathit{no} \end{array} \right\}$$

Figure 2 shows the AND/OR tree generated by the CDTL algorithm. The OR-nodes selected are surrounded by a dotted line, and the rest OR-Nodes at the same label are suspended. The AND-nodes doubly surrounded correspond to the leaves of the tree. For the sake of simplicity we only consider partition 1, and we do not show the application of this partition on the output class when the AND-node is a leaf.

The following functional-logic program corresponds to the rules extracted from the AND/OR tree of Figure 2. Note that conditions have been included into the equations. This program covers the whole evidence.

$$\begin{array}{l} \mathit{playt}(\mathit{overcast}, X2, X3, X4) = \mathit{yes} \\ \mathit{playt}(\mathit{rain}, X2, X3, \mathit{weak}) = \mathit{yes} \\ \mathit{playt}(\mathit{rain}, X2, X3, \mathit{strong}) = \mathit{no} \\ \mathit{playt}(\mathit{sunny}, X2, \mathit{normal}, X4) = \mathit{yes} \\ \mathit{playt}(\mathit{sunny}, X2, \mathit{high}, X4) = \mathit{no} \end{array}$$



Playtennis(X1,X2.X3.X4)=X5
Partition 1 Argument 1
Partition 1 Argument 2
Partition 1 Argument 3
Partition 1 Argument 4
Partition 5 Argument 5
Playtennis(X1,X2.X3.X4)=X5 X1=rain
Playtennis(X1,X2.X3.X4)=X5 X1=overcast,X5=yes
Playtennis(X1,X2.X3.X4)=X5 X1=sunny
Partition 1 Argument 2
Partition 1 Argument 3
Partition 1 Argument 4
Partition 5 Argument 5
Partition 1 Argument 2
Partition 1 Argument 3
Partition 1 Argument 4
Partition 5 Argument 5
Playtennis(X1,X2.X3.X4)=X5 X1=rain,X4=strong,X5=no
Playtennis(X1,X2.X3.X4)=X5 X1=rain,X4=weak,X5=yes
Playtennis(X1,X2.X3.X4)=X5 X1=sunny,X3=high,X5=no
Playtennis(X1,X2.X3.X4)=X5 X1=sunny,X3=normal,X5=yes

Figure 2: AND/OR tree for the *playtennis* example.

5 Experiments

We have performed different experiments which show that our multi-tree approach pays off in practice.

We include some examples concerning to the learning of programs for classification problems. All the examples were extracted from the UCI repository [13] and are well-known by the machine learning community. A short description of these problems is included in Table 5.

Problem	Arguments	Classes	Examples	Description
playt	4	2	14	Simple data set that contains the conditions where it is possible to play a tennis match
lenses	4	4	24	Database for fitting contact lenses.
tic-tac-toe	8	2	958	This database encodes the complete set of possible board configurations at the end of tic-tac-toe games.
house-votes	16	2	453	This data set includes votes for each of the U.S. House of Representatives Congressmen on the 16 key votes identified by the CQA in 1984.
cars	6	4	1728	Car Evaluation Database, this model evaluates cars according to a concept structure.
nursery	9	5	12960	Nursery Database was derived from a decision model originally developed to rank applications for nursery schools in Slovenia.

Table 5: Description of the classification problems used in the experiments.

Table 6 contains the results of the experiments: the number of rules of the solution program and the time (in seconds) required for the learning process depending on the

Numtree	1		5		10	
Example	Time	Rules	Time	Rules	Time	Rules
playt	0.05	10	0.17	5	0.23	5
lenses	0.13	10	0.22	10	0.28	9
tic-tac-toe	2.66	398	8.9	344	13.53	344
house-votes	4.47	58	19.86	52	43.73	52
cars	1.67	316	5.31	304	6.82	304
nursery	22.06	1109	87.2	988	161.81	965

Table 6: Time required and number of rules generated in the learning of some classification problems.

Numtree parameter. The experiments were executed on a Pentium III 733 mhz with 128 MB of memory running Linux version 2.2.16. The experiments demonstrate that the system is able to induce programs from a complex evidence (i.e. large number of examples and many parameters), and it is able to deal with large programs too.

Figure 3 shows in detail the induction process depending on *Numtree*. The increase of the number of trees generated allows us to get shorter programs. However, it also influences on the system performance.

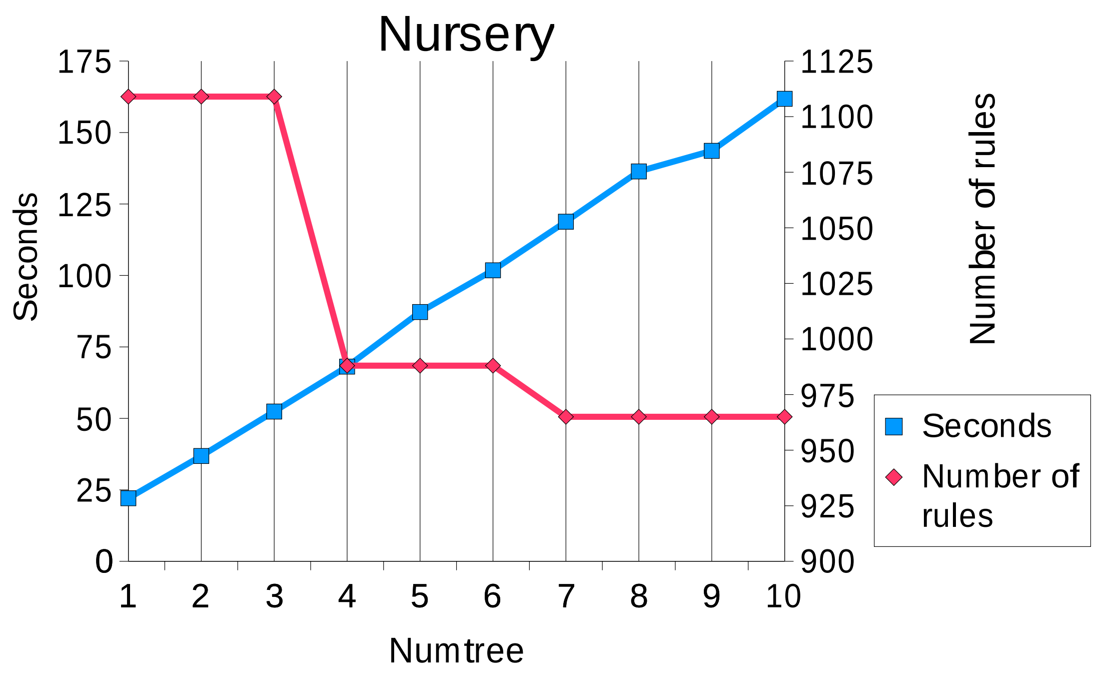


Figure 3: Time required and number of rules generated for the *nursery* problem, depending on *Numtree*.

6 Conclusions and future work

Machine learning provides a very interesting approach for generating programs that are difficult or impossible to be generated by hand. Programs with a large number of rules depending on different values of the arguments, estimated classifiers and any model that must compress a huge evidence composed of ground facts can be addressed by automated inductive techniques.

In this paper we have explored the construction of functional logic trees by the use of different partitions. These partitions are based on non-overlapping conditions. Consequently, the generated programs are confluent. These partitions are optional and allow us to parametrise the kind of programs to be generated: with recursion, with negation, with real numbers constraints, etc., according to the user's needs or the final language that will run the program. Moreover, the search politics permits a better and more customisable handling of resources, depending on the complexity of the problem and the time the user wants to devote for generating the program.

With respect to expressiveness, we plan to better study the treatment of the recursive partitions in order to ensure termination of the programs which are generated. As future work, we are also extending the stop criterion to handle evidence with noise. The MDL principle has been successfully used to prune trees in order to avoid overfitting noisy data. In our case, branches leading to exceptions which are costly to be described should not be exploited in order to produce more general programs where some branches have examples of different classes. Another extension in this sense would be the generation of programs with probabilistic rules.

References

[1] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth Publishing Company, 1984.

[2] C. Ferri, J. Hernández, and M.J. Ramírez. The FLIP system: From theory to implementation. Technical report, Department of Information Systems and Computation, Valencia University of Technology, 2000.

[3] C. Ferri, J. Hernández, and M.J. Ramírez. The FLIP system homepage. `http://www.dsic.upv.es/~flip/`, 2001.

[4] C. Ferri, J. Hernández, and M.J. Ramírez. Incremental Learning of Functional Logic Programs. In *Proc. of the Fifth Int. Symposium on Functional and Logic Programming*, pages 231–247, 2001.

[5] J. Hernández and M.J. Ramírez. Inverse Narrowing for the Induction of Functional Logic Programs. In *Proc. Joint Conference on Declarative Programming, APPIA-GULP-PRODE'98*, pages 379–393, 1998.

[6] J. Hernández and M.J. Ramírez. A Strong Complete Schema for Inductive Functional Logic Programming. In *Proc. of the Ninth International Workshop on Inductive Logic Programming, ILP'99*, volume 1634 of *LNAI*, pages 116–127, 1999.

[7] H. Kuchen, F. Lopez-Fraguas, J. J. Moreno-Navarro, and M. Rodriguez-Artalejo. Implementing a Lazy Functional Logic Language with Disequality Constraints. In *Joint Int. Conf. and Symp. on Logic Prog.*, pages 207–224. MIT Press, 1992.

[8] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications.* 2nd Ed. Springer-Verlag, 1997.

[9] F.J. López-Fraguas. A general scheme for constraint functional logic programming. In H. Kirchner and G. Levi, editors, *Proc. of the Third Int'l Conf. on Algebraic and Logic Programming ALP'92*, volume 632 of *Lecture Notes in Computer Science*, pages 213–227. Springer-Verlag, 1992.

[10] W. Lux. Adding linear constraints over real numbers to curry. In *Proc. of the 5th Int'l Symp. on Functional and Logic Programming FLOPS'01*, volume 2024 of *Lecture Notes in Computer Science*, pages 185–200. Springer-Verlag, 2001.

[11] T. M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[12] N.J. Nilsson. *Artficial Intelligence: a new synthesis.* Morgan Kaufmann, 1998.

[13] University of California. UCI Machine Learning Repository Content Summary. `http://www.ics.uci.edu/~mlearn/MLSummary.html`.

[14] J. Pearl. *Heuristics: Intelligence search strategies for computer problem solving.* Addison-Wesley, 1985.

[15] J. R. Quinlan. Induction of Decision Trees. In *Readings in Machine Learning.* Morgan Kaufmann, 1990.

[16] J. R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5(3):239–266, 1990.

[17] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.

[18] J. Rissanen. Modelling by the shortest data description. *Automatica-J.IFAC*, 14:465–471, 1978.