Learning MDL-guided Decision Trees for Constructor-Based Languages *

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 a method for generating very expressive decision trees based on several partitions over a functional logic language. First, trees are generated top-down and partitions are selected according to the MDL principle. Secondly, not only is a decision tree obtained, but multiple decision trees can be guided as well by the MDL principle. This means that the overall search space is scoured in a short-to-long fashion, thus allowing for a better use of computational resources. The most important result of this paper is a framework for the efficient generation of constructor-based models.

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

1 Introduction

Decision trees are among the most popular tools for machine learning and data mining. There are several reasons for this. First, they are an intuitive representation of the concept that is to be learned. Second, the trees can be constructed relatively fast compared to other methods. Nonetheless, their greedy behaviour together with the absence of backtracking may result in the exploration of partitions leading to bad solutions. However, exploring the entire search space would not be feasible. It is then necessary to sort out the search space in order to first obtain the best subset of trees.

The top-down induction of decision trees is usually performed in two phases: the *building* phase and the *post-pruning* phase. In the first phase, a decision tree is grown by partitioning the input data into two or more subsets using a condition on an attribute (the *splitting attribute*) until a *stopping criterion* holds. Then, the pruning phase may remove some nodes to prevent *overfitting*. Some of these algorithms are CART [2], ID3 [16], C4.5 [18] and FOIL [17].

However, in the ILP framework, the existing decision-tree algorithms are not capable of dealing directly with attributes whose values belong to a constructor-based data type, such as the natural numbers defined by the 0 and $s(\_)$[1] construc-

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

[1] The s symbol stands for the successor function symbol.

tor symbols, or lists defined by the constructor symbols *emptylist* and $\cdot(\_,\_)$[2]. This is due to the fact that the type definition of an attribute is not considered as a splitting criterion. This is usually overcome by the use of background predicates such as $suc(X, Y)$, $member(X, Y)$, etc., but this can only be solved when there are few types and they are known a priori. Nevertheless, constructor-based types are not only interesting in the definition of functions but are essential for certain applications[3].

In this paper, we present a top-down method for inducing decision trees in a functional logic inductive framework (IFLP) [5]. 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 $f(X_1, \ldots, X_{n-1}) = X_n$. The attribute splits proposed here include tests on data types, on recursive definitions or on the definition of an attribute as the result of a call to a background knowledge function. Since we consider that the class is another attribute to be tested, the above split conditions allow for the induction of nested and recursive functions as well as the use of background knowledge in the definition of the target function. In this way, this algorithm extends the IFLP framework making it conditional and capable of inducing functions with a high arity more efficiently.

Unlike other decision-tree induction algorithms, for which general-to-specific or specific-to-general searches are used, our approach follows a short-to-long search. Hence, the splitting criterion is based on the Minimum Description Length (MDL) principle. The MDL principle has been previously used in the induction of decision trees in the post-pruning phase [11, 21]. Also, the MDL principle has been used as a stopping criterion (*pre-pruning*) [15, 17], as a measure for globally evaluating discretizations of continuous attributes [14], and for restructuring decision trees [13]. In our approach, the MDL principle is used at the generation phase which is justified because other quality criteria based on discrimination such as the *information gain* [18], the *information gain ratio* [18] or the *Gini heuristic* [2] are not useful for problems such as $f(\ldots) = f(\ldots)$ or $f(\ldots) = g(\ldots)$ with g being a function of the background knowledge. Another reason is that the guidance of the search by the MDL principle ensures a better use of computational resources [7, 22]. On the other hand, the majority of decision-tree induction algorithms perform a greedy search in order to find the best-first solution. We also perform a greedy search for each solution, but we are able to obtain N solutions and therefore the method is not so prone to getting

[2] In functional programming and functional logic programming, it is usual to define a function as a program whose equations establish the function behavior w.r.t. the constructor terms of (some of) its arguments.

[3] For instance, XML documents are represented as terms of constructor-based types and the handling of these documents requires being able to work with their structure, usually without background knowledge to 'navigate' inside them.

blocked by a bad choice. Each new solution is built following the construction of a complete tree by selecting nodes to be split according to our measure. It differs from other approaches such as the boosting method [4, 19] which induces a new decision tree for each solution.

The paper is organised as follows. In Section 2, we introduce some basic notions about functions and the IFLP framework. Section 3 defines the set of partitions and compares them with those of some well-known algorithms. Several criteria for guiding the search are presented in Section 4. Some considerations about unary partitions are analysed in Section 5. 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*[4], 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 expects to see as input and in what order, and the s expresses the type of data it returns. Given $f \in \Sigma_{(s_1,\ldots,s_n),s}$, we define the following auxiliary concepts:

- $Arity(f) = n$.
- $Arity^t(f)$ denotes the times that the type t is in the string $(s_1, \ldots, s_n)$.
- $Type(f) = s$.
- $Set\_Type(f) = \{s_i, 1 \leq i \leq n\}$

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. We extend the $Type$ function defined above to (sub)terms in a natural way: given a term $f(t_1, \ldots, t_n)$ where $f \in \Sigma_{(s_1,\ldots,s_n),s}$, then $Type(t_i) = s_i$. The set of variables occurring in a term t is denoted $Var(t)$. A term t is a *ground term* if $Var(t) = \emptyset$. Given a set of syntactic objects (like terms, variables or function symbols) $O = \{O_1, \ldots, O_n\}$, we define $O^s = \{O_i | Type(O_i) = s\}$. If S is a set of types, then $O^S = \bigcup_{s \in S} O^s$. 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 [5] 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

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

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 [5] 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

The problem of learning decision-tree classifiers lies in generating a decision tree from a set of cases. Each one is described by a vector of attribute values, and constructs a mapping from attribute values to classes. The attributes can be continuous or discrete, whereas we consider that the class may have only discrete values.

In particular, the most popular systems for learning decision-trees, ID3/C4.5 [16, 18, 20] perform the following tests in order to split a tree node:

- If the attribute A is of a discrete type T, the node has as many outcomes (children) as possible values v_i of T, with condition $A = v_i$.
- If the attribute A is of a continuous type T, the node has two outcomes (children), $A < t$ and $A \geq t$ where t is a threshold that maximises the splitting criterion.

A propositional decision-tree learning can be constructed with these possible splits and a splitting criterion (which tells which tests are chosen first). In general, decision trees represent a disjunction of conditions over the attribute values. Hence, learned trees can also be represented as sets of conditional rules to improve their readability.

Some ILP systems, like FOIL [17] or TILDE [1], allow us to introduce recursion in the definition tree by flattening the predicates; however, they do not allow us to deal with problems with (semi)-structured data based on constructors.

Given a node ν of a decision tree, we denote its set of conditions by C_ν. The Boolean function $leaf(\nu)$ returns true if ν is a leaf of the tree; $\Pi(\nu)$ denotes the partition in ν; $child_i(\nu)$ represents the i-th child of ν and $range(\nu)$ is the number of children of ν. By E_ν let us denote the set of examples which are consistent with the conditions C_ν. The open variables OV_ν of ν are the variables that do not appear in the lhs of an equality of a condition of C_ν. By OVR_ν let us denote the set of variables of real type in OV_ν. $OVNR_\nu = OV_\nu - OVR_\nu$. By τ_i, we denote each different type of the set $OVNR_\nu$.

First of all, let us define the possible types we are going to deal with in our approach. Table 1 shows these types.

Next, in Table 2, 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.

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)	$[0.0 \ldots 360.0]$	47.34	$<$
U	Undefined Type	-	-	-
R	Restricted (Dummy) Type [5]	Elements of a List	a	-

Table 1. Kind of types allowed

#	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	$X_i < t \mid X_i \geq t$[6]where t is a threshold	Ordered (odf,odi) and Continuous (c)
4	$X_i = Y$ where $Y \in \{X_1, \ldots, X_n\}$ and $Y \neq X_i$	Discrete (udf,udi,odf,odi) and (u,r)
5	$X_i = a \mid X_i \neq a$[7]	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 2. 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. Since the variables in conditions are existentially quantified, different children might give a non-disjoint partition. These kinds of partitions entail an extension of ID3, FOIL and our CRG method, as is shown in Table 3.

4 Guiding the Search

The previous partitions have to be well handled in order to convert such a variety of conditions and constructors into something useful. The adaptation of classical splitting 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

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

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

[7] There are extensions of the functional logic programming which are able to handle disequalities as constraints ([6, 9]).

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

Table 3. Comparison between splits allowed by CDTL w.r.t. ID3, FOIL, and CRG methods

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.

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 longer to describe. However, this is not sufficient. The idea is to devise a splitting 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 h, and $P(E|h) = 2^{-K(E|h)}$, 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. The MDL principle also has been justified for first-order theories (see e.g. [3]).

The idea is that the two terms to be minimised in the expression of the MDL principle nicely represent the source and the sink of the information during the construction of a decision tree. Initially, when there is only the root of the tree, i.e. the function profile $f(X_1, X_2, ..., X_{n-1}) = X_n$, the length of the description of the hypothesis $K(h)$ is almost zero (just the profile) while the description of the data $K(E|h)$ is large, because the instance for each variable has to be provided for each example, since no pattern of the evidence has yet been captured. 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 way to construct the tree is to select those partitions (whose description will swell the $K(h)$ part) so that the $K(E|h)$ is reduced considerably. In other words, the best partition will be the one which minimises the term $K(h)+K(E|h)$ after the partition.

In what follows, we will introduce an approximation for $K(E|h)$ and then an approximation for $K(h)$.

4.1 Estimate for $K(E|h)$

Given a node ν, a table T_i is constructed for each different type τ_i of the set $OVNR_\nu$. The table contains an entry for each different term of type τ_i which appeared in the evidence. With $|T_i|$, we denote the number of elements in table T_i. Each term is denoted by $term_{i,j}$, with j ranging from 1 to $|T_i|$. The information required for each entry in the table $Info(term_{i,j})$ is defined in the following way:

- for finite discrete types, $Info(term_{i,j}) = \log n$ with n being the number of possible values of the type[8].
- for constructor-based types, $Info(term_{i,j})$ is defined as the cost in bits of selecting the appropriate constructors (from the possible set of constructors applicable at each moment) to describe the term. For instance, given the list of naturals $[s(0), s(s(s(0))), 0]$, the information of this term is $\log 2$ (for choosing between the empty list or the constructor $\cdot(\_,\_)$) + $\log 2$ (for choosing between 0 and $s(\_)$) + $\log 2$ (for choosing between 0 and $s(\_)$ again) + $\log 2$ (for choosing between the empty list or the constructor $\cdot(\_,\_)$) + ... In the end, 4 (the list has three elements) + 2 for the first natural, 4 for the second one and 1 for the last natural, i.e. 11 bits.

From here, we can define the information which is required to describe the table:

$$InfoTable(T_i) = T_i + \sum_{j=1..|T_i|} Info(term_{i,j})$$

Note that a table is constructed for each different type, not for each different non-real open variable ($OVNR$).

Next we have to give a definition for the information required to give values to all the open variables in order to describe an example. Given an example e from E_ν, we have to code the substitution for non-real variables (just referring to the position in its corresponding table) and the substitution for real (continuous) variables in a different way. Let us denote the argument k of the lhs of example e of real type as $RealValue_k(e)$, and let us denote the integer part of r as $int(r)$ and the digits of the non-integer part of r in binary representation as $rat(r)$. We approximate the cost in bits for describing the real number as $InfoR(r) = 1 + \log int(r) + rat(r)$. The first bit is for the sign. For instance, the number 36.25, which can be represented as 10010.01 in binary notation has $InfoR = 1 + \log 36 + 2 = 8.17$ bits. Type ODI (e.g. integers) is made in a similar way (but only for the integer part). Another possibility would be the use of a constant value for real numbers described as floating-point format.

[8] All logarithms in this paper are binary logarithms.

#	Partition on Attribute X (Split)	Info
1	$X_i = a_1 \mid ... \mid X_i = a_k$	$\log OV_\nu$
2	$X_i = c_0 \mid ... \mid X_i = c_k(Y_l, ..., Y_{mi})$	$\log OV_\nu$
3	$X_i < t \mid X_i \geq t$	$\log OV_\nu + InfoR(t)$
4	$X_i = X_j i \neq j$	$1^* + \log OV_\nu + \log(OV_\nu - 1)$
5	$X_i = a \mid X_i \neq a$	$\log OV_\nu + \log TCard(Type(X_i))$
6	$a_1 = f(...X_i...) \mid ... \mid a_k = f(...X_i...)$	$1 + min[\log OV_\nu + \log Arity^{Type(X_i)}(f),$ $\log OV_\nu^{Type(f)} + \log Arity(f)]$
7	$X_i = f(Y_1, ..., Y_n)$	$1^* + \log OV_\nu^{SetType(f)}$
8	$a_1 = g(...X_i...) \mid ... \mid a_k = g(...X_i...)$	$1 + min[\log OV_\nu + \log Arity^{Type(X_i)}(g),$ $\log OV_\nu^{Type(g)} + \log Arity(g)] + \log \lvert \Sigma_B^\tau \rvert$
9	$X_i = g(Y_1, ..., Y_n)$	$1^* + 1 + min[\log OV_\nu + \log \lvert \Sigma_B^{Type(X_i)} \rvert,$ $\log OV_\nu^{Type(g)} + \log \lvert \Sigma_B^\tau \rvert]$

Table 4. Information corresponding to splits allowed

From here, we can define the information which is required to code an example, given the node ν and using the tables, as:

$$Info(e|\nu) = \sum_{k \in OVNR_\nu} \log |T_i| + \sum_{k \in OVR_\nu} (1 + InfoR(RealValue_k(e)))$$

Note that the first part is the same for all the examples.

And finally, we can define the cost of coding the whole set of examples that fall under ν, i.e. E_ν, as:

$$Info(E_\nu|\nu) = |E_\nu| + \sum_i InfoTable(T_i) + \sum_{e \in E_\nu} Info(e|\nu)$$

The first term codes the number of examples that will be described. An example is included in the appendices.

With this, we have an estimate for the second term of the definition of the MDL principle, $K(h) + K(E|h)$. In this way, $K(E|h) \approx \sum_{\nu \in leaves(h)} Info(E_\nu|\nu)$.

4.2 Estimate for $K(h)$

Table 4 includes the cost in bits of each partition at a node ν, denoted by $InfoP(P, \nu)$. Note that unary partitions 4, 7 and 9 have an extra bit, denoted by 1^*. From this table, given a partition P, its information can be obtained as:

$$InfoPart(P, \nu) = \log 10 + InfoP(P, \nu)$$

The first term is used to select the partition from the 9 possible partitions (the tenth option corresponds to no split, i.e. a leaf). Note that leaves also have $InfoPart$, which is equal to log 10, because the node cannot be exploited further.

And $K(h)$ can be estimated as follows:

$$K(h) \approx InfoHead(h) + \sum_{\nu \in nodes(h)} InfoPart(\pi(\nu), \nu)$$

where $InfoHead(h)$ captures the information which is required to code the profile of the function to be learned (which must include the arity and types of the function).

4.3 Information of the Tree

Finally, we have to introduce the information for the whole tree. Given a tree T with root node ν, the info of T is defined as:

$$InfoTree(E,T) = InfoHead(T) + InfoN(E,\nu)$$

where $InfoN(E,\nu)$ is obtained recursively in the following way:

$$InfoN(E,\nu) = \begin{cases} Info(E_\nu|\nu) & \text{if } leaf(\nu) \\ InfoPart(\pi(\nu)) + \sum\limits_{i=1..range(\nu)} InfoN(E, child_i(\nu)) & \text{otherwise} \end{cases}$$

Initially, the tree with just one node only has information about the function profile. Since this node is a leaf, $InfoTree(E,T) = InfoHead(T) + Info(E_\nu|\nu)$. When the tree is being constructed, any exploited node ν (which is still a leaf) has an approximate value for the information, given by $Info(E_\nu|\nu)$, independently of the possible partitions that there could be underneath. However, when this node is exploited, then the value is substituted by the sum of the information of the partition and the information required for the children subtrees.

Since the first approximations are useful when populating and pruning the tree, we will denote the $InfoN$ of a node up to depth d by $InfoN_d(E,\nu)$. Obviously $InfoN_\infty(E,\nu) = InfoN(E,\nu)$ whereas $InfoN_0(E,\nu) = Info(E_\nu|\nu)$.

4.4 Constructing the Multi-Tree

Now we can establish the way in which one tree is constructed from the root $f(X_1, X_2, ..., X_{n-1}) = X_n$, which is an open node:

- NODE SELECTION CRITERION: From all the open nodes, we select the node with less $InfoN_0(E,\nu)$, i.e. with less $Info(E_\nu|\nu)$. This usually corresponds to nodes with less examples.
- SPLIT SELECTION CRITERION: The $InfoN_1(\nu)$ of the node ν is determined for every possible split. The split with less $InfoN_1$ is selected. Its children are new open nodes.
- STOPPING CRITERION: a node is closed when the class is consistent with all the examples that fall into that node (i.e. E_ν). The generation of the tree stops when all nodes are closed.

In some cases, as any greedy method, an initial choice could result in a tree that is not optimal. Nonetheless, the contrary strategy, to explore all the possible splits, would not be feasible. However, a trade-off can be made. After a tree is finished, we will explore second-best splits in accordance with space-time resources. In other words, *CDTL populates the tree up to a limit number of nodes or time.* To do this, we have to establish a new criterion:

– TREE SELECTION CRITERION: Consider the set of possible splits at depth 1 of a node ν, where σ_1 is the best split and σ_k is the best split that has not yet been exploited. Let us denote the node with split σ_1 as ν_1. Let us denote the node with split σ_k as ν_k. We define the 'rival ratio' $\rho(\nu) = InfoN_1(\nu_1)/InfoN_1(\nu_k)$. Once a tree has been completely constructed, the next tree can be explored, beginning with the split with the greatest rival ratio. This next tree has to be fully completed before selecting another tree.

The result of this process is a **multi-tree** rather than a tree. Note that the use of information approximations at depth 1 prevents the overexplotation of splits at the bottom of the tree, where the information is usually reduced when good selections have been made. This generates a wider variety of trees than would be generated if the real value (depth ∞) were used instead.

The use of a multi-tree which is populated in a short-to-long way has some advantages:

– Each set of leaves of a solution is a program that covers the evidence.
– The memory resources and time invested can be better adjusted.
– The procedure generates several solutions. The MDL principle ($K(h) + K(E/h)$) or Occam's Razor ($K(h)$) can be applied to select the best solution.

Moreover, these hypotheses could be combined to give more precise predictions. According to the MAP hypothesis, we can use the previous estimates for the three a priori probabilities in the Bayes theorem:

$$P(E|h) = 2^{-K(E|h)} \approx 2^{-\sum_{\nu \in leaves(h)} Info(E_\nu|\nu)}$$

$$P(h) = 2^{-K(h)} \approx 2^{-[InfoHead(h) + \sum_{\nu \in nodes(h)} InfoPart(\pi(\nu),\nu)]} = 2^{-InfoTree(\emptyset|h)}$$

$$P(E) = P(E|h_0) = 2^{-K(E|h_0)} \approx 2^{-InfoTree(E|h_0)}$$

where h_0 represents the empty hypothesis (in our case, the tree with one node, the function profile). This gives an approximation for the a posteriori probability. The use of $P(h|E)$ allows for the combination of the different hypotheses provided by a multi-tree to give more accurate predictions (e.g. through weighted majority) and to give estimates for the reliability of a prediction.

5 Coding Exceptions of Unary Partitions

As stated, some partitions (4, 7 and 9) have only one child. The reason is that the complementary sibling node could not be evaluated under the usual semantics of functional logic programs. With only one child, we prevent the generation of inappropriate programs.

However, the existence of just one child of a node ν (which carries along with it an extra condition) can involve the fact that some examples are left uncovered (denoted by $E_{u,\nu}$), and the tree would not describe the entire evidence. These cases should not be avoided, because some unary partitions can be good for describing most of the evidence. Moreover, case 7, which is evaluated w.r.t. the evidence, could, in principle, leave uncovered examples, but when the base rules of the programs are discovered, these uncovered examples could fall into these nodes. The idea is to code these exceptions. To do this, partitions 4, 7 and 9 have an extra bit. If this bit is set to one, then the uncovered examples of the partition $E_{u,\nu}$ will be coded by using the parent node ν and its conditions by the previous equation, i.e.

$$Info(E_{u,\nu}|\nu) = |E_{u,\nu}| + \sum_i InfoTable(T_i) + \sum_{e \in E_{u,\nu}} Info(e|\nu)$$

Background knowledge is evaluated intensionally. Recursive calls to the function which is being learned are treated both extensionally and intensionally. Extensional evaluation uses the entire evidence (such as FOIL); the $Info$ is obtained by using the substitutions for the example which minimises the information. For the intensional evaluation the closed nodes can already be used (in a similar way as FOIDL [12]). The node selection criterion favours the appearance of base cases first. Note that this evaluation is limited and can leave some examples uncovered. Nonetheless, when the tree is completed, some uncovered examples could be recovered. The web page `http://www.dsic.upv.es/~flip` contains examples of the learning process of simple problems by the CDTL mechanism.

6 Conclusions

In this paper, we have presented a new splitting criterion, a node-selection criterion, and a tree-selection criterion, all of which are based on the MDL principle. These criteria allow us to deal with partitions with constructor-based types.

The MDL-guided search has three important advantages. First, MDL sorts out the search space providing an optimal use of computational resources. Secondly, it gives an estimate for $P(E|h)$, which allows for a weighted combination of the different hypotheses obtained from the multi-tree. Last, the annotated *Infos* can be used directly to prune the trees for noisy evidence.

As future work, the expressiveness of the approach could be extended by allowing real numbers in the class, i.e. the result of the function. To do this, the cost should be modified in order to code the description of the error in such a way that the cost would increase for growing quadratic error.

References

1. H. Blockeel. Top-down Induction of First Order Logical Decision Trees. *AI Communications*, 12:119–20, 1999.
2. Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth Publishing Company, 1984.
3. D. Conklin and I.H. Witten. Complexity-Based Induction. *Machine Learning*, 16:203–225, 1994.

4. Y. Freund and R.E. Schapire. Experiments with a new boosting algorithm. In *Proceedings of the Thirteenth International Conference on Machine Learning (ICML'1996)*, pages 148–156. Morgan Kaufmann, 1996.
5. 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.
6. 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.
7. L.A. Levin. Universal Search Problems. *Problems Inform. Transmission*, 9:265–266, 1973.
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. M. Mehta, J. Rissanen, and R. Agrawal. MDL-Based Decision Tree Pruning. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining (KDD'95)*, pages 216–221, 1995.
12. R. J. Mooney and M. E. Califf. Induction of First–Order Decision Lists: Results on Learning the Past Tense of English verbs. *J. of A.I. Research*, 3:1–24, 1995.
13. N.C. Berkman P.E. Utgoff and J.A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(1):5–44, 1997.
14. B. Pfahringer. Compression-based discretization of continuous attributes. In *Proc. 12th International Conference on Machine Learning*, pages 456–463. Morgan Kaufmann, 1995.
15. B. Pfahringer. A new MDL measure for robust rule induction. In N. Lavrač and S. Wrobel, editors, *Proceedings of the 8th European Conference on Machine Learning*, volume 912 of *LNAI*, pages 331–334, Berlin, 1995. Springer.
16. J. R. Quinlan. Induction of Decision Trees. In *Readings in Machine Learning*. Morgan Kaufmann, 1990.
17. J. R. Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5(3):239–266, 1990.
18. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.
19. J. R. Quinlan. Bagging, Boosting, and C4.5. In *Proc. of the Thirteenth Nat. Conf. on A.I. and the Eighth Innovative Applications of A.I. Conference*, pages 725–730. AAAI Press / MIT Press, 1996.
20. J. R. Quinlan. Improved Use of Continuous Attributes in C4.5. *Journal of Artificial Intelligence*, 4:77–90, 1996.
21. J. R. Quinlan and R. L. Rivest. Inferring Decision Trees Using The Minimum Description Length Principle. *Information and Computation*, 80:227–248, 1989.
22. Jürgen Schmidhuber, Jieyu Zhao, and Marco Wiering. Shifting Inductive Bias with Success-Story Algorithm, Adaptive Levin Search, and Incremental Self-Improvement. *Machine Learning*, 28:105–130, 1997.