Induction of Decision Multi-trees using Levin search *

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 over a functional logic language. The generation of the tree follows a short-to-long search which is guided by the MDL principle. Once a solution is found, the construction of the tree goes on in order to obtain more solutions ordered as well by description length. The result is a multi-tree which is populated taking into consideration computational resources according to a Levin search. Some experiments show that the method pays off in practice.
Keywords: Machine Learning, Decision-tree Induction, Inductive Logic Programming (ILP), Levin search, Minimum Description Length (MDL).

1 Introduction

In a classification problem, the goal is to produce a model that will predict the class of future examples with high accuracy from a set of training examples (cases). 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, whereas we consider that the class can have only discrete values. One of the most known classification methods is based on the construction of decision trees. In a decision tree, each node contains a test on an attribute, each branch from a node represents a possible outcome of the test, and each leaf contains a class prediction. A decision tree is usually induced by recursively replacing leaves by test nodes, starting at the root. Classic decision-tree learners such as CART [2], ID3 [16], C4.5 [18] or FOIL [17] have given good results; however, they do not have enough flexibility to trading result quality with processing cost.

In this paper, we present an algorithm for the induction of decision trees which is able to obtain more than one solution, looking for the best one or combining them in order to improve the accuracy. To do this, once a node has been selected to be split (an AND node) the other possible splits at this point (OR nodes) are suspended until a new solution is required. In this way, the search space is an AND/OR tree [12] which is traversed producing an increasing number of solutions for increasing provided time. Since each new solution is

* This work has been partially supported by CICYT under grant TIC 2001-2705-C03-01, Generalitat Valenciana under grant GV00-092-14, and Acción Integrada hispano-italiana HI2000-0161

built following the construction of a complete tree, our method differs from other approaches such as the boosting method [6, 19] which induces a new decision tree for each solution. The result is a multi-tree rather than a tree. We perform a greedy search for each solution, but once the first solution is found the following ones can be obtained taking into consideration a limited computation time. Therefore, our algorithm can be considered *anytime* in a certain way [3].

We focus on functional logic languages as representation languages. In the Functional Logic Programming (FLP) paradigm [7], 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. This allows us to include the type information of the function profile in the split criterion. On the other hand, since a function f is defined by equations of the form $f(X_1, \ldots, X_n) = Y$, we consider the function result Y as another attribute to be tested. All this extends the kind of tests performed in classical decision-tree induction approaches.

Our method uses a heuristic based on the Minimum Description Length (MDL) principle [21]. Hence, the decision tree is built in a short-to-long way. The MDL principle has been previously used in the induction of decision trees but just within the post-pruning phase [11, 20]. Also, the MDL principle has been used as a stopping criterion (*pre-pruning*) [17], as a measure for globally evaluating discretisations of continuous attributes [15], and for restructuring decision trees [14]. 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 functions that have a recursive definition or that use concepts of the background knowledge. Another reason is that the guidance of the search by the MDL principle ensures a better use of computational resources following a Levin search [9]. We use the MDL principle as split criterion, as stopping criterion and also as solution tree selection criterion. In this way, we present a uniform framework based on the same measure for constructing the tree, selecting the split, selecting second-best trees to explore and selecting or combining hypotheses.

The paper is organised as follows. Section 2 presents our method for constructing decission trees defining new criteria. Section 3 illustrates the construction of decision multi-trees and presents a way of combining the obtained solutions in order to give more precise predictions. Some experimental results are shown in Section 4. Finally, Section 5 concludes the paper.

2 Decision Trees by Levin search

For the construction of a decision tree we consider 9 kinds of partitions [5], including the comparison of an attribute to a variable, the inclusion of constructor based types, the split over real numbers, the equality between variables, the introduction of a function from the background knowledge and recursive calls.

With these kinds of 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 right hand side of a 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 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 guiding the construction of the tree: 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 [10]:

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

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 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. Once a tree is finished, we will explore second-best splits in accordance with space-time resources. Therefore, *we populate the tree up to a limit number of nodes or time.* The result of this process behaves as a Levin search since solutions are found in a short-to-long fashion. The Levin search guarantees the optimal order of computational complexity [9] or, more precisely, it is the fastest method for inverting functions save for a large multiplicative constant [10]. In what follows, we will introduce an approximation for $K(E|h)$ and for $K(h)$.

Notation

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 $Var(t) = \emptyset$. Given a term $f(t_1, \ldots, t_n)$ where $f \in \Sigma_{(s_1,\ldots,s_n),s}$, then $Type(t_i) = s_i$.

Estimate for K(h) and K(E|h)

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. Let us denote with E_ν 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. Let us denote with OVR_ν the set of variables of real type in OV_ν. $OVNR_\nu = OV_\nu - OVR_\nu$.

A *predictive* MDL criterion just describes the class of the examples. For instance, if a leaf is assigned a class c_j then the examples that fall into this leaf need not be described. Only exceptions need be described. However, the construction of our algorithm is guided by *descriptive* MDL, i.e., the examples must be described completely, including not only the class but also the arguments. For instance, if a leaf is assigned a class c_j then the examples that fall into this leaf need be described (except the class). Additionally, exceptions need be described completely, including the class.

Given a partition P, its information can be obtained as[2]:

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

The first term of the above formula 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. $InfoP(P, \nu)$ denotes the cost in bits of each partition. Details about the cost of the partitions can be found in [5]. Thus, $K(h)$ can be estimated as follows:

$$K(h) \approx InfoHead(h) + \textstyle\sum_{\nu \in nodes(h)} InfoPart(\pi(\nu), \nu) \qquad (1)$$

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).

The estimate for $K(E|h)$ is based on the construction of tables. 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:

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

– for finite discrete types, $Info(term_{i,j}) = \log n$, with n being the number of possible values of the type.
– 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.

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

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

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 $RealValue_k(e)$. We consider the cost of a real number (denoted as $infoR$) as a constant.

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

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

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

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. With this, we have an estimate for the second term of the definition of the MDL principle, $K(E|h)$. In this way,

$$K(E|h) \approx \sum_{\nu \in leaves(h)} Info(E_\nu|\nu) \qquad (2)$$

Information of the Tree

Now, we introduce the information for the whole tree. Given a tree T with root node ν, the cost of describing T is defined as:

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

where $InfoN(E,\nu)$ can be 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 $InfoN_0(E,\nu) = Info(E_\nu|\nu)$.

3 Constructing the Multi-Tree

In this section, we define the construction of decision multi-trees. At this time, we can establish the way in which one tree is build from the root $f(X_1, X_2, ..., X_{n-1}) = X_n$, which is an open node. First of all, the *node selection criterion* chooses the node that is explored first. From all the open nodes, we select the node with less $InfoN_0(E,\nu)$, i.e. with less $Info(E_\nu|\nu)$. This criterion has little relevance, since all the open nodes must be explored sooner or later. Secondly, the *split selection criterion* is much more important, since it selects between the many possible partitions. 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. The other partitions are preserved as suspended nodes.

When pruning is not activated (when data is assumed to be noise-free) the stopping criterion is easy to determine. A node is closed when the class is consistent with all the examples that fall into that node (i.e. E_ν). When pruning is activated, the stopping criterion is given by the pruning criterion that we describe next.

The MDL criterion has been used for pre-pruning and post-pruning decision trees. Usually, a predictive MDL criterion has been applied for this purpose [20]. Note that exceptions are much costlier in the case of predictive MDL criterion, because there is a great difference from regular examples (no extra bits are needed) and exceptions (the arguments and class need be coded). This means that the predictive MDL criterion would prune too late in many cases. For this reason, we will also use a descriptive criterion for pruning.

More concretely, the descriptive MDL criterion is used for the *pre-pruning criterion* in the following way. A node is pruned when the description at level $n+1$ is greater than at level n. More formally, let us suppose a node ν, then the tree should be pre-pruned when:

$$InfoN_0(E_\nu|\nu) < InfoPart(\pi(\nu)) + \sum_{i=1..range(\nu)} InfoN_0(E, child_i(\nu))$$

Now that we can have non-closed nodes which can be pruned and we need an extra bit for every node to tell whether a node is pruned or not. In the case a node is pruned, the node is assigned the most common class under that node. This has to be coded as well, with a cost $\log nC$ where nC is the number of classes. Note that, in the above formula $InfoN_0$ does not need to code the class for all the examples which are consistent with their most common class. For exceptions, however, this class has to be coded as usual, taking into account all the possible values appearing in the exceptions.

The generation of the tree stops when all nodes are closed or pruned.

As we have stated in the introduction, once a tree is concluded, new trees can be generated in order to construct a multi-tree. To do this, we have to establish a new criterion, a *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.

Finally, when different solutions are obtained we have to select one of them. We have considered two *solution selection criteria*: the MDL principle ($K(h) + K(E|h)$) or Occam's Razor ($K(h)$).

The use of a multi-tree ensures that, when the problem is small or there is time enough, many solutions will be generated. The first idea is to select the best solution according to the solution selection criterion. As we have commented on in the introduction, each solution of the tree can be expressed as a functional logic program, which provides a comprehensible model.

Another option is to combine hypothesis in order to improve the accuracy, however this represents the loose of a comprehensible model. The method used for combining solutions is propagating upwards a vector of the probabilities of nodes and they are combined whenever an OR-node is found.

4 Experiments

The method presented in the previous section has been implemented in a machine learning system (named FLIP2), which is able to induce problems from arbitrary evidence using a functional logic language as representation language. This system and examples is publicly available in [4]. We have performed different experiments using this system which show that our multi-tree approach pays off in practice.

All the examples induced were extracted from the UCI repository [13] and from the Clementine system [8] sample examples, and are well-known by the machine learning community. Some of them contain noisy data and real arguments. We have split the data sets in two similar-sized parts, using one part as the train set and the other one as the test set.

Table 1 contains the results of the experiments: the accuracy and the number of rules of the solution program depending on the number of hypotheses induced ($Numtree$). The experiments were executed on a Pentium III 800 Mhz with 175 MB of memory. The experiments demonstrate that the system is able to induce programs from a complex evidence (i.e. large number of examples and many parameters). The increasing of $Numtree$ allows to get shorter theories (i.e. more comprehensible), without an important worsening of accuracy.

Numtree	1		10		100		1000	
	Rules	Accuracy	Rules	Accuracy	Rules	Accuracy	Rules	Accuracy
cars	126	85.53	126	85.53	101	85.65	69	84.03
house-votes	71	86.70	71	86.70	53	93.11	49	89.90
tic-tac-toe	346	65.55	297	70.35	263	75.99	252	74.94
nursery	471	91.34	467	91.37	408	91.77	364	92.37
monks1	17	94.90	17	94.90	7	100	7	100
monks2	100	69.90	97	69.90	89	79.16	61	79.62
monks3	35	88.42	35	88.42	28	87.26	22	88.19
drugs	134	92.09	132	92.90	131	92.72	129	93.00
tae	41	57.33	40	60.00	38	60.00	37	61.33

Table 1. Accuracy and number of rules generated in the learning of some classification problems.

The accuracy obtained by using the combination of hypotheses is detailed in Table 2. In most cases the use of the combination produces an improvement of the result. However, the raise of the accuracy is not linear w.r.t. $Numtree$. It could be interesting to determine automatically the optimal $Numtree$ depending on the features of the problem.

Numtree	1	10	100	1000
Example	Accuracy	Accuracy	Accuracy	Accuracy
cars	85.53	85.53	90.16	92.12
house-votes	86.69	88.99	92.66	92.20
tic-tac-toe	65.55	82.46	83.71	85.39
nursery	91.34	91.45	92.98	93.98
monks1	94.90	94.90	100	91.90
monks2	69.90	69.91	79.17	79.63
monks3	88.43	90.05	92.59	86.80
drugs	92.09	92.45	93.09	93.09
tae	57.33	57.33	57.33	58.67

Table 2. Accuracy obtained in the learning of some classification problems using the combination of hypotheses.

An experimental comparison of accuracy with some well known ML systems is illustrated in Table 3. The results for FLIP2 are obtained with $Numtree = 1000$ and using solution combination. *C5.0* is a decision tree learner, *Rules* is a rule induction algorithm[3]. The two are part of the data-mining package Clementine 5.2 from SPSS[8]. *C4.5* [18] represents the results of FLIP2 using the splitting criterion of *C4.5* generating just one solution. As can be seen in the tables, the accuracy of FLIP2 is superior or similar to the other systems in general depending on the problem.

Another technique that permits to adapt the learning process to the amount of resources available is *Boosting* [19]. This mechanism has been incorporated successfully to *C5.0*. Figure 1 shows how *Boosting* increases constantly the resources required (time and memory) depending on the number of iterations of the algorithm.The reason is because each run does not use the information

[3] Note that *Rules* is not able to deal with examples with noisy classes like *tae*.

Example	FLIP2	C5.0	Rules	C4.5
cars	92.12	88.54	85.88	90.39
house-votes	92.20	94.50	94.5	94.04
tic-tac-toe	85.39	80.38	77.45	78.70
nursery	93.98	95.99	95.73	95.67
monks1	91.90	87.90	100	78.00
monks2	79.63	65.05	65.74	69.90
monks3	86.80	97.22	94.44	92.12
drugs	93.09	97.27	95.05	90.18
tae	58.67	54.67	-	38.67

Table 3. Accuracy comparison between some ML systems.

generated in previous iterations. On the contrary, FLIP2 needs more resources initially, every step it reuses the trees generated previously, thus the increasing of resources required is slower.

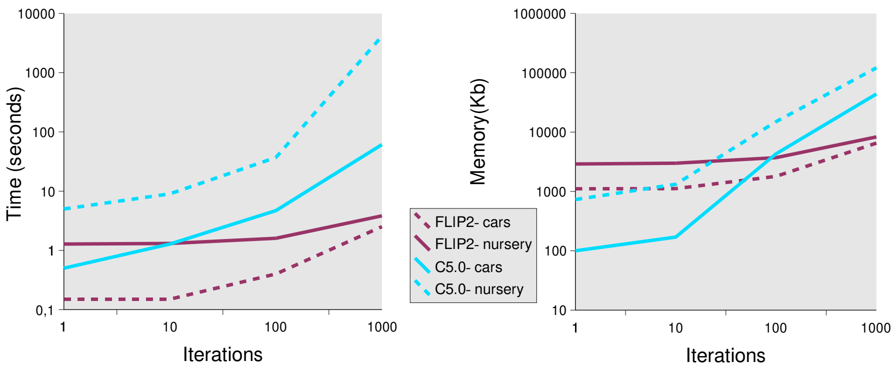

Fig. 1. Time and memory required by FLIP2 and *C5.0* with *Boosting* depending on the number of iterations.

5 Conclusions and future work

Much recent work in the area of machine learning has been devoted to the combination of results given by different learners and by several iterations of the same learner [1, 19]. In the latter case, the algorithm is re-run from scratch with different samples from the training data, giving a bank of hypotheses, from which the best solution can be selected or a voting can be made to give a combined solution.

In this paper, we have presented an algorithm that has been designed to give multiple solutions in an efficient way, re-using parts of other solutions and maintaining the same common structure for all the solutions. We have called this structure a multi-tree. The accuracy of the first solution given by our algorithm is comparable to other systems, such as C5.0. However, the multi-tree can be further populated in order to improve this accuracy. Moreover, a new tree is not generated each time from scratch, but just some new branches are explored. This makes that once a first solution has been found the time which is required to produce the next n trees increases in a sublinear way.

This behaviour is similar to an anytime algorithm, as we have stated, which increases the quality of the solution with increasing time. This quality is improved for the best solution (which can be expressed as a comprehensible functional logic program) or as a combination of the multiple solutions (which is less comprehensible but usually improves further the accuracy). The anytime character of our algorithm makes it very suitable for data-mining applications.

As future work we plan to include the complete set of partitions, in order to induce recursive problems and to address problems with deep structure. Finally, more experimental work with this kind of problems must also be performed.

References

1. E. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting and variants. *Machine Learning*, 36:105–139, 1999.
2. Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth Publishing Company, 1984.
3. T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proc. of the 7th National Conference on Artificial Intelligence*, pages 49–54, 1988.
4. C. Ferri, J. Hernández, and M.J. Ramírez. The FLIP system homepage. `http://www.dsic.upv.es/~flip/`, 2000.
5. C. Ferri, J. Hernández, and M.J. Ramírez. Learning MDL-guided Decision Trees for Constructor-Based Languages. In *WIP track of 11th Int. Conf. on Inductive Logic Progr,ILP01*, pages 39–50, 2001.
6. Y. Freund and R.E. Schapire. Experiments with a new boosting algorithm. In *Proc. of the 13th Int. Conf. on Machine Learning (ICML'1996)*, pages 148–156. Morgan Kaufmann, 1996.
7. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19-20:583–628, 1994.
8. SPSS Inc. Clementine homepage. `http://www.spss.com/clementine/`.
9. L.A. Levin. Universal Search Problems. *Problems Inform. Transmission*, 9:265–266, 1973.
10. M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. 2nd Ed. Springer-Verlag, 1997.
11. M. Mehta, J. Rissanen, and R. Agrawal. MDL-Based Decision Tree Pruning. In *Proc. of the 1st Int. Conf. on Knowledge Discovery and Data Mining (KDD'95)*, pages 216–221, 1995.
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. 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.
15. B. Pfahringer. Compression-based discretization of continuous attributes. In *Proc. 12th International Conference on Machine Learning*, pages 456–463. Morgan Kaufmann, 1995.
16. J. R. Quinlan. Induction of Decision Trees. In *Read. in Machine Learning*. M. Kaufmann, 1990.
17. J. R. Quinlan. Learning Logical Definitions from Relations. *M.L.J*, 5(3):239–266, 1990.
18. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
19. J. R. Quinlan. Bagging, Boosting, and C4.5. In *Proc. of the 13th Nat. Conf. on A.I. and the Eighth Innovative Applications of A.I. Conf.*, pages 725–730. AAAI Press / MIT Press, 1996.
20. J. R. Quinlan and R. L. Rivest. Inferring Decision Trees Using The Minimum Description Length Principle. *Information and Computation*, 80:227–248, 1989.
21. J. Rissanen. Modelling by shortest data description. *Automatica*, 14:465–471, 1978.