

# Top-down induction of logical decision trees

*Hendrik Blockeel and Luc De Raedt*

*Report CW 247, January 1997*



Katholieke Universiteit Leuven  
Department of Computer Science

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# Top-down induction of logical decision trees

*Hendrik Blockeel and Luc De Raedt*

*Report CW 247, January 1997*

Department of Computer Science, K.U.Leuven

## **Abstract**

Top-down induction of decision trees (TDIDT) is a very popular machine learning technique. Up till now, it has mainly been used for propositional learning, but seldomly for relational learning or inductive logic programming. The main contribution of this paper is the introduction of logic decision trees, which make it possible to use TDIDT in inductive logic programming. An implementation of this top-down induction of logic decision trees, the TILDE system, is presented and experimentally evaluated.

# Top-down Induction of Logical Decision Trees

Hendrik Blockeel and Luc De Raedt  
Katholieke Universiteit Leuven  
Department of Computer Science  
Celestijnenlaan 200A  
3001 Heverlee

e-mail: {Hendrik.Blockeel, Luc.DeRaedt}@cs.kuleuven.ac.be

January 21, 1997

## Abstract

Top-down induction of decision trees (TDIDT) is a very popular machine learning technique. Up till now, it has mainly been used for propositional learning, but seldomly for relational learning or inductive logic programming. The main contribution of this paper is the introduction of logical decision trees, which make it possible to use TDIDT in inductive logic programming. An implementation of this top-down induction of logical decision trees, the TILDE system, is presented and experimentally evaluated.

## 1 Introduction

Top-down induction of decision trees [Quinlan, 1986; Quinlan, 1993a] is the best known and most succesful machine learning technique. It has been used to solve numerous practical problems. It employs a divide-and-conquer strategy, and in this it differs from its rule-based competitors (such as CN2 [Clark and Niblett, 1989], AQ [Michalski *et al.*, 1986]), which are based on covering strategies (cf. [Boström, 1995]). Within attribute value learning (or propositional concept-learning) TDIDT is more popular than the covering approach. Yet, within first order approaches to concept-learning, only a few learning systems have made use of decision tree techniques ([Watanabe and Rendell, 1991; Bergadano and Giordana, 1988]), and in the field of inductive logic programming, the approach has almost totally been ignored. With the exception of [Boström, 1995] and some systems that transform ILP problems into propositional form (e.g. LINUS [Lavrač and Džeroski, 1994], Indigo [Geibel and Wysotzki, 1996]) almost every ILP system uses a covering approach.

The main reason why divide-and-conquer approaches are not popular within inductive logic programming, lies in the discrepancies between the clausal representation employed within inductive logic programming and the structure underlying a decision tree. Our main contribution is the introduction of a logical decision tree representation that corresponds to a clausal representation. Logical decision trees upgrade the attribute value representations used within classical TDIDT algorithms, and also generalize the earlier work by [Watanabe and Rendell, 1991].

Given the logical decision tree representation, it is easy to design and implement an algorithm for top-down induction of logical decision trees by adapting C4.5's heuristics. This results in the TILDE system, which is the main topic of this paper. TILDE works within the *learning from interpretations* paradigm introduced by [De Raedt and Džeroski, 1994], and used in the ICL system of [De Raedt and Van Laer, 1995].

Within TILDE, we also incorporated two other novelties w.r.t. inductive logic programming: discretization of numeric attributes (based on [Van Laer *et al.*, 1996; Fayyad and Irani, 1993]) and dynamic lookahead facilities. We also report on a number of encouraging experiments, in the domains of mutagenesis, finite element mesh design, and musk molecules.

This text is organized as follows. In Section 2, we briefly discuss the ILP setting that will be used. In Section 3, we introduce the notion of a logical decision tree. In Section 4, an algorithm for the induction of logical decision trees is presented, and in Section 5, a prototype implementation of this algorithm is discussed. Section 6 contains an empirical evaluation of this implementation, and finally, in Section 7 we conclude and touch upon related work.

## 2 The Learning Problem

We assume familiarity with the Prolog programming language (see e.g. [Bratko, 1990]).

We essentially use the *learning from interpretations* paradigm for inductive logic programming, introduced by [De Raedt and Džeroski, 1994], used in ICL [De Raedt and Van Laer, 1995], and related to other inductive logic programming settings in [De Raedt, 1996].

In this paradigm, each example is a Prolog knowledge base (i.e. a set of definite clauses), encoding the specific properties of the example. Furthermore, each example is classified into one of a finite set of possible classes. One may also specify background knowledge  $B$  in the form of a Prolog knowledge base.

More formally, the problem specification is :

### Given

- a set of classes  $C$
- a set of classified examples  $E$ ,
- a background theory  $B$

**Find:** a hypothesis  $H$  (a set of definite clauses in Prolog), such that :

- for all  $e \in E$   $H \wedge e \wedge B \models c$ , and  $H \wedge e \wedge B \not\models c'$  where  $c$  is the class of the example  $e$  and  $c' \in C - \{c\}$ .

### Example 1 (Running example)

*Imagine the following situation: an engineer has to check a set of machines. A machine consists of several parts that may be worn and in need of replacement. Some of these can be replaced by the engineer, others only by the manufacturer of the machine. If a machine contains worn parts that cannot be replaced by the engineer, it has to be sent back to the manufacturer.*

*Given the following set of examples (where each example corresponds to one machine):*

Example 1	Example 2	Example 3	Example 4
<i>class(keep)</i> <i>worn(gear)</i> <i>worn(chain)</i>	<i>class(sendback)</i> <i>worn(engine)</i> <i>worn(chain)</i>	<i>class(sendback)</i> <i>worn(control_unit)</i>	<i>class(keep)</i> <i>worn(chain)</i>

*and the following background knowledge:*

Background knowledge
replaceable(gear)
replaceable(chain)
not_replaceable(engine)
not_replaceable(control_unit)

a valid hypothesis, written in clausal logic, is:

$$\text{class}(\text{send\_back}) \leftarrow \text{worn}(X), \text{not\_replaceable}(X)$$

Due to the restrictions on the hypothesis  $H$  imposed by using logical decision trees, this notion of coverage is essentially the same as that known under the label *learning from interpretations*<sup>1</sup>.

### 3 Logical Decision Trees

**Definition 3.1** A binary decision tree (BDT) can be defined as follows:

- $\boxed{k}$ , with  $k$  some class, is a BDT; it is also called a *leaf*. We will use the notation  $T.\text{class}$  to denote the class indicated by a leaf  $T$ .
- if  $L$  and  $R$  are BDTs, and  $C$  is a test that can have two outcomes, then  $T = \begin{array}{c} C \\ \swarrow \quad \searrow \\ L \quad R \end{array}$  is a BDT. It is also called an *internal node*.  $L$  and  $R$  are called the left and right subtrees of  $T$ . We will also use the notation  $T.\text{test}$ ,  $T.\text{left}$  and  $T.\text{right}$  to refer to the components  $C$ ,  $L$  and  $R$  respectively.

**Definition 3.2** A logical decision tree (LDT) is a binary decision tree that fulfils the following constraints:

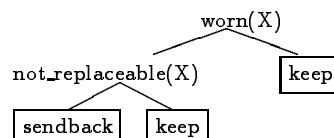
- every test is a (first-order logic) conjunction of literals
- a variable that is introduced in some node (this means it does not occur in higher nodes) can not occur in its right subtree

Why the second requirement is necessary, follows from the logical meaning of an LDT, which will be specified further. In short, it corresponds to the fact that newly introduced variables are quantified within the conjunction, and the right subtree is relevant when this conjunction fails; now if the conjunction fails (“there is no such  $X$ ”) it does not make sense to speak of this  $X$  further down the tree.

Classifying an example using a logical decision tree can be done as shown in Algorithm 1. As an example  $e$  is a Prolog database, a test corresponds to checking whether a query  $Q$  succeeds in  $e \wedge B$ .

**Example 2** Recall the hypothesis that was stated in Example 1: a machine has to be sent back to the manufacturer if it contains worn parts that can't be replaced by the engineer; otherwise it can be kept in place.

The above rule can be represented by a logical decision tree in the following way:



<sup>1</sup> The interpretation corresponding to the each example  $e$  is then the minimal Herbrand model of  $B \wedge e$ .

```

procedure classify( $E$ ):
   $C := true$ 
   $N := root$ 
  while  $N$  is not a leaf do
    if  $E \models C \wedge N.test$ 
      then
         $C := C \wedge N.test$ 
         $N := N.left$ 
      else
         $N := N.right$ 
  return  $N.class$ 

```

Figure 1: Classification of an example using an LDT

In order to formalize the notion of a logical decision tree, we will now define a mapping between logical decision trees and logic programs. After this mapping has been established, we will also show how a Prolog program that is equivalent to the LDT can be found.

A logic program can be derived from a logical decision tree in the following way. First, with each node in the tree, we will associate two things:

1. a definite clause, defining a nullary predicate; in each node a different predicate is defined
2. a Horn query; this query can make use of the predicates defined in nodes higher in the tree

The idea behind this association of queries with nodes, is the following: when the tree is used to classify an example, the example will be sorted down the tree. When during this sorting process some specific node is encountered, this implies that the associated query succeeds for the example. The converse also holds: if the query succeeds for the example, then this node will be encountered at some time during the classification process. The definite Horn clauses define invented predicates that are needed to express the queries.

All this implies that running the query associated with a leaf is equivalent to sorting the example down the tree and checking whether it is sorted into this leaf. Therefore, the tree can be transformed into a logic program that consists of clauses, each containing a class-indicating literal in the head, and in the body a query associated with some leaf of the tree.

Algorithm 2 illustrates the process of associating clauses and queries to nodes. It proceeds in the following manner:

- With the top node  $T$ , the clause  $c_0 \leftarrow T.test$  and the empty query  $\leftarrow$  are associated.
- For every internal node  $N$  with associated clause  $c_i \leftarrow P$  and associated query  $\leftarrow Q$ , the query  $\leftarrow P$  is associated with its left subtree. The query  $\leftarrow Q, not(c_i)$  is associated with the right subtree.
- With each internal node  $N$  with associated query  $\leftarrow Q$ , the clause  $c_i \leftarrow Q, N.test$  is associated;  $i$  is a number unique for this node.

Finally, for each leaf with associated query  $\leftarrow Q$ , the clause  $class(k) \leftarrow Q$  is added to the logic program (with  $k$  the class indicated by the leaf).

```

procedure associate( $N$ ):
  if  $N$  is not a leaf then
    assign a unique predicate  $c_i$  to this node
     $(\leftarrow Q) := \text{query}(N)$ 
    assert  $c_i \leftarrow Q, N.test$ 
     $\text{query}(N.left) := \leftarrow Q, N.test$ 
     $\text{query}(N.right) := \leftarrow Q, \text{not}(c_i)$ 
    associate( $N.left$ )
    associate( $N.right$ )
  else
     $k := N.class$ 
     $Q := \text{query}(N)$ 
    assert  $class(k) \leftarrow Q$ 
procedure deriveLogicProgram( $T$ ):
   $\text{query}(T) := \leftarrow$ 
  associate( $T$ )

```

Figure 2: Mapping LDT's onto logic programs

It may seem strange that with the right subtree of  $N$ , the algorithm associates the query  $\leftarrow Q, \text{not}(c_i)$  instead of  $\leftarrow Q, \text{not}(N.test)$ , as might be expected. The reason for this is that the queries of the left and right subtree should be complementary: for each example sorted down to this node (i.e.  $\leftarrow Q$  succeeds for the example), exactly one of both queries should succeed. Now, it is clear that if  $\leftarrow Q$  succeeds, exactly one of  $\leftarrow Q, c_i$  (which is equivalent to  $\leftarrow c_i$ , as  $c_i$  is a strictly stronger condition than  $Q$ ) and  $\leftarrow Q, \text{not}(c_i)$  must succeed. This is not the case, however, with  $\leftarrow Q, N.test$  and  $\leftarrow Q, \text{not}(N.test)$ , when  $N.test$  shares variables with  $Q$ . The former clause succeeds if there exists a substitution such that  $Q$  and  $N.test$  are true, and the complementary clause should state that there is no such substitution. The latter clause however, succeeds if there exists a substitution such that  $Q$  is true and  $N.test$  is false. This is not the same, when  $Q$  and  $N.test$  have variables in common. An illustration of this will be given in the next example.

**Example 3** *Figure 3 shows the queries and clauses associated with each node of the tree built in the previous example.*

*This example also illustrates the fact that the query associated to the right subtree of a node ought to contain the negation of the predicate associated with that node, and not just the negation of the added literal. There is a difference between*

$$\leftarrow \text{worn}(X), \text{not}(\text{not\_replaceable}(X))$$

and

$$\begin{aligned} c_1 &\leftarrow \text{worn}(X), \text{not\_replaceable}(X) \\ &\leftarrow \text{worn}(X), \text{not}(c_1) \end{aligned}$$

*because of the fact that the first query succeeds if there is a worn part in the machine that is replaceable (not not\_replaceable), while the second query succeeds if there are worn parts, but all of the worn parts are replaceable (there are no worn parts that are not\_replaceable).*

This difference also introduces an asymmetry in the test which does not occur in the propositional case: adding a literal, and adding the negation of the literal,

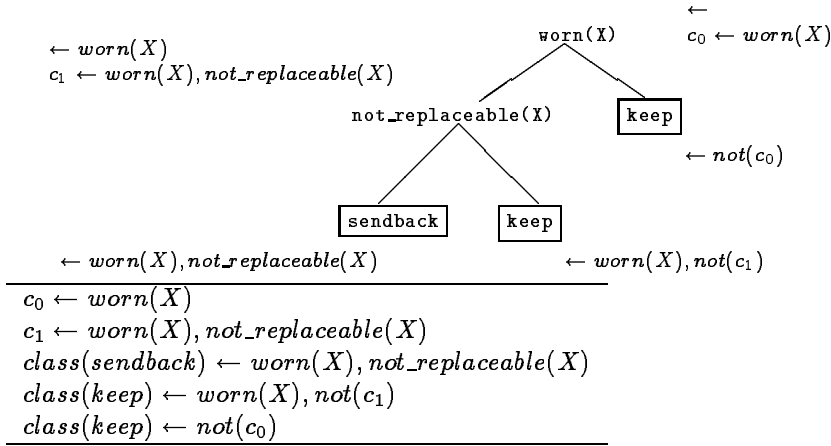


Figure 3: Clauses and queries associated with nodes of a tree, and the logic program derived from the tree

is *not* the same. This means that it may be interesting to allow negated literals in queries. In our example, it is indeed not possible to represent the same theory in an LDT (or in a logic program for that matter) if only the predicate *replaceable*, and not its negation, were available.

The logic program shown in Figure 3 can be executed in Prolog. However, it is possible to derive a simpler and more efficient Prolog program using the cut operator. By adding a cut at the end of each clause, the negated  $c_i$ -literals in the program become superfluous. The use of cuts reflects the fact that the choice of subtree with which to continue is deterministic: if an example has been sorted to the left subtree of some node, there is no need to check afterwards whether it can also be sorted to the right subtree: the complementary nature of the subtrees guarantees that it cannot.

**Example 4** *The following Prolog program is equivalent to the logic program in Figure 3.*

```
class(sendback) :- worn(X), not_replaceable(X), !.
class(keep) :- worn(X), !.
class(keep).
```

*This can, of course, be simplified a bit: as the second and third clause predict the same class, the second clause is superfluous.*

A drawback of the use of cuts, is that the order in which the clauses appear is important. The clauses must occur in the same order of the leaves with which they are associated. This also implies that, if clauses for only one class are needed, it is not possible to remove the clauses that predict another class.

## 4 An Algorithm for the Induction of Logical Decision Trees

In this section, we present an algorithm for top-down induction of logical decision trees. Recall that the input to the learning algorithm consists of a set of labelled examples  $\mathcal{E}$ , background knowledge  $\mathcal{B}$ , and a language  $\mathcal{L}$  stating which kind of tests are allowed in the decision tree.

```

procedure buildtree( $T, \mathcal{E}, Q$ ):
  if  $\mathcal{E}$  is homogeneous enough
  then
     $K := \text{most\_frequent\_class}(\mathcal{E})$ 
     $T := \text{leaf}(K)$ 
  else
     $Q_b := \text{best element of } \rho(Q), \text{ according to some heuristic}$ 
     $T.\text{test} := C' \text{ where } Q_b = \leftarrow Q, C'$ 
     $\mathcal{E}_1 := \{E \in \mathcal{E} \mid E \cup B \models Q_b\}$ 
     $\mathcal{E}_2 := \{E \in \mathcal{E} \mid E \cup B \not\models Q_b\}$ 
    buildtree( $T.\text{left}, \mathcal{E}_1, Q'$ )
    buildtree( $T.\text{right}, \mathcal{E}_2, Q$ )

```

Figure 4: Algorithm for logical decision tree induction

Our algorithm employs the basic TDIDT algorithm. Most of its heuristics have been implemented exactly as in C4.5 (the gainratio heuristic, the post-pruning algorithm, ...). The only point where our algorithm differs is in the computation of the tests to be placed in a node. To this aim, it employs a classical refinement operator under  $\theta$ -subsumption. A clause  $c_1$   $\theta$ -subsumes another clause  $c_2$  if and only if there is a substitution  $\theta$  such that  $c_1\theta \subseteq c_2$ . A refinement operator under  $\theta$ -subsumption is an operator  $\rho$  mapping clauses onto sets of clauses, such that for any clause  $c$  and  $\forall c' \in \rho(c)$ ,  $c$   $\theta$ -subsumes  $c'$ . The operator could, for instance, add literals to the clause, or unify several variables in it.

For our decision tree induction algorithm, we will use a refinement operator that adds one or more literals to a clause. When a node  $N$  is to be refined, the set of all the conjunctions that can possibly be put in that node is computed as

$$\{C \mid (\leftarrow Q, C) \in \rho(\leftarrow Q)\}$$

where  $Q$  is the query associated with the node, as defined by Algorithm 2.

Declarations defining the refinement operator  $\rho$  are an input parameter of the algorithm. Since  $\rho$  determines the language bias, a separate language bias specification is not needed. When we refer to the language bias specification, we mean the definition of  $\rho$ .

The algorithm shown in Figure 4 builds a logical decision tree  $T$  for a set of examples  $E$ . The query associated with a node is passed as the third argument of the procedure. The background knowledge  $B$  is assumed to be available implicitly.

## 5 A Practical Implementation: TILDE

Algorithm 4 has been implemented in a system called TILDE (Top-down Induction of Logical DEcision trees). In this section, we discuss the heuristics and the language definition formalism that have been chosen, and also a number of other practical aspects.

### 5.1 The Heuristic Function

The implementation of TILDE is strongly based on C4.5 [Quinlan, 1993a], one of the most successful systems for learning decision trees in the attribute value context. This is, among other things, reflected in the heuristic value assigned to tests. As in

C4.5, the gain ratio of a test is used to evaluate it. Suppose that on the basis of a clause  $C$ , a set of examples  $E$  is partitioned into those where the clause succeeds ( $E_1$ ) and those where it doesn't ( $E_2$ ). The *gain* of the clause is the difference between the entropy of  $E$  and the weighted average of the entropies of  $E_1$  and  $E_2$ . The *splitting information* of the clause is the maximal gain that any clause with the same success ratio (number of examples for which it succeeds, divided by the total number of examples) could achieve. This will be larger for clauses that split the examples more evenly.

The gain ratio is then defined as gain divided by splitting information. Quinlan [Quinlan, 1993a] mentions that this ratio is a better criterion for the usefulness of a clause than gain itself, because some tests inherently stand a good chance of leading to high gain, without really being relevant. Quinlan notes that this is especially the case when the test can have more than two outcomes, but even when only two outcomes are possible (which is the case in our implementation, a clause can only succeed or fail), gainratio usually chooses better tests than gain ([Quinlan, 1993a], p. 24).

As in C4.5, tests with a smaller gain than the average gain of all the tests considered, have been filtered out; this is done because the gain ratio criterion becomes unstable for clauses with very low splitting information (a small denominator in the computation of the heuristic can lead to accidentally very large heuristic values).

Although our current implementation allows the user to choose between gain and gainratio as a heuristic, some experimentation with TILDE has shown the same trends that were mentioned by Quinlan: gainratio usually performs better than gain. For this reason, the experiments shown in Section 6 have all been performed with the gainratio criterion used as heuristic.

## 5.2 Post-Pruning of Trees

A second feature of our implementation that has been copied from C4.5, is its ability to post-prune trees. It is possible that a complex tree, although scoring a higher accuracy on the training data, performs less accurately on a test set than a more simple tree, due to overfitting. Joining two leaves of a tree into one can in this respect be profitable. Because this cannot be reflected in the accuracy on the training data (which most probably decreases), Quinlan proposes to estimate the error rate that a tree can be expected to score on a test set. This estimation is based on a statistical reasoning that “should be taken with a large grain of salt” (Quinlan, p. 41)<sup>2</sup>, but seems to work in practice.

TILDE uses the same error estimate to prune the induced trees afterwards. It always returns both the unpruned and pruned versions of the trees, so that a comparison can be made.

## 5.3 The Stopping Criterion

At this moment, TILDE uses very conservative stopping criteria. Splitting of example sets will stop if the set has become too small to split it up any more (the minimal number of cases that a leaf should contain can be specified by the user). This is in correspondence with C4.5's stopping criterion. More precisely, a significance test (checking whether the test that is added is significantly better than a random test) is (as yet) not included. While this gives rise to overfitting, the post-pruning mechanism compensates for this.

---

<sup>2</sup>There is an implicit assumption that incorrectly classified examples are some sort of “rare events” that occur randomly in a set of examples, with small frequency; it is this frequency that is estimated.

## 5.4 The Language Specification

As we have already mentioned, the specification of the refinement operator and of the language bias coincide. In TILDE, this specification basically consists of the conjunctions that can be put in the nodes of the tree. Modes and types of the variables in the conjunction are added to this, as well as the maximal number of occurrences of the conjunction that is allowed in one clause. Three different modes are allowed: *in* (+) means that the variable must be unified with an existing variable, *in or out* (−) means that it can but need not be unified with an existing variable, and *out* (no preceding symbol) means that it is a new variable. Types can (but need not) be specified separately, and guarantee that the derived logic program will be type-conform.

**Example 5** *The specification*

$$\text{rmode}(8 : (\text{p}(+X, -Y, Z), \text{q}(Z)))$$

tells TILDE that the conjunction  $p(X, Y, Z), q(Z)$  can occur maximally 8 times in the associated query of a node, and specifies possible unifications of the variables of  $p$  with other variables in the query. A node with associated query

$$\leftarrow a(A), b(B, C).$$

can then be refined in the following ways: a literal  $p$  can be added with first argument  $A$ ,  $B$  or  $C$ , second argument  $A, B, C$  or a new variable  $Y$ , the third argument  $Z$ , and the literal  $q$  is simultaneously added with argument  $Z$ . This yields 12 possible refinements.

If there are type specifications, e.g.

```
type(p(name, int, real)).
type(a(name)).
type(b(int, real)).
type(q(real)).
```

then only the following two queries are refinements:

$$\begin{aligned} &\leftarrow a(A), b(B, C), p(A, B, Z), q(Z) \\ &\leftarrow a(A), b(B, C), p(A, Y, Z), q(Z) \end{aligned}$$

### Dynamic Refinement

Dynamic refinement is a generalization of the generation of constants that many ILP systems provide. Muggleton's Prolog system [Muggleton, 1995], for instance, allows the user to specify that a constant, and not a variable, should be put in a certain position in a literal. The system will itself determine meaningful constants and generate clauses with these constants filled in.

TILDE's dynamic refinement is inspired by the so-called "call handling" procedure implemented in the CLAUDIEN system [De Raedt and Dehaspe, 1996; Muggleton and Page, 1994]. When some literal is to be added to a clause, but it will only be clear at the time of refinement exactly what the literal will look like, then it is possible for the user to leave some parts blank and specify that some predicate has to be called at the time of refinement to determine the missing information.

TILDE's way of specifying how literals can be generated dynamically is as follows:

$$\#(E * M * X : \langle \text{generating\_call} \rangle, \langle \text{conjunction} \rangle)$$

(the '#' notation stems from the Prolog system, where it is used to specify that a constant is required at that position).  $X$  is a variable or list of variables that occurs

in both *generating\_call* and *conjunction*. Given a context consisting of the current set of examples and the query that is being refined, *generating\_call* instantiates  $X$  in such a way that it makes sense to add *conjunction* to the clause. After that, the literal is added. The numbers  $E$  and  $M$  specify the maximal number of examples in which *generating\_call* is called and the maximal number of instances of  $X$  it is allowed to return per example.

For instance, when a literal should be introduced comparing someone's age with some age limit, instead of specifying manually which literals can be added (e.g.  $+A < 7$ ,  $+A < 16$ ,  $+A < 18$ , ... — the  $+$  denoting that  $A$  must be bound) one could use the following construct:

```
 #(10*1*L:find_suitable_age_limit(L), +A < L)
```

The *find\_suitable\_age\_limit* ought to instantiate  $L$  with a suitable number, e.g. 16, and the literals that will be added to the clause are  $X < 16$  with  $X$  some variable that already occurs in the clause. In this example, only a constant is generated; the predicate itself ( $<$ ) is known in advance. In principle, the whole literal could be generated.

It is possible that the constant generator succeeds multiple times, generating multiple values for  $L$ ; in that case, each value gives rise to a new set of literals. This way, it is possible to simulate the behaviour of many ILP systems. FOIL [Quinlan, 1993b], for instance, will determine constants based on the values that effectively occur in the data for that literal. For the comparison of ages, the constant occurring in the test is always an age that occurs in the data. This can easily be simulated in the following way:

```
 #(1000*1*L:age(L), +A < L)
```

As a person has only one age, the *age* predicate is determinate; therefore it is normal to let it return one instantiation per example. If a generating call makes use of background knowledge only (i.e. is independent of the example), the first number would be 1. Other numbers are allowed but it is clear that in these cases this will not change the outcome.

Dynamic refinement is also useful for handling numerical data. Since the rules for generating the literal that is to be added can be specified completely by the user, any computation can be done. For instance, the following specification

```
 #(1*1*(A,B): (regression(p(X), q(Y), A, B), +U > A*(+V)+B)
```

would cause TILDE to perform regression from  $Y$  to  $X$ , where the variables  $X$  and  $Y$  are generated by  $p$  and  $q$  (assuming the *regression* predicate is implemented that way). The regression line would then be used to check whether  $U$  is above or below the value predicted using  $V$ , supposing  $U$  and  $V$  are bound variables. (Of course, this is only meaningful if  $U$  and  $V$  are of the right type.)

In this example, the regression is performed each time the corresponding test is considered. It is also possible to perform regression beforehand, store the results, and simply retrieve them during the tree induction. Since full Prolog is available to the user for implementing the literal generation rules, almost everything is possible.

It may seem that this mechanism is a bit too powerful for number handling, in the sense that many things are possible that are not needed, and the user needs to do some programming in order to use it. In practice, however, a few Prolog rules often suffice to specify what is wanted; more complex programs are only needed for rather exotic tests. Our first impressions are that dynamic refinement provides a convenient and expressive way to adapt TILDE's refinement operator to specific numerical domains.

## 5.5 Lookahead

An important challenge for ILP systems using greedy algorithms, is the fact that, in contrast with the attribute value setting, it is possible that a conjunction may need to be added even if it does not provide any gain. The conjunction may introduce extra variables into the clause that are relevant for classification. This implies that the gain ratio heuristic may not work very well for conjunctions introducing new variables. The problem has been acknowledged a long time ago, and greedy ILP systems sometimes have some method for alleviating it. The FOIL system [Quinlan, 1993b], for instance, handles determinate literals (which cannot result in gain but introduce new, possibly interesting variables) by adding them automatically to the clause.

The TILDE system does not detect itself whether lookahead is interesting for certain literals, but offers the user a way to specify this. Facts of the form

```
lookahead(<conj12
```

can be added to the language specification in order to tell the system that whenever a conjunction matching the template *conj<sub>1</sub>* is added, it should immediately try the addition of (all instants of) *conj<sub>2</sub>* to the test as well. For instance, the facts

```
rmode(1: neighbour(+X1, -X2)).  
lookahead(neighbour(X, Y), circuit(Y)).
```

specify that the *neighbour* literal can be added with a bound variable as first argument, and a bound or free variable as second argument; and that after adding *neighbour* TILDE should also try to test whether the new variable has the *circuit* property. If this test turns out to be the best one in the given situation, the conjunction of the two literals, as a whole, is the test that will be added to the tree. It is still possible that only the first literal is added, if the second literal offers no advantage over it.

The lookahead could be unbounded if the same literal occurs to the left and to the right in the lookahead specification. TILDE does not prohibit this, but uses an extra parameter *max\_lookahead* that limits the depth to which lookahead can be performed. For instance, in the context of the Mutagenesis dataset (discussed in Section 6):

```
max_lookahead(5).  
...  
lookahead(bond(A1, A2, B), #(1*10*E:element(E), atom(A2, E, _, _))).  
lookahead(bond(A1, A2, B12), bond(A2, A3, B23)).  
...  
rmode(10:bond(+A1, -A2, B)).
```

would cause TILDE to add tests such as (with *A* a bound variable and the other variables free):

```
bond(A, B, C).  
bond(A, B, C), atom(B, c1, _, _).  
bond(A, B, C), bond(B, D, E), atom(D, n, _, _).  
...
```

In other words, TILDE does not only inspect existing atoms but also atoms “near” existing atoms in the molecule (near meaning that they are connected by only a few bonds). Of course, in cases such as this one, too much lookahead leads to excessive computational costs; therefore lookahead should be used with caution, and avoided whenever possible.

Whether lookahead is preferable for some literal, seems to depend largely on the semantics of the literal. Although some guidelines allowing the system to decide for itself would come in handy, it is hard to specify general guidelines that always work. Therefore, TILDE leaves it up to the user to decide where lookahead is needed. The user often has a clear idea of this; e.g. in the *neighbour*-example given above, which was taken from the Mesh dataset, it is clear that most (if not all) edges have neighbours, so *neighbour* itself will never lead to any gain; but checking a neighbouring edge for a certain property seems much more interesting. It should also be noted that the *neighbour* predicate is not determinate; given its first argument, it will succeed for several second arguments.

## 5.6 The Refinement Operator

We now give a precise definition of the refinement operator  $\rho$  that TILDE uses, in terms of the language specification. This means we have to take into account the `rmode`, `type` and `lookahead` specifications. We first define an auxiliary operator  $\rho'$  as follows:

$$\rho'(\leftarrow Q) = \{\leftarrow Q, C\theta \mid \begin{array}{l} \text{rmode}(n : C) \\ \wedge C \text{ has been used less than } n \text{ times to form } Q \\ \wedge (Q, C\theta) \text{ is mode-conform and type-conform} \end{array}\}$$

$\rho'$  takes into account all specifications except lookahead. Defining an operator  $\lambda$  as follows:

$$\lambda(S) = \{\leftarrow Q, C', C' | (\leftarrow Q, C') \in S \wedge \exists \theta : \text{lookahead}(C'\theta, C)\}$$

we have

$$\rho(C) = \bigcup_{i=1}^N \lambda^i(\rho'(C))$$

where  $N$  is the maximum number of lookahead steps allowed (may be infinity).

## 5.7 Discretization

The motivation for discretizing numeric data is twofold and based on the findings in attribute value learning. On the one hand, there is an efficiency concern. On the other hand, by discretizing the data, one may sometimes obtain higher accuracy rates.

Current procedures in ILP to handle numbers, such as those by FOIL, and those employed by the older versions of CLAUDIEN, are quite expensive. The reason is that for each candidate clause, all values for a given numeric variable have to be generated and considered in tests. In large databases, the number of such values can be huge, resulting in a high branching factor in the search. Furthermore, in these approaches, discretization is done at runtime, i.e. it is repeated for every candidate clause. If one clause is a refinement of another one, a lot of redundant work may be done.

What TILDE does is to generate beforehand (i.e. before the actual learning starts) some interesting thresholds to test upon. This makes that thresholds are computed only once (instead of once for each candidate clause considered), and secondly, that the number of interesting thresholds (to be considered when refining clauses) is kept to a minimum, yielding a smaller branching factor. This has also yielded positive results in attribute value learning, cf. [Catlett, 1991].

In our approach to discretization, the user has to identify the relevant queries and the variables for which the values are to be discretized using a template of the form :

*to\_be\_discretized(Query, Varlist).*

The resulting numeric attributes (indicated by *Varlist*) are then discretized using a simple modification of Fayyad and Irani’s method.

The details of the Fayyad and Irani’s method can be found in [Fayyad and Irani, 1993] and [Dougherty *et al.*, 1995]. The only differences we apply are :

- due to the fact that one example may have multiple values for a numeric attribute, we use sum of weights instead of numbers of examples in the appropriate places of Fayyad and Irani’s formulae (i.e. when we count the real values that are less than a threshold, we sum their weights — in the attribute value case all values have weight 1 as each example has only one value for one attribute). The sum of the weights of all values for one numeric attribute or query in one example always equals one.
- the stopping criterion of Fayyad and Irani’s discretization method is very strict, in the sense that the method generates very few subintervals. When applying this criterion in TILDE, almost no subintervals would be generated. Therefore, TILDE’s discretization method takes as a parameter the desired number of thresholds to be generated.

## 6 Experimental Evaluation

We have performed experiments with the TILDE system on a broad range of problems.

### 6.1 Mutagenesis Dataset

The Mutagenesis data are a frequently used ILP benchmark. The aim is to discriminate mutagenic molecules from non-mutagenic ones, by looking at their molecular structure (i.e. the atoms and bonds of which it consists).

[Srinivasan *et al.*, 1995] introduces four levels of background knowledge that can be used with these data. Most experiments that have been done on the Mutagenesis dataset use one of these. We briefly describe the four backgrounds. Note that  $B_1 \subset B_2 \subset B_3 \subset B_4$ .

- $B_1$ : the atoms that are present in the molecule are given, as well as the bonds between them; the type of each bond is given (single, double, ...) as well as the element and type of each atom (for one element, several types may be distinguished)
- $B_2$ : as  $B_1$ , but tests on continuous attributes (atom charge) are allowed
- $B_3$ : as  $B_2$ , but 2 attributes describing the molecule as a whole have been added, of which experts know they are relevant
- $B_4$ : as  $B_3$ , with explicit knowledge about complex structures (benzene rings etc.) added

#### 6.1.1 Comparison With Other Systems

[Srinivasan *et al.*, 1995] compares the accuracies and complexities of the induced theories for the ILP systems FOIL and Progol, as well as the time consumed by the induction process.

We have performed experiments on all four levels of background knowledge. Table 1 compares TILDE’s results with those of Progol and other systems that have

Accuracies				
	Progol	FOIL	Indigo	TILDE
$B_1$	76 %	61 %		75 %
$B_2$	81 %	61 %	86 %	79 %
$B_3$	83 %	83 %		85 %
$B_4$	88 %	82 %		86 %

Times				
	Progol	FOIL	Indigo	TILDE
$B_1$	117039 s	4950 s		93 s
$B_2$	64256 s	9138 s	$\pm 900$ s	355 s
$B_3$	41788 s	0.5 s		221 s
$B_4$	40570 s	0.5 s		651 s

Theory complexity				
	Progol	FOIL		TILDE
$B_1$	24.3	24		8.3 (8.8)
$B_2$	11.2	49		10.2
$B_3$	11.1	54		8.3
$B_4$	9.9	46		11.0 (19.9)

Table 1: Accuracies, times and complexities of theories found by Progol, FOIL and TILDE

been tested on the Mutagenesis data. Figures for Progol and FOIL have been taken from [Srinivasan *et al.*, 1995], those for Indigo from [Geibel and Wyszotzki, 1996].

We conclude that TILDE’s results are at a par with Progol’s, as far as accuracy is concerned. The one result given for Indigo suggests, however, that it is possible to do better. Time complexity for TILDE is much better than Progol’s; compared to FOIL it is sometimes better, sometimes worse.

Theory complexities are hard to compare because of the different format of the theory. For TILDE, the average number of nodes of the induced trees is given; for Progol and FOIL, it is the number of literals. Problems when comparing these results are that nodes can contain more than one literal (for some cases the number of literals in the tree has been added between parentheses), and that TILDE learns one theory for both classes at the same time. Transforming the tree into a Prolog program and then counting only the rules for one class might yield a better way of comparing the theories, but the Prolog representation of a logical decision tree is inherently more complex, as many literals are duplicated, so it seems that this comparison would be unfair as well. One might as well transform the Prolog program returned by FOIL or Progol into an LDT and compare the complexity of the LDT representations, which would favour TILDE.

In the light of these problems and the results in table 1, it seems reasonable to say that TILDE performs better than FOIL (the difference is quite large here), but in relation to Progol it probably performs slightly worse. An interesting conclusion that can be drawn from these experiments is that, although several authors (e.g. [Boström, 1995], [Watanabe and Rendell, 1991]) have mentioned the fact that rule based systems return on the average a more compact theory than decision tree induction systems, the actual induction method that is used probably has more influence on this than the representation itself. Progol performs an exhaustive search, while FOIL and TILDE do not; this may well have a larger impact on the theory complexity than the fact that it is rule-based.

Background	No lookahead		Lookahead	
	accuracy (%)	time (s)	accuracy (%)	time (s)
Background 1	73.5	51	74.5	93
Background 2	78.3	271	79.4	355
Background 3	85.1	183	84.6	221
Background 4	84.0	104	86.1	651

Table 2: Comparison of TILDE’s performance with and without lookahead, for the Mutagenesis dataset

### 6.1.2 The Impact of Using Lookahead

We have also investigated the use of lookahead, which in this case seems particularly useful when bonds are added. The bonds themselves seldomly cause any significant gain, but they provide links to atoms that may. The following table compares the results with/without allowing lookahead on *bond*-literals. A lookahead of at most one literal was allowed; i.e. when an atom occurs in the tree, TILDE can test its immediate neighbours, but no atoms that are further removed. It is in principle possible to extend this further, but one neighbouring atom should be enough to make the use of bonds possible.

Table 2 suggests that lookahead may offer a slight advantage, but the results are not at all convincing.

## 6.2 Mesh Dataset

The Mesh dataset, since its introduction in ILP by [Dolšak and Muggleton, 1992], has been used several times as a benchmark to compare ILP systems. In many engineering applications, objects are described using finite element meshes. The resolution of such a mesh depends on the shape of the object and of neighbouring objects. The learning task is to find a relationship between the shape of an object and the resolution of its mesh. More specifically, for each edge of the object the number of parts the edge is divided in should be predicted. Because this number depends not only on the edge itself, but also on neighbouring edges, the learning task is a typical ILP task.

Table 3 compares TILDE’s performance on this dataset with several state-of-the-art systems. FOIL [Quinlan, 1993b] is a general-purpose ILP system. FFOIL [Quinlan, 1996] is a variant of it that can only learn functional definitions, but is very good at that. FORS [Karalič, 1995] is also specialized for learning functional definitions. Indigo [Geibel and Wysotzki, 1996] uses the transformational approach to ILP: it first transforms the learning data into a propositional representation. Then, a propositional decision tree induction system is used for the actual induction process. Figures for this table were copied from [Quinlan, 1996] (FOIL, FFOIL, FORS) and [Geibel and Wysotzki, 1996] (Indigo).

## 6.3 Musk Dataset

The musk dataset was studied by Dietterich *et al.* [Dietterich *et al.*, 1996], who donated it to the UCI repository [Merz and Murphy, 1996]. Dietterich used these data to study the so-called multiple instance problem: an example corresponds to multiple feature vectors, and is to be classified as positive if *at least one* of its feature vectors has certain properties. In the specific case of the musk dataset, examples are molecules which can have different conformations. The molecules are to be classified into musk and non-musk molecules.

	TILDE	FOIL	Indigo 1	Indigo 2	FFOIL	FORS
A	22	16	21	21	21	22
B	9	9	9	14	15	12
C	7	8	9	9	11	8
D	16	10	41	18	22	16
E	45	16	27	33	54	29
Total	99	59	107	95	123	87
%	36	21	38	34	44	31

Table 3: Comparison of TILDE’s performance with other systems for the Mesh dataset

Although the musk activity of a molecule can be measured, and different possible conformations of the molecule can be computed, it is not known which of these conformations causes the musk activity. It is sufficient for a molecule to have one musk conformation, in order for the molecule to be musk. So, if a molecule is a musk, it is only known that at least one of its conformations must be musk; it is not known which ones.

The multiple instance problem poses problems for propositional learners such as C4.5 or neural networks. There is a feature vector for each conformation, but the class of each conformation (musk / non-musk) is not known. Dietterich *et al.* [Dietterich *et al.*, 1996] developed a number of algorithms that explicitly deal with the multiple instance problem, thereby practically solving the problem for those cases where the theory can be represented as a set of axis-parallel rectangles (APR’s).

Inductive logic programming systems do not suffer from the multiple instance problem. The use of quantified variables in predicate logic makes it possible to have rules such as “if there is an  $X$  with the following properties: . . . , then the example is positive”. In the musk problem,  $X$  would simply be a conformation. For this reason, we expect TILDE (or any ILP system) to have no serious problems with the Musk dataset.

The Musk data are divided into 2 datasets: one small dataset (Musk1, where only the most important conformations are given for each molecule) and a large one (Musk2, all conformations given for each molecule). All feature vectors have 166 attributes (not counting the class attribute and tuple identifiers), which we will refer to as  $A_i$ ,  $i = 1..166$ . The large data set consists of four megabytes of data; its logical representation in TILDE is about 25 megabytes. According to ILP standards, this is a very large database.

### 6.3.1 Results for the Musk1 Dataset

On the Musk1 dataset, we have tried out several language biases.

- $\mathcal{L}_1$ : features of several conformations of a molecule can be tested
- $\mathcal{L}_2$ : all the tests in a tree are about a single conformation of the example;  $\mathcal{L}_2 \subset \mathcal{L}_1$ .
- $\mathcal{L}_3$ : as  $\mathcal{L}_2$ , but tests are used that check whether a value belongs to a certain interval, instead of simple inequality tests

$\mathcal{L}_1$  is the most simple language bias, in the sense that it gives the system the least information about the form of the theory that can be found. It allows rules such as

*if there is a conformation  $X$  for which  $A_{129} < -179$ , and there is a conformation  $Y$  for which  $A_6 < -178$ , then the molecule is non-musk*

$X$  and  $Y$  need not be the same conformation, which seems a bit counterintuitive, as the musk activity of a molecule can usually be traced back to one conformation. The second language bias demands that the rule tests only one conformation, e.g.

*if there is a conformation  $X$  for which  $A_{129} < -179$  and  $A_6 < -178$ , then the molecule is non-musk*

$\mathcal{L}_3$ , finally, makes use of intervals instead of inequality tests; e.g.

*if there is a conformation  $X$  for which  $A_{129} \notin [-\infty, -179]$ ,  $A_6 \in [-178, 73]$ ,  $A_{68} \notin [26, \infty]$  and  $A_7 \in [17, \infty]$  then the molecule is non-musk*

The constants used as boundaries have been found using TILDE’s built-in discretization procedure. Discretization has been performed separately for each of the 166 attributes. The number of bounds that the discretization procedure should return, is a parameter of the system; it was set to 1, 2, 3 or 5 in our experiments. The number of intervals is always the number of bounds + 1. The figures shown below have been obtained with a tenfold cross-validation. (When using 1 bound,  $\mathcal{L}_2 = \mathcal{L}_3$ ; therefore the test with  $\mathcal{L}_3$  has been skipped).

	1 bound	2 bounds	3 bounds	5 bounds
$\mathcal{L}_1$	79.3 %	85.9 %	83.7 %	83.7 %
$\mathcal{L}_2$	77.3 %	85.9 %	87.0 %	82.6 %
$\mathcal{L}_3$	(77.3 %)	86.9 %	81.7 %	79.7 %

The following table, taken from [Dietterich *et al.*, 1996], allows to compare TILDE’s performance with that of other algorithms. The algorithms marked with an asterisk are those that have been adapted specifically for the multiple-instance problem.

Algorithm	% correct
*iterated-discrim APR	92.4
*GFS elim-kde APR	91.3
GFS elim-count APR	90.2
GFS all-positive APR	83.7
all-positive APR	80.4
backpropagation	75.0
C4.5 (pruned)	68.5

An interesting result is that the use of  $\mathcal{L}_1$  resulted in theories that are approximately as good as those found with  $\mathcal{L}_2$ , but are not part of it. More precisely, the algorithm could be expected to automatically find rules where most or all of the tests are about one conformation. Instead, it found other rules, which seem to perform equally well though. The difference between using inequality tests and using intervals is also small. This suggests that even with a rather inappropriate bias, reasonably good results can be obtained.

In the following table, the algorithms are partitioned along two dimensions: appropriateness of bias and ability to cope with the multiple instance problem. For each class of algorithms, the best result on the Musk1 dataset is shown.

	no mult. inst.	mult. inst.
no appr. bias	75.0	87.0
appr. bias	90.2	92.4

These results lend further support to Dietterich’s claim that the worse performance of C4.5 and backpropagation networks, compared to APR’s, can be attributed to both the inability of the former to cope with the multiple instance problem, and the fact that these systems search less actively for an APR representation. It seems that the multiple instance problem accounts for a large part of the difference. This becomes even more clear if we compare TILDE to C4.5, the system that it resembles most; there is a difference of about 20% in accuracy.

Although using an appropriate bias also increases predictive accuracy a lot (compare non-multiple instance APR-algorithms with neural nets), this is not reflected in the experiments with TILDE. A possible explanation for the fact that changing the language bias in TILDE makes little difference, is that even when only APR-based theories can be found, TILDE’s search heuristics are not optimized for this specific kind of theories, nor is the discretization algorithm it uses. Changing the language bias is not only a matter of syntax, but also of search strategy, and this aspect is ignored in TILDE’s language bias specification.

### 6.3.2 Results for the Musk2 Dataset

Finally, we have run two experiments on the Musk2 dataset. This set contains much more irrelevant information (more conformations per molecule), making the problem a bit harder. The test was run once with language bias  $\mathcal{L}_2$ , using 3 discretization bounds per attribute, and a second time with bias  $\mathcal{L}_3$ , using 2 bounds per attribute (which defines 3 intervals, i.e. 3 possible tests per attribute, as in the first test).

The following table compares TILDE’s results with those mentioned in [Dietterich *et al.*, 1996].

Algorithm	% correct
iterated-discrim APR	89.2
GFS elim-kde APR	80.4
TILDE ( $\mathcal{L}_3$ )	<b>79.4</b>
TILDE ( $\mathcal{L}_2$ )	<b>77.5</b>
GFS elim-count APR	75.5
all-positive APR	72.6
backpropagation	67.7
GFS all-positive APR	66.7
C4.5 (pruned)	58.8

It shows that also for this “more difficult” dataset, where the multiple instance problem is more prominent, TILDE’s results are comparable with those of the APR-algorithms; only Dietterich’s iterated-discrimination algorithm clearly outperforms the others.

## 7 Conclusions and Related Work

Although the divide-and-conquer paradigm has been largely ignored in the field of ILP, it seems that in many cases it offers a number of advantages over the covering paradigm. In an attempt to fill this gap, we have introduced and formalized the notion of logical decision trees. An algorithm was presented for top-down induction of logical decision trees, as well as a practical implementation of this algorithm that takes advantage of the current state of the art in both propositional and relation learning.

Experiments were performed with this learning system, showing that logical decision trees have good potential for finding simple theories that have high predictive accuracy, and this for a broad range of problems. Moreover, they can be induced

quite fast (in fact, the current implementation is only a rather inefficient prototype). And finally, because their ability to learn several classes at once, they seem to be particularly advantageous for multi-class classification tasks.

Logical decision trees are a generalization of Watanabe's structural decision trees [Watanabe and Rendell, 1991]. Our work extends Watanabe's in the sense that the idea of structural decision trees is generalized, formalized, and integrated with inductive logic programming techniques (use of background, declarative language bias specification, ...) as well as with state-of-the-art decision tree induction techniques (as used by C4.5). The fact that more than one literal can occur in a node of a logical decision tree allows TILDE to handle determinate literals, lookahead facilities etc. more easily than Watanabe's system could.

This work is also related to [Boström, 1995], on induction of logic programs using the divide-and-conquer paradigm. The main difference is that we use a different logical setting (learning from interpretations instead of learning from entailment), aimed specifically at classification rather than logic program synthesis. Also, Boström seems to make an implicit assumption concerning non-overlapping literals when refining a clause; it is not clear to what extent this approach is valid in general.

## Acknowledgements

Hendrik Blockeel is supported by the Flemish Institute for the Promotion of Scientific and Technological Research in the Industry (IWT). Luc De Raedt is supported by the Belgian National Fund for Scientific Research. This work is also part of the European Community Esprit project no. 20237, Inductive Logic Programming 2.

The authors would like to thank Luc Dehaspe for many interesting discussions, and for pointing to the possibility of using TDIDT for ILP; and Wim Van Laer for providing highly reusable code from the ICL system.

## References

- [Bergadano and Giordana, 1988] F. Bergadano and A. Giordana. A knowledge intensive approach to concept induction. In *Proceedings of the 5th International Workshop on Machine Learning*. Morgan Kaufmann, 1988.
- [Boström, 1995] H. Boström. Covering vs. divide-and-conquer for top-down induction of logic programs. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1995.
- [Bratko, 1990] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1990. 2nd Edition.
- [Catlett, 1991] J. Catlett. On changing continuous attributes into ordered discrete attributes. In Yves Kodratoff, editor, *Proceedings of the 5th European Working Session on Learning*, volume 482 of *Lecture Notes in Artificial Intelligence*, pages 164–178. Springer-Verlag, 1991.
- [Clark and Niblett, 1989] P. Clark and T. Niblett. The CN2 algorithm. *Machine Learning*, 3(4):261–284, 1989.
- [De Raedt and Dehaspe, 1996] L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 1996. To appear.
- [De Raedt and Džeroski, 1994] L. De Raedt and S. Džeroski. First order  $jk$ -clausal theories are PAC-learnable. *Artificial Intelligence*, 70:375–392, 1994.

- [De Raedt and Van Laer, 1995] L. De Raedt and W. Van Laer. Inductive constraint logic. In *Proceedings of the 5th Workshop on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1995.
- [De Raedt, 1996] L. De Raedt. Induction in logic. In R.S. Michalski and Wnek J., editors, *Proceedings of the 3rd International Workshop on Multistrategy Learning*, pages 29–38, 1996.
- [Dietterich *et al.*, 1996] Thomas G. Dietterich, Richard H. Lathrop, and Tomás Lozano-Pérez. Solving the multiple-instance problem with axis-parallel rectangles. *Artificial Intelligence*, 1996. In press.
- [Dolšák and Muggleton, 1992] B. Dolšák and S. Muggleton. The application of Inductive Logic Programming to finite element mesh design. In S. Muggleton, editor, *Inductive logic programming*, pages 453–472. Academic Press, 1992.
- [Dougherty *et al.*, 1995] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In A. Prieditis and S. Russell, editors, *Proc. Twelfth International Conference on Machine Learning*. Morgan Kaufmann, 1995.
- [Fayyad and Irani, 1993] U.M. Fayyad and K.B. Irani. Multi-interval discretization of continuous-valued attributes for classification learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1022–1027, San Mateo, CA, 1993. Morgan Kaufmann.
- [Geibel and Wysotzki, 1996] P. Geibel and F. Wysotzki. Learning relational concepts with decision trees. In L. Saitta, editor, *Proceedings of the 13th International Conference on Machine Learning*, pages 166–174, July 1996.
- [Karalič, 1995] A. Karalič. *First Order Regression*. PhD thesis, Faculty of Electrical Engineering and Computer Science, University of Ljubljana, Slovenia, 1995.
- [Lavrač and Džeroski, 1994] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.
- [Merz and Murphy, 1996] C.J. Merz and P.M. Murphy. UCI repository of machine learning databases [<http://www.ics.uci.edu/~mllearn/mlrepository.html>], 1996. Irvine, CA: University of California, Department of Information and Computer Science.
- [Michalski *et al.*, 1986] R. Michalski, I. Mozetic, J. Hong, and N. Lavrac. The AQ15 inductive learning system: an overview and experiments. In *Proceedings of IMAL 1986*, Orsay, 1986. Université de Paris-Sud.
- [Muggleton and Page, 1994] S. Muggleton and D. Page. Beyond first order learning: inductive learning with higher order logic. Technical report, OUCL Programming Research Group Technical Report PRG-TR-13-94, 1994.
- [Muggleton, 1995] S. Muggleton. Inverse entailment and progol. *New Generation Computing*, 13, 1995.
- [Quinlan, 1986] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.
- [Quinlan, 1993a] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann series in machine learning. Morgan Kaufmann, 1993.

- [Quinlan, 1993b] J.R. Quinlan. FOIL: A midterm report. In P. Brazdil, editor, *Proceedings of the 6th European Conference on Machine Learning*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 1993.
- [Quinlan, 1996] J. R. Quinlan. Learning first-order definitions of functions. *Journal of Artificial Intelligence Research*, 5:139–161, October 1996.
- [Srinivasan *et al.*, 1995] A. Srinivasan, S.H. Muggleton, and R.D. King. Comparing the use of background knowledge by inductive logic programming systems. In L. De Raedt, editor, *Proceedings of the 5th International Workshop on Inductive Logic Programming*. IOS Press, 1995.
- [Van Laer *et al.*, 1996] W. Van Laer, S. Dzeroski, and L. De Raedt. Multi-class problems and discretization in ICL (extended abstract). In *Proceedings of the MLnet Familiarization Workshop on Data Mining with Inductive Logic Programming (ILP for KDD)*, 1996.
- [Watanabe and Rendell, 1991] L. Watanabe and L. Rendell. Learning structural decision trees from examples. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 770–776, 1991.