ELSEVIER Artificial Intelligence 101 (1998) 285–297

Artificial Intelligence

Top-down induction of first-order logical decision trees

Hendrik Blockeel *, Luc De Raedt [1]

Katholieke Universiteit Leuven, Department of Computer Science Celestijnenlaan 200A, 3001 Heverlee, Belgium

Received 11 March 1997; received in revised form 25 March 1998

Abstract

A first-order framework for top-down induction of logical decision trees is introduced. The expressivity of these trees is shown to be larger than that of the flat logic programs which are typically induced by classical ILP systems, and equal to that of first-order decision lists. These results are related to predicate invention and mixed variable quantification. Finally, an implementation of this framework, the TILDE system, is presented and empirically evaluated. © 1998 Elsevier Science B.V. All rights reserved.

1. Introduction

Top-down induction of decision trees (TDIDT) [28] is the best known and most successful 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 (e.g., AQ [21], CN2 [6]), which are based on covering strategies (cf. [4]). 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. The main reason why divide-and-conquer approaches are not yet so popular within first-order learning, lies in the discrepancies between the clausal representation employed within inductive logic programming and the structure underlying a decision tree.

* Corresponding author. Email: hendrik@cs.kuleuven.ac.be.

[1] Email: Luc.DeRaedt@cs.kuleuven.ac.be.

The main contributions of this paper are threefold. First, we introduce a logical representation for relational decision trees and show how to correctly translate first-order logical decision trees into logic programs. The resulting programs contain invented predicates and contain both universal and existential quantifiers through the use of negation. We show that this makes logical decision trees more expressive than the programs typically induced by ILP systems, and equally expressive as first-order decision lists.

Second, in our framework each single example is a Prolog program, which means that we are learning from interpretations (cf. [7,9]). This learning setting has the classical TDIDT setting (using attribute value representations) as a special case. This makes it possible to elegantly upgrade attribute-value learners to the first-order logic context. As an illustration, we implemented TILDE, an upgrade of Quinlan's C4.5 [27].

Thirdly, we report on a number of experiments on large data sets with TILDE which clearly show that TILDE is competitive both in terms of efficiency and accuracy with state-of-the-art inductive logic programming systems such as PROGOL [23] and FOIL [29].

This text is organized as follows. In Section 2, we briefly discuss the ILP setting that will be used. In Section 3, we introduce first-order logical decision trees and discuss their properties. We show in Section 4 how TDIDT can be upgraded to first-order learning, and empirically evaluate our algorithm in Section 5. Finally, in Section 6 we conclude and touch upon related work.

2. The learning problem

We assume familiarity with Prolog (see, e.g., [5]) and standard logic programming terminology (e.g., [18]). In our framework, each example is a Prolog program that encodes its specific properties. Furthermore, each example is classified into one of a finite set of possible classes. One may also specify background knowledge in the form of a Prolog program.

More formally, the problem specification is:

Given: a set of classes C, a set of classified examples E and a background theory B,
Find: a hypothesis H (a Prolog program), 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 $c' \in C - \{c\}$.

This setting is known under the label *learning from interpretations*[2] [7,9]. Notice that within this setting, one always learns first-order definitions of propositional predicates (the classes). An implicit assumption is that the class of an example depends on that example only, not on any other examples. This is a reasonable assumption for many classification problems, though not for all; it precludes e.g. recursive concept definitions.

Example 1. An engineer has to check a set of machines. A machine consists of several parts that may be 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 (class

[2] The interpretation corresponding to each example e is then the minimal Herbrand model of $B \wedge e$.

sendback). If all the worn parts can be replaced, it is to be fixed (fix). If there are no worn parts, nothing needs to be done (ok).

Given the following set of examples (each example corresponds to one machine) and background knowledge:

Example 1	**Example 2**	**Example 3**	**Example 4**
class(fix)	*class(sendback)*	*class(sendback)*	*class(ok)*
worn(gear)	worn(engine)	worn(wheel)	
worn(chain)	worn(chain)		

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

a Prolog rule for the sendback class is:

```
class(send_back):-worn(X), not_replaceable(X)
```

3. First-order logical decision trees

A first-order logical decision tree (FOLDT) is a *binary* decision tree in which (1) the nodes of the tree contain a conjunction of literals, and (2) different nodes may share variables, under the following restriction: a variable that is introduced in a node (which means that it does not occur in higher nodes) must not occur in the right branch of that node.[3] An example of a logical decision tree is shown in Fig. 1. It encodes the target hypothesis of Example 1.

We use the following notation: a tree T is either a leaf with class k, in which case we write $T = \textbf{leaf}(k)$, or it is an internal node with conjunction c, left branch l and right branch r, in which case we write $T = \textbf{inode}(c, l, r)$.

Fig. 2 shows how to use FOLDTs for classification. As an example e is a Prolog database, a test in a node corresponds to checking whether a query $\leftarrow C$ succeeds in $e \wedge B$ (with B the background knowledge). Note that it is not sufficient to use for C the conjunction *conj* in the node itself. Since *conj* may share variables with nodes higher in the tree, C consists of several conjunctions that occur in the path from the root to the current node. Therefore, when an example is sorted to the left, C is updated by adding *conj* to it. When sorting an example to the right, C need not be updated: a failed test never introduces new variables.

[3] The need for this restriction follows from the semantics of the tree. A variable X that is introduced in a node, is existentially quantified within the conjunction of that node. The right subtree is only relevant when the conjunction fails ("there is no such X"), in which case further reference to X is meaningless.

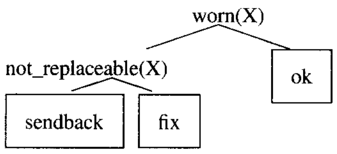


Fig. 1. Logical decision tree encoding the target hypothesis of Example 1.

```
procedure classify($e$ : example) returns class:
    $C := true$
    $N :=$ root
    while $N \neq$ leaf($c$) do
        let $N =$ inode($conj, left, right$)
        if $C \wedge conj$ succeeds in $B \wedge e$
        then $C := C \wedge conj$
             $N := left$
        else $N := right$
    return $c$
```

Fig. 2. Classification of an example using an FOLDT (with background knowledge B).

Fig. 3 shows how an equivalent logic program can be derived from a FOLDT.[4] With each internal node in the tree a clause defining a newly invented nullary predicate is associated, as well as a query. This query can make use of the predicates defined in higher nodes. With leaves only a query is associated, no clause.

The queries are defined in such a way that the query associated with a node succeeds for an example if and only if that node is encountered during classification of that example. Therefore, if a query associated with a *leaf* succeeds, the leaf indicates the class of the example. The clauses define invented predicates that are needed to express these queries. Thus the queries associated with leaves, together with these clauses, form a logic program that is equivalent to the tree.

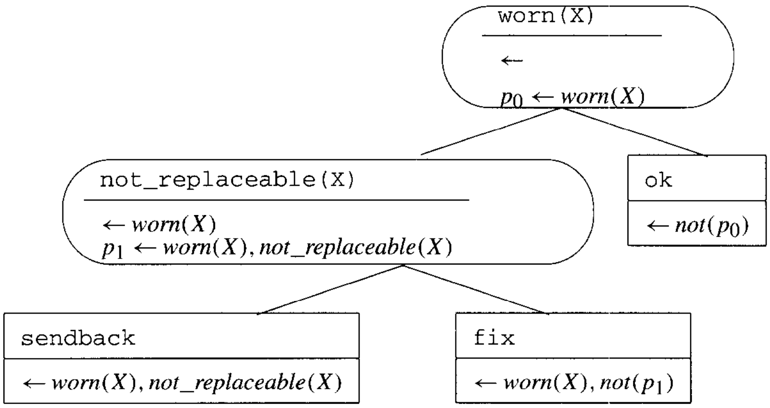
An important point is that the algorithm adds to a query the negation of the invented predicate p_i, and not the negation of the conjunction itself (see Fig. 3: the query $\leftarrow Q, not(p_i)$ and not $\leftarrow Q, not(conj)$ is associated with the right subtree of T). Indeed, the queries of the left and right subtree should be complementary: for each example sorted into this node (i.e., $\leftarrow Q$ succeeds), exactly one of both queries should succeed. Now, $\leftarrow Q, conj$ (which is equivalent to $\leftarrow Q, p_i$) and $\leftarrow Q, not(p_i)$ are complementary, but $\leftarrow Q, conj$ and $\leftarrow Q, not(conj)$ are not, when $conj$ shares variables with Q. For instance, in the interpretation $\{q(1), p(1), q(2)\}$ both $\leftarrow q(X), p(X)$ and $\leftarrow q(X), not(p(X))$ succeed.

[4] In this text "logic programs" means *normal* logic programs, i.e., programs that may contain negative literals in the body of clauses. When the latter is not allowed, we will explicitly refer to *definite* logic programs.

```
procedure associate($T$ : foldt, $\leftarrow Q$ : query):
    if $T$ = inode($conj$, $left$, $right$) then
        assign a unique predicate $p_i$ to this node
        assert $p_i \leftarrow Q, conj$
        associate($left$, ($\leftarrow Q, conj$))
        associate($right$, ($\leftarrow Q, not(p_i)$))
    else
        let $T$ = leaf($k$)
        assert $class(k) \leftarrow Q$

procedure deriveLogicProgram($T$: foldt):
    associate($T$, $\leftarrow$)
```

Fig. 3. Mapping FOLDT's onto logic programs.


worn(X)
$\leftarrow$
$p_0 \leftarrow worn(X)$
not_replaceable(X)
$\leftarrow worn(X)$
$p_1 \leftarrow worn(X), not\_replaceable(X)$
ok
$\leftarrow not(p_0)$
sendback
$\leftarrow worn(X), not\_replaceable(X)$
fix
$\leftarrow worn(X), not(p_1)$

p_0	$\leftarrow$	$worn(X)$
p_1	$\leftarrow$	$worn(X), not\_replaceable(X)$
$class(sendback)$	$\leftarrow$	$worn(X), not\_replaceable(X)$
$class(fix)$	$\leftarrow$	$worn(X), not(p_1)$
$class(ok)$	$\leftarrow$	$not(p_0)$

Fig. 4. The tree of Fig. 1, with associated clauses and queries added; and the logic program derived from the tree.

Fig. 4 shows the result of applying the algorithm in Fig. 3 to our running example. Consider in Fig. 4 the node containing *not_replaceable*(X). The query associated with the right subtree of this node contains $not(p1)$ and not $not(not\_replaceable(X))$. Indeed, in the latter case the query would succeed if there is a worn part in the machine that is replaceable, while it ought to succeed if there are worn parts in the machine, but *all* of them are replaceable.

Because a literal and its negation are not complementary, adding a literal is not equivalent to adding the negation of the literal while at the same time switching the branches. This means it may be interesting to allow negated literals in queries. In our example, *not_replaceable*(X) partitions the set $\{E_1, E_2, E_3\}$ (see Example 1) into $\{\{E_1\}, \{E_2, E_3\}\}$, while *replaceable*($X$) would partition it into $\{\{E_1, E_2\}, \{E_3\}\}$.

This is an important difference with the propositional case, where a test (e.g., $X < 5$) and its negation ($X \geqslant 5$) always generate the same partition. In the first-order context they may generate different partitions. This fact and its influence on the tree-to-ruleset conversion are new findings that have not been mentioned in existing literature on relational decision trees (e.g., [17,35]), but are important for a correct understanding of their semantics.

While auxiliary predicates p_i are needed in the logic program, in Prolog they can be avoided by using the cut operator. We then get a *first-order decision list* (FODL): [5]

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

In general, a tree can always be transformed into a decision list and vice versa. The following functions *dl* and *tr* define such mappings (@ represents concatenation of lists): [6]

$$dl(T) = dl'(T, true)$$

$$dl'(\mathbf{leaf}(c), PC) = [(\texttt{class}(c)\ \texttt{:-}\ PC,\ \texttt{!})]$$

$$dl'(\mathbf{inode}(conj, left, right), PC) = dl'(left, (PC, conj))\,@\,dl'(right, PC)$$

$$tr([(\texttt{class}(c)\ \texttt{:-}\ conj,\ \texttt{!})|Rest] = \mathbf{inode}(conj, \mathbf{leaf}(c), tr(Rest))$$

$$tr([\texttt{class}(c)]) = \mathbf{leaf}(c).$$

This establishes the equivalence of FOLDTs and FODLs. We now turn to the relationship with logic programs.

For classification, we consider only non-recursive logic programs as hypotheses. The hypotheses essentially define the target predicate *class*. They may also define *invented* predicates which do not occur in the background theory. We assume here that the background is not changed by the learner and not part of the hypothesis. Then the hypotheses H_1 and H_2 are equivalent iff for all backgrounds B, $B \wedge H_1$ assigns the same class to any possible example as $B \wedge H_2$. We call a hypothesis in the form of a logic program *flat* if it contains no invented predicates, otherwise we call it *layered*.

A hypothesis in the form of a layered *definite* logic program can always be transformed into a flat definite logic program by unfolding calls to invented predicates. Layered *normal*

[5] In a decision list rules are ordered, and only the first rule of which the body succeeds is applied.

[6] In *tr*, we make use of the fact that nodes can contain conjunctions of literals. If only one literal is allowed in each node, the conversion is still possible but more complex.

logic programs, however, cannot always be transformed to flat normal logic programs in this way. Unfolding a negative literal for an invented predicate may introduce universally quantified variables in the body, which is beyond the expressive power of logic program clauses (by definition, variables not occurring in the head of a clause are existentially quantified in its body).

For instance, unfolding the layered logic program of our running example yields:

$$\begin{array}{ll}
\texttt{lass(ok)} & \leftarrow \forall X : \neg worn(X) \\
\texttt{class(sendback)} & \leftarrow \exists X : worn(X) \wedge not\_replaceable(X) \\
\texttt{class(fix)} & \leftarrow \exists X : worn(X) \wedge \\
 & \quad \forall Y : (\neg worn(Y) \vee \neg not\_replaceable(Y)).
\end{array}$$

Since any flat logic program, when written in this format, only contains existential quantifiers (by definition of logic program clauses), no flat hypothesis exists that is equivalent to this theory (e.g., $\forall X \neg worn(X)$ cannot be written with only existential variables).[7]

We conclude from the above that FOLDTs can always be converted to layered normal logic programs (Fig. 3 gives the algorithm), but not always to flat normal logic programs.

Finally, observe that a flat logic program that predicts only one class for a single example (which is not a restrictive condition in the context of classification) can always be transformed into an equivalent decision list by just adding a cut to the end of each clause.

As FODLs and FOLDTs can be converted into one another, and flat logic programs can be converted into FODLs or FOLDTs but not the other way around, we have the following property:

> *In the learning from interpretations setting, the set of theories that can be represented by FOLDTs is a strict superset of the set of theories that can be represented by flat normal logic programs, and is equivalent to the set of theories that can be represented by FODLs.*

All this means that systems that induce trees or decision lists (examples of the latter are FFOIL [26] and FOIDL [22]) can find theories that cannot be found by systems that induce flat (normal or definite) logic programs (e.g., FOIL [29], PROGOL [23], and most other ILP systems). This extends the classical claim that the use of cuts allows for a more compact representation (see, e.g., [22]) with the claim that also a greater expressivity is achieved. The same expressivity could be achieved by classical ILP systems if they allow negation and perform predicate invention (or if they allow Prolog's unsound negation as failure: \+ with as argument a conjunction of literals, which essentially amounts to the same).

[7] Using \+ to denote Prolog's unsound version of negation as failure (which does not check the groundness of its argument), one might remark that, e.g., `class(fix) :- worn(X), \+ (worn(Y), not_replaceable(Y))` correctly computes the class `fix`. However, we do not call this a flat program. Operationally, \+ starts a subquery. Declaratively, the meaning is $class(fix) \leftarrow worn(X), \neg(\exists Y(worn(Y), not\_replaceable(Y)))$ which is beyond the expressive power of a normal clause.

In this respect we want to mention Bain et al.'s nonmonotonic induction method [1]. The hypotheses generated with this method have a structure similar to that of FOLDTs (when only two classes are involved), in that the induced theory is typically also layered through the use of invented predicates, and the invented predicates occur as negative literals in the clauses, accommodating exceptions to them. However, in Bain et al.'s framework the learning method is incremental and rule-based.

4. Top-down induction of logical decision trees

In this section we present the TILDE system, which induces FOLDTs from data. TILDE was implemented to illustrate that in the learning from interpretations setting, upgrading good propositional algorithms to the first-order logic context is feasible and results in highly performant systems. It employs the basic TDIDT algorithm, and behaves exactly as C4.5 [27] (for binary attributes) in that it uses the same heuristics, post-pruning algorithm etc. The system is available for academic purposes upon request.

The only point where our algorithm differs is in the computation of the set of tests to be considered at a node. To this aim, it employs a classical refinement operator under θ-subsumption [24,25]. Such an operator ρ maps clauses onto sets of clauses, such that for any clause c and $\forall c' \in \rho(c)$, c θ-subsumes c'. A clause c_1 θ-subsumes another clause c_2 if and only if there is a variable substitution θ such that $c_1\theta \subseteq c_2$.

In order to refine a node with associated query $\leftarrow Q$, TILDE computes $\rho(\leftarrow Q)$ and chooses that query $\leftarrow Q'$ that results in the best split. [8] The conjunction put in the node consists of $Q' - Q$, i.e., the literals that have been added to Q in order to produce Q'.

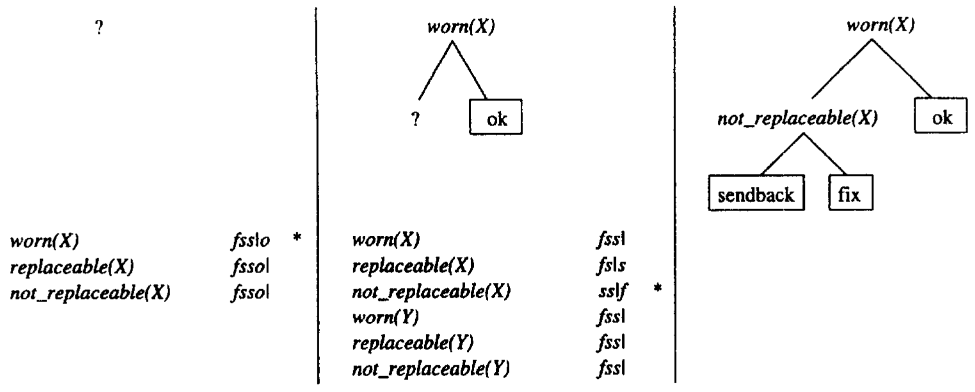
The specific refinement operator that is to be used, is defined by the user in a PROGOL-like manner [23]. A set of specifications of the form `rmode(`*n*: *conjunction*`)` is provided, indicating which conjunctions can be added to a query, the maximal number of times the conjunction can be added (n), and the modes and types of the variables in it.

For instance, `rmode(8:(p(+X, -Y, Z), q(Z)))` specifies that the conjunction $p(X, Y, Z), q(Z)$ can occur maximally 8 times in the associated query of a node, and allows possible unifications of the variables of p with other variables in the query (`+X`, `-Y` and `Z` indicating that X must be unified with an existing variable, Y may but need not be unified, and Z is a new variable).

In addition to this, so-called lookahead specifications can be provided. These allow TILDE to perform several successive refinement steps at once. This alleviates the well-known problem in ILP that a refinement may not yield any gain, but may introduce new variables that are crucial for classification. By performing successive refinement steps at once, TILDE can look ahead in the refinement lattice and discover such situations.

For instance, `lookahead(p(X,Y,Z), r(X))` specifies that whenever p is added, additional refinement by adding $r(X)$ (with X the first variable in p) should be tried in the same refinement step.

[8] "The best split" means that subsets are obtained that are as homogeneous as possible with respect to the classes of the examples. An often used criterion is that the subsets have minimal class entropy. TILDE uses a variant of this criterion called *gain ratio* [27].



worn(X)	*fss|o*	*
replaceable(X)	*fsso|*	
not_replaceable(X)	*fsso|*	

worn(X)	*fss|*	
replaceable(X)	*fs|s*	
not_replaceable(X)	*ss|f*	*
worn(Y)	*fss|*	
replaceable(Y)	*fss|*	
not_replaceable(Y)	*fss|*	

Refinement operator specification:
```
rmode(5: replaceable(-X)).
rmode(5: not_replaceable(-X)).
rmode(5: worn(-X)).
```

Fig. 5. TILDE illustrated on the running example. Each step shows the partial tree that has been built, the literals that are considered for addition to the tree, and how each literal would split the set of examples. E.g., *fss*|*o* means that of four examples, one with class `fix` and two with class `sendback` are in the left branch, and one example with class `ok` is in the right branch. The best literal is indicated with an asterisk.

TILDE handles numerical data by means of a discretization algorithm that is based on Fayyad and Irani's [14] and Dougherty's [11] work, but extends it to first-order logic (see [34]). The algorithm accepts input of the form `discretize(Query, Var)`, with `Var` a variable occurring in `Query`. It runs `Query`, collecting all instantiations of `Var` that can be found, and finally generates discretization thresholds based on this set of instantiations. For more details on discretization and lookahead see [3,34].

Fig. 5 illustrates the TILDE algorithm on the running example.

5. Experimental evaluation

We have evaluated TILDE by performing experiments on several benchmark datasets: Mutagenesis [32], Musk [10,20], and Diterpene [12]. For all the experiments, TILDE's default parameters were used; only the choice of the number of thresholds for discretization, when applicable, was supplied manually. Full details on the experimental settings, as well as the datasets that were used (except for the Diterpene dataset, which we cannot make public), are available at `http://www.cs.kuleuven.ac.be/~ml/Tilde/Experiments/`.

In the Mutagenesis dataset each example consists of a structural description of a molecule (the atoms and bonds it consists of), and some numerical information describing the molecule as a whole. The molecules have to be classified into mutagenic and non-mutagenic ones. Table 1 compares TILDE's performance on this task with that of FOIL

Table 1
Accuracies, times and complexities of theories found by PROGOL, FOIL and TILDE for the Mutagenesis problem; averaged over tenfold crossvalidation. Times for TILDE were measured on a Sun SPARCstation-20, for the other systems on a Hewlett Packard 720. Because of the different hardware, times should be considered to be indicative rather than absolute

	Accuracies (%)				**Times (s)**				**Complexity (literals)**			
	B_1	B_2	B_3	B_4	B_1	B_2	B_3	B_4	B_1	B_2	B_3	B_4
PROGOL	76	81	83	88	117039	64256	41788	40570	24.3	11.2	11.1	9.9
FOIL	61	61	83	82	4950	9138	0.5	0.5	24	49	54	46
TILDE	75	79	85	86	41	170	142	352	8.8	14.4	12.8	19.9

(version 6.2) and PROGOL (actually P-Progol, Srinivasan's implementation in Prolog), as reported in [31] (four levels of background knowledge B_i are distinguished there, each one being a superset of its predecessor). For the numerical data, TILDE's discretization procedure was used. Lookahead was allowed when adding *bond*-literals (addition of a bond typically does not lead to any gain, but enables inspection of nearby atoms).

From the table it can be concluded that TILDE efficiently finds theories with high accuracy. The complexity of the induced theories is harder to compare, because TILDE uses a radically different format to represent the theory. When simply counting literals, TILDE's theories are about as complex as PROGOL's, but clearly simpler than FOIL's. (Converting the trees towards decision lists and then counting literals yields much larger numbers (respectively 20.4, 34.0, 53.3 and 73.5 literals for B_1 to B_4) due to duplication of many literals, but this comparison method is biased in favour of rule induction systems.) The fact that TILDE finds more compact theories than PROGOL on B_1, although PROGOL performs an exhaustive search, can be attributed to the greater expressivity of FOLDTs.

With the Musk dataset, the main challenge was its size. We used the largest of the two Musk datasets available at the UCI repository [20]. This dataset is about 4.5 MB large and consists of only numerical data, which makes it a non-typical ILP application. On the other hand, it suffers from the so-called multiple instance problem (an example is described by multiple feature vectors, only one of which is relevant), which makes it hard for propositional learners as well. Dietterich et al.'s approach [10] is to adapt propositional learning algorithms to the multiple instance problem in the specific case of learning single axis-parallel rectangles (APRs). For ILP systems no adaptations are necessary. Still, TILDE's performance is comparable with most other algorithms discussed in [10], with only one (special-purpose) algorithm outperforming the others (Table 2). For the experiments with TILDE, all the numerical attributes were discretized.[9] The average running time of TILDE on one crossvalidation step was about 2 hours on a SUN Sparc Ultra-2.

The Diterpene dataset describes the ^{13}C-NMR spectra of a number of molecules; the aim is to classify molecules based on the information in these spectra. This is a multiple class problem (more than two classes). Several versions of the data are distinguished: purely

[9] The number of discretization bounds was determined by running experiments on the smaller dataset and choosing the number of bounds that works best on that set.

Table 2
Comparison of accuracy of theories obtained with TILDE with those of other systems on the Musk dataset

Algorithm	% correct
Iterated-discrim APR	89.2
GFS elim-kde APR	80.4
TILDE	**79.4**
Backpropagation network	67.7
C4.5	58.8

Table 3
Results with TILDE on the Diterpene dataset, making use of propositional data, relational data or both; standard errors are shown between parentheses

	FOIL	RIBL	TILDE
	acc.(%)	acc.(%)	acc.(%)
Prop	70.1	79.0	78.5 (1.3)
Rel	46.5	86.5	81.0 (1.0)
Both	78.3	91.2	90.4 (0.6)

propositional data (containing engineered features), relational data (non-engineered), and both. Best performance up till now was achieved by the RIBL system [13], an instance-based relational learner. Table 3 shows that TILDE achieves slightly lower accuracy than RIBL, but outperforms FOIL. Moreover, it returns a symbolic, interpretable (although complex) theory, in contrast to RIBL.

We want to stress that in all these experiments, we have compared TILDE's accuracies with the best known results, which were obtained with different systems. While not outperforming the best systems, TILDE consistently performs almost as well and is very efficient.

6. Conclusions and related work

We have observed that the TDIDT approach is very successful in propositional learning and might offer a number of advantages over the covering approach in ILP (cf. Boström [4]). In an attempt to make the TDIDT paradigm more attractive to ILP we have investigated the logical aspects of decision trees. The resulting framework should provide a sound basis for logical decision tree induction.

Our investigation shows that first-order logical decision trees are more expressive than the flat non-recursive logic programs typically induced by ILP systems for classification tasks, and that this expressive power is related to the use of cuts, or the use of negation combined with predicate invention. This in turn relates our work to some of the work on induction of decision lists and predicate invention [1,22,26], showing that these algorithms too have an expressivity advantage over algorithms inducing flat logic programs.

From a practical perspective, we have developed the TILDE system, of which Quinlan's C4.5 [27] with binary attributes is a special case (due to the learning from interpretations setting). Experiments show that logical decision trees have good potential for efficiently finding simple theories that have high predictive accuracy, and this for a broad range of problems.

Of the existing decision tree approaches in the field of relational learning [2,4,17,19, 35], STRUCT [35] and SRT [17] come closest to our approach. Boström's work [4] has in common with ours that he compared the covering and divide-and-conquer paradigms in the context of ILP. However, all this work has focused on induction techniques and has ignored the logical and representational aspects of decision trees, needed to fully understand the potential of this technique for first-order learning.

A lot of propositional TDIDT techniques (boosting [15,30], incremental learning [33], TDIDT-based clustering [8], ...) can be upgraded to the first-order case, which offers opportunities for future work.

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 Fund for Scientific Research, Flanders. This work is also part of the European Community Esprit project no. 20237, Inductive Logic Programming 2.

The authors wish to thank Luc Dehaspe for many interesting discussions, and for pointing to the possibility of using TDIDT for ILP; Wim Van Laer for providing highly reusable code from the ICL system; and the anonymous referees and Maurice Bruynooghe for their useful comments on this paper.

The Diterpene dataset was provided by Sašo Džeroski and Steffen Schulze-Kremer. The Musk dataset, originally used by Dietterich et al. [10], is available at the UCI repository [20]. The Mutagenesis dataset, made public by King and Srinivasan [32], is available at the ILP data repository [16].

References

[1] M. Bain and S. Muggleton, Non-monotonic learning, in: S. Muggleton (Ed.), Inductive Logic Programming, Academic Press, London, 1992, pp. 145–161.

[2] F. Bergadano and A. Giordana, A knowledge intensive approach to concept induction, in: Proceedings 5th International Workshop on Machine Learning Ann Arbor, MI, Morgan Kaufmann, San Mateo, CA, 1988.

[3] H. Blockeel and L. De Raedt, Lookahead and discretization in ILP, in: Proceedings 7th International Workshop on Inductive Logic Programming, Lecture Notes in Artif. Intell., Vol. 1297, Springer, Berlin, 1997.

[4] H. Boström, Covering vs. divide-and-conquer for top-down induction of logic programs, in: Proceedings 14th International Joint Conference on Artificial Intelligence (IJCAI-95), Montreal, Que., Morgan Kaufmann, San Mateo, CA, 1995.

[5] I. Bratko, Prolog Programming for Artificial Intelligence, 2nd ed., Addison-Wesley, Reading, MA, 1990.

[6] P. Clark and T. Niblett, The CN2 algorithm, Machine Learning 3 (4) (1989) 261–284.

[7] L. De Raedt, Induction in logic, in: R.S. Michalski and J. Wnek (Eds.), Proceedings 3rd International Workshop on Multistrategy Learning, 1996, pp. 29–38.

[8] L. De Raedt and H. Blockeel, Using logical decision trees for clustering, in: Proceedings 7th International Workshop on Inductive Logic Programming, Lecture Notes in Artif. Intell., Vol. 1297, Springer, Berlin, 1997, pp. 133–141.

[9] L. De Raedt and S. Džeroski, First-order jk-clausal theories are PAC-learnable, Artificial Intelligence 70 (1994) 375–392.

[10] T.G. Dietterich, R.H. Lathrop and T. Lozano-Pérez, Solving the multiple-instance problem with axis-parallel rectangles, Artificial Intelligence 89 (1–2) (1997) 31–71.

[11] J. Dougherty, R. Kohavi and M. Sahami, Supervised and unsupervised discretization of continuous features, in: A. Prieditis and S. Russell (Eds.), Proceedings Twelfth International Conference on Machine Learning, Morgan Kaufmann, San Mateo, CA, 1995.

[12] S. Džeroski, S. Schulze-Kremer, K.R. Heidtke, K. Siems and D. Wettschereck, Applying ILP to diterpene structure elucidation from 13C NMR spectra, in: Proceedings 6th International Workshop on Inductive Logic Programming, 1996, pp. 14–27.

[13] W. Emde and D. Wettschereck, Relational instance-based learning, in: L. Saitta, ed., Proceedings 13th International Conference on Machine Learning, Morgan Kaufmann, San Mateo, CA, 1996, pp. 122–130.

[14] U.M. Fayyad and K.B. Irani, Multi-interval discretization of continuous-valued attributes for classification learning, in: Proceedings 13th International Joint Conference on Artificial Intelligence (IJCAI-93), Chambery, France, Morgan Kaufmann, San Mateo, CA, 1993, pp. 1022–1027.

[15] Y. Freund and R. Shapire, Experiments with a new boosting algorithm, in: L. Saitta (Ed.), Proceedings 13th International Conference on Machine Learning, Morgan Kaufmann, San Mateo, CA, 1996, pp. 148–156.

[16] D. Kazakov, L. Popelinsky and O. Stepankova, ILP datasets page [`http://www.gmd.de/ml-archive/datasets/ilp-res.html`], 1996.

[17] S. Kramer, Structural regression trees, in: Proceedings 13th National Conference on Artificial Intelligence (AAAI-96), Portland, OR, 1996.

[18] J.W. Lloyd, Foundations of Logic Programming, 2nd ed., Springer, Berlin, 1987.

[19] M. Manago, Knowledge intensive induction, in: A.M. Segre (Ed.), Proceedings 6th International Workshop on Machine Learning, Morgan Kaufmann, San Mateo, CA, 1989, pp. 151–155.

[20] C.J. Merz and P.M. Murphy, UCI repository of machine learning databases [`http://www.ics.uci.edu/~mlearn/mlrepository.html`], University of California, Department of Information and Computer Science, Irvine, CA, 1996.

[21] R.S. Michalski, I. Mozetič, J. Hong and N. Lavrač, The multi-purpose incremental learning system AQ15 and its testing application to three medical domains, in: Proceedings 5th National Conference on Artificial Intelligence (AAAI-86), Philadelphia, PA, Morgan Kaufmann, San Mateo, CA, 1986.

[22] R.J. Mooney and M.E. Califf, Induction of first-order decision lists: results on learning the past tense of English verbs, J. Artif. Intell. Res. (1995) 1–23.

[23] S. Muggleton, Inverse entailment and progol, New Generation Computing 13 (1995).

[24] S. Muggleton and L. De Raedt, Inductive logic programming: theory and methods, J. Logic Programming 19–20 (1994) 629–679.

[25] G. Plotkin, A note on inductive generalization, in: B. Meltzer and D. Michie (Eds.), Machine Intelligence, Vol. 5, Edinburgh University Press, 1970, pp. 153–163.

[26] J.R. Quinlan, Learning first-order definitions of functions, J. Artif. Intell. Res. 5 (1996) 139–161.

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

[28] J.R. Quinlan, Induction of decision trees, Machine Learning 1 (1986) 81–106.

[29] J.R. Quinlan, FOIL: a midterm report, in: P. Brazdil (Ed.), Proceedings 6th European Conference on Machine Learning, Lecture Notes in Artif. Intell., Springer, Berlin, 1993.

[30] J.R. Quinlan, Bagging, boosting, and C4.5, in: Proceedings 13th National Conference on Artificial Intelligence (AAAI-96), Portland, OR, 1996, pp. 725–730.

[31] A. Srinivasan, S.H. Muggleton and R.D. King, Comparing the use of background knowledge by inductive logic programming systems, in: L. De Raedt (Ed.), Proceedings 5th International Workshop on Inductive Logic Programming, 1995.

[32] A. Srinivasan, S.H. Muggleton, M.J.E. Sternberg and R.D. King, Theories for mutagenicity: a study in first-order and feature-based induction, Artificial Intelligence 85 (1996) 277–299.

[33] P.E. Utgoff, Incremental induction of decision trees, Machine Learning 4 (2) (1989) 161–186.

[34] W. Van Laer, L. De Raedt and S. Džeroski, On multi-class problems and discretization in inductive logic programming, in: Z.W. Ras and A. Skowron (Eds.), Proceedings 10th International Symposium on Methodologies for Intelligent Systems (ISMIS97), Lecture Notes in Artif. Intell., Vol. 1325, Springer, Berlin, 1997, pp. 277–286.

[35] L. Watanabe and L. Rendell, Learning structural decision trees from examples, in: Proceedings 12th International Joint Conference on Artificial Intelligence (IJCAI-91), Sydney, Australia, 1991, pp. 770–776.