Integrating Decision Tree Learning into Inductive Databases

Élisa Fromont, Hendrik Blockeel, and Jan Struyf

Department of Computer Science, Katholieke Universiteit Leuven,
Celestijnenlaan 200A, 3001 Leuven, Belgium
{elisa.fromont,hendrik.blockeel,jan.struyf}@cs.kuleuven.be

Abstract. In inductive databases, there is no conceptual difference between data and the models describing the data: both can be stored and queried using some query language. The approach that adheres most strictly to this philosophy is probably the one proposed by Calders et al. (2006): in this approach, models are stored in relational tables and queried using standard SQL. The approach has been described in detail for association rule discovery. In this work, we study how decision tree induction can be integrated in this approach. We propose a representation format for decision trees similar to the format proposed earlier for association rules, and queryable using standard SQL; and we present a prototype system in which part of the needed functionality is implemented. In particular, we have developed an exhaustive tree learning algorithm able to answer a wide range of constrained queries.

1 Introduction

An inductive database (IDB) [11] is a database that contains not only data, but also generalizations (patterns and models) valid in the data. In an IDB, ordinary queries can be used to access and manipulate data, while inductive queries can be used to generate (mine), manipulate, and apply patterns.

Two approaches have been studied to represent and query patterns and models in IDBs. First, depending on the models that will be stored, a special-purpose storage and query language can be created. In this context, several researchers have proposed extensions to the popular relational query language SQL. For example, Meo et al. [15] and Imielinski & Virmani [12] present extensions to SQL specifically designed for mining association rules. Kramer et al. [14] and Han et al. [10] extend this approach to other models such as classification rules, but they do not give any details about how to actually store those models in the IDB. ATLaS [24] defines new table functions and aggregates in SQL, such that writing data mining algorithms (e.g., decision tree learners) in SQL becomes convenient. This approach has the closure property due to the use of SQL in the whole data mining process, but requires a deep understanding of SQL and data mining algorithm implementation to be used in practice. De Raedt [6] proposes a query language based on first order logic, which is especially suited for relational

data. Michalski & Kaufman [16] propose to use a knowledge generation meta-language (KGL) that combines conventional database operators with operators for conducting a large number of inductive inference tasks and operators for managing knowledge.

The second approach consists of storing the patterns and models in standard relational database tables, which are provided by any relational database management system (RDBMS), and using the standard SQL language, to represent, store, and query the generalizations made on the data. This approach is being investigated by members of the ADReM group in Antwerp[1] for frequent itemset and association rule mining [4]; we will refer to it in the rest of the paper as "the ADReM approach". This approach has a number of advantages over other approaches with respect to extensibility and flexibility. In this paper, we investigate how the ADReM approach can be used for learning global models, such as decision trees, and to which extent its advantages carry over to this new setting. In particular, while *support* and *confidence* constraints are obvious when querying association rules, it is much less clear which constraints are useful for decision trees. We propose some interesting constraints for decision trees and show how they can be enforced by means of two different decision tree learning algorithms.

Section 2 presents the basic ideas behind the ADReM approach and shows how they apply in the context of association rule discovery. Section 3 extends the ADReM approach to decision tree learning. We first discuss how a standard greedy decision tree learner can be used in this context (Section 4.1). Because this approach has a number of disadvantages, we propose a new decision tree learning algorithm that employs exhaustive search (Section 4.2). The latter is more suitable for answering certain inductive queries for decision trees. Section 5 presents the perspectives of this work and Section 6 states the main conclusions.

2   The ADReM Approach to Association Rule Mining

The basic idea behind the ADReM approach is that models are stored in a relational database in the same way that other information is stored: as a collection of tuples in relational tables. While applicable to a wide range of models, this idea has primarily been studied in the context of association rules [1]. Below, we briefly review the proposed representation for association rules, the conceptual view on association rule mining that it leads to, and some implementation issues. More information can be found in [4].

2.1   The Conceptual View

Consider a set of transactions D. The set is often represented as a table with one row per transaction and one Boolean attribute per item, but conceptually it can also be represented as a binary relational table with, for each transaction, a set

---

[1] {toon.calders,bart.goethals,adriana.prado}@ua.ac.be

Sets

isid	item
i1	p3
i1	p5
i1	p6
i2	red
i3	p3
i3	p5
i3	p6
i3	red
...	...

Supports

isid	support
i1	10
i2	20
i3	8
...	...

Rules

rid	isida	isidc	isid	conf
r1	i1	i2	i3	0.8
r2	i4	i5	i6	0.4
...	...	...	...	...

Fig. 1. Storing association rules in a relational database. For example, the rule "p3,p5,p6 $\Rightarrow$ red" is represented by r1.

of tuples of the form (*tid,item*), where *tid* is the transaction identifier and *item* is an item name. The crucial difference between the first and second representation is that in the second, the item names are values instead of attributes. A query can therefore return an item name as part of its result set. Note that we are talking about the conceptual representation here; how the transaction table is really implemented is not important.

Itemsets can be represented in a similar way. Figure 1 shows the ADReM representation of frequent itemsets and association rules. The *Sets* table represents all itemsets. A unique identifier (*isid*) is associated to each itemset $IS(isid)$, and for each itemset of size n, there are n rows $(isid, item_j)_{1 \leq j \leq n}$ where $item_j$ is the jth item of $IS(isid)$. The *Supports* table stores the *support* of each itemset. The *Rules* table stores the computed association rules. For each rule $X \Rightarrow Y$, there is a row (*rid, isida, isidc, isid, conf*) in the IDB where *rid* is the association rule identifier, *isida* (resp. *isidc*) is the identifier of the itemset used in the antecedent (resp. consequent) of the rule, $IS(isid) = IS(isida) \cup IS(isidc)$ and *conf* is the confidence of the rule.

With this representation, finding association rules subject to certain constraints can be done easily using an SQL query. For instance, the query

```
select rid
from rules r
where r.conf >= 0.8 and
      r.isidc in (select isid from sets where item = ''red'')
```

finds all association rules with a confidence of at least 0.8 that contain the item "red" in the consequent of the rule.

2.2   The Implementation

Conceptually, the database has tables that contain all itemsets and all association rules. But in practice, obviously, the large number of patterns may make it

impractical to explicitly store them all in the database. This problem can be solved by making these tables virtual. As far as the user is concerned, these tables or *virtual mining views* contain all the tuples needed to answer the user query. In reality, each time such a table is queried, a efficient data mining algorithm is triggered by the DBMS to populate the views with the tuples that the DBMS needs to answer the query.

More specifically, the procedure works as follows: given a query, an execution plan is created; on the highest level this is a regular relational algebra tree with tables as leaves. Standard query optimization procedures push projection and selection conditions down this tree as far as possible, thus reducing the size of intermediate results and making the overall computation more efficient. In the ADReM approach, the leaves may be the result of a data mining process, and the projection/selection conditions may be pushed further down into the data mining algorithm. Calders et al. [4] describe this optimization process in detail.

2.3   Advantages of the Approach

The ADReM approach has several advantages over other approaches to inductive querying. The main point is that the data mining processes become much more transparent. From the user's point of view, tables with itemsets and rules etc. exist and can be queried like any other table. How these tables are filled (which data mining algorithm is run, with which parameter values, etc.) is transparent to the user. The user does not need to know about the many different implementations that exist and when to use which implementation, nor does she need to familiarize herself with new special-purpose query languages. The whole approach is also much more declarative: the user specifies conditions on the models that should result from the mining process, not on the mining process itself.

In the light of these advantages, it seems useful to try a similar approach for other data mining tasks as well. In this paper, we focus on decision tree induction.

3   Integration of Decision Tree Learning

A decision tree classifies instances by sorting them down the tree from the root to a leaf node that provides the classification of the instance [17] (Figure 2). Each internal node of the tree specifies a test on the value of one of the attributes of the instance, and each branch descending from the node corresponds to one of the possible outcomes of the test. In this paper, for simplicity reasons, we focus on decision trees with Boolean attributes. Each attribute can then be seen as an item in a transaction table and transactions are instances to classify.

In this section, we discuss the motivations for integrating decision trees into an IDB and propose a representation for decision trees that will enable the user to pose queries supporting several types of useful constraints.

3.1   Motivation

To see the motivation behind using the ADReM approach for decision tree learning, consider the current practice in decision tree induction. Given a data set, one runs a decision tree learning algorithm, e.g., C4.5 [20], and obtains one particular decision tree. It is difficult to characterize this decision tree in any other way than by describing the algorithm. The tree is generally not the most accurate tree on the training set, nor the smallest one, nor the one most likely to generalize well according to some criterion; all one can say is that the learning algorithm tends to produce relatively small trees with relatively high accuracy. The algorithm usually has a number of parameters, the meaning of which can be described quite precisely in terms of how the algorithm works, but not in terms of the results it yields. To summarize, it is difficult to describe exactly what conditions the output of a decision tree learner fulfills without referring to the algorithm.

This situation differs from the discovery of association rules, where the user imposes constraints on the rules to be found (typically constraints on the confidence and support) and the algorithm yields the set of all rules satisfying these conditions. A precise mathematical description of the result set is easy to give, whereas a similar mathematical description of the tree returned by a decision tree learner is quite difficult to give.

Are people using decision tree learners interested in having a precise specification of the properties of the tree they obtain? Aren't they only interested in finding a tree that generalizes the training instances well, without being interested in exactly how this is defined? This may often be the case, but certainly not always. Many special versions of decision tree learners have been developed: some use a cost function on attributes and try to find trees that combine a high accuracy with a low cost of the attributes they contain [22]; some take different misclassification costs into account while building the tree [7]; some do not aim for the highest accuracy but for balanced precision-recall [25]; etc. The fact that researchers have developed such learning algorithms shows that users sometimes do have more specific desiderata than just high predictive accuracy.

By integrating decision tree learning into inductive databases, we hope to arrive at an approach for decision tree learning that is just as precise as association rule learning: the user specifies what kind of trees she wants, and the system returns such trees.

Here are some queries the user might be interested in:

1. find $\{T | size(T) < 8 \wedge acc(T) > 0.8 \wedge cost(T) < 70\}$
2. find one element in $\{T | size(t) < 8 \wedge acc(t) > 0.8 \wedge cost(t) < 70\}$
3. find $\{T | size(T) < 8 \wedge (\forall T' | size(T') < 8 \Rightarrow acc(T') < acc(T))\}$
4. find $\{T | T = (t(X, t(Y, l(+), l(-)), t(Z, l(C_1), l(C_2))),$
   $X \in [A, B, C], Y \in [D, E], acc(T) > 0.8\}$

In the first query, the user asks for all decision trees T with a *size* smaller than 8 nodes, a *global accuracy* greater than 0.8 and a *cost* less than 70 (assuming that each item has a given cost). To describe the tree of interest, other criteria

such as the number of misclassified examples (*error*), the *accuracy* computed for a particular class, the *precision*, the *recall*, or the area under the ROC curve (AUC) might also be interesting.

Since the user is interested in all the trees that fulfill the given criteria, the query cannot be answered by triggering a standard *greedy* decision tree learning algorithm. Instead, a decision tree learner is needed that can search the space of all possible trees *exhaustively* for trees that satisfy the constraints. The number of such trees might be huge and as a result executing the query might not be tractable. Note that without constraints, the number of decision trees that can be constructed from a database with d attributes and a possible values for each attribute is lower-bounded by $\prod_{i=0}^{d-1}(d-1)^{a^i}$. As in the association rule case presented in Section 2, we assume that the queries are sufficiently constrained so that a realistic number of trees is returned and stored. Which kind of constraints are adequate here is still an open question.

In the second query, the user asks for one tree that fulfills some criteria. This tree can normally be computed by a regular *greedy* tree learner. Note, however, that for greedy learners, given that at least one tree satisfying the constraints exists in the search space, there is usually no guarantee that the learner will find one.

With the third query, the user looks for the set of trees of size smaller than 8 nodes with maximal accuracy. Again, this means that the search space of trees smaller than 8 nodes must be searched exhaustively to ensure that the accuracy of the returned trees are maximal.

In the last query, the user provides *syntactic constraints* on the shape of the tree and provides some attributes that must appear in it.

More generally, we are interested in queries of the form $\{t \in T | C(t)\}$ where $C(t)$ is "any conjunction of constraints on a particular tree". This does not include the whole range of possible queries. In particular, queries that specify constraints on a *set of trees* such as "find the k best trees as different as possible that fulfill some constraints" [13] are out of the scope of this paper.

3.2   Representing Trees in a Relational Database

The virtual mining view that holds the predictive models should be precise enough to enable the user to ask SQL queries as easily as possible without having to design new keywords for the SQL language. We use the same data representation (transactions) as in Section 2, and assume that each transaction contains either the '+' item or the '-' item, which indicate the class of the transaction. We further assume that all the data-dependent measures (such as accuracy) are referring to these transactions.

Note that due to the well-known correspondence between trees and rule sets, a tree can be represented as a set of association rules: each leaf corresponds to one association rule, with as antecedent the conditions on the path from the root to that leaf and as consequent the class label of that leaf. However, while the representation with one rule per leaf is interesting for prediction purposes, the structure of the tree gives more information: e.g., the higher an attribute occurs

in the tree, the more informative it is for the prediction. Such information is lost if we represent a tree as a set of association rules. We therefore choose a slightly different representation.

The decision trees generated from the data D can be stored in the same database as D and the association rules computed from D, by using the following schema (Figure 2):

1. The *Tree_sets* table is inspired by the *Sets* table used for representing association rules. We choose to represent a node of a tree by the itemset that characterizes the examples that belong to the node. For instance, if the itemset is $A\overline{B}$, the examples in this node must fulfill the criteria $A=true$ and $B=false$. In association rules, only the presence of certain items is indicated: there is no condition that specifies the absence of an item. To cope with association rules derived from trees such as the one corresponding to leaf $L2$ of the tree in Figure 2 ($A\overline{B} \Rightarrow -$), we add a *sign* attribute to the *Tree_sets* table indicating whether the presence (1) or the absence (0) of the item is required.

As in the *Sets* table, a unique identifier ($isid$) is associated to each itemset and, for each itemset of size n, there are n rows $(isid, item_j, sign_j)_{1 \leq j \leq n}$
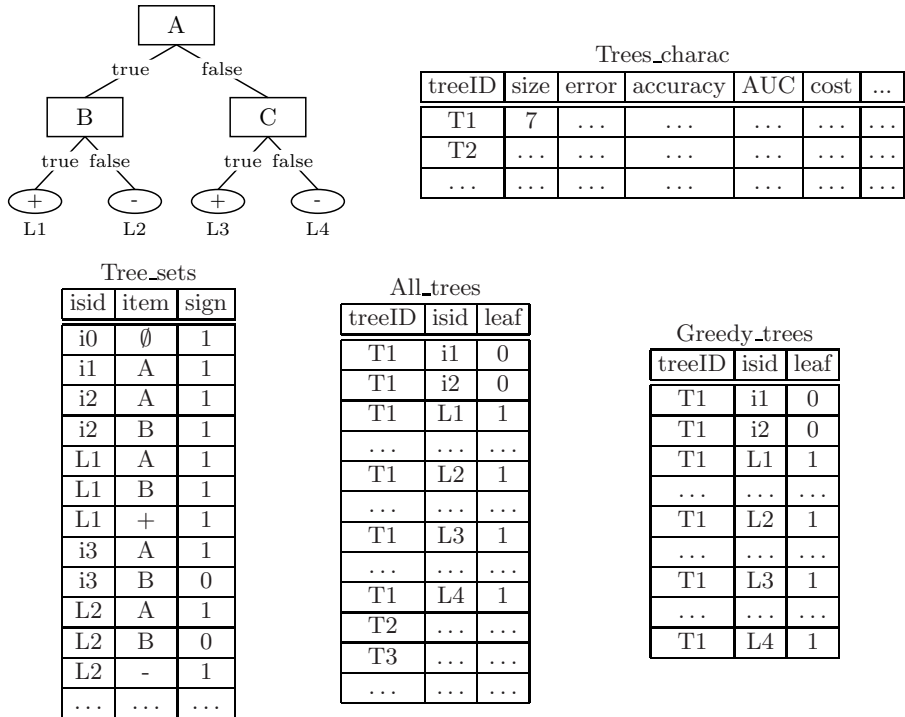

Fig. 2. Storing decision trees in a relational database

where $item_j$ is the jth item of the itemset identified by $isid$ and $sign_j$ is its sign. $i0$ stands for the empty itemset.

2. The *All_trees* and *Greedy_trees* tables give a precise description of each tree in terms of the itemsets from *Tree_sets*. Each tree has a unique identifier *treeID* and each itemset corresponding to one of its nodes is associated with this *treeID*. A Boolean attribute *leaf* distinguishes the internal nodes of the tree ($leaf = 0$) from its leaves ($leaf = 1$). The nodes at level k of the tree, are defined in terms of k-itemsets. The *All_trees* table holds all possible trees, whereas the *Greedy_trees* table can be queried for an implementation-dependent subset of all trees (the trees that may be found by a greedy learner under certain conditions and constraints).

3. The *Trees_charac* table lists all the tree characteristics where the user might be interested in. The table contains for each tree identified by *treeID*, a row with all characteristics *(size, error, accuracy, cost, AUC)* computed for the tree (see Section 3.1).

The *All_trees* and *Greedy_trees* tables correspond, as discussed before, to the *ideal* approach and the current and *practically feasible* approach to compute decision trees. Both tables are thus relevant in our framework.

Note that the semantics of the SELECT operator applied to the *Greedy_trees* table is different from the standard relational algebra definition of this operator. For example, querying the *Greedy_trees* table for a tree will yield a particular decision tree as result. If we subsequently add a certain constraint to the query, then the system will return a different tree (a tree satisfying this constraint). This is different from what happens when querying a regular database table because the answer set of the query with the additional constraint is not a subset of the answer to the original query. In our framework, the SELECT operator applied to the *Greedy_trees* table only outputs a single tree that fulfills the user's constraints.

3.3   Querying Decision Trees Using Virtual Views

The database schema of Section 3.2 is sufficient to be able to answer interesting queries, provided that the data mining algorithms connected to the database compute the different characteristics of the trees that hold in the IDB. We continue by presenting a number of example queries.

```
SELECT trees_charac.* FROM trees_charac, all_trees
WHERE trees_charac.treeID = all_trees.treeID AND
 accuracy >= 0.8 and size <= 8;
```

This query selects the characteristics of all trees that can be computed from the database, that have an accuracy of at least 0.80, and that contain at most 8 nodes.

```
SELECT treeID FROM trees_charac, greedy_trees
WHERE trees_charac.treeID = greedy_trees.treeID
and trees_charac.error < 10;
```

This query selects a tree constructed with a greedy algorithm that misclassifies fewer than 10 instances.

```
SELECT trees_charac.* FROM trees_charac, all_trees
WHERE trees_charac.treeID = all_trees.treeID
AND accuracy = (select max(accuracy) from trees_charac);
```

This query selects the characteristics of the most accurate tree(s).

```
SELECT treeID FROM greedy_trees, tree_sets
WHERE greedy_trees.isid = tree_sets.isid
AND (tree_sets.isid
     IN (select isid from tree_sets where item = ''A''));
```

This query selects a tree constructed with a greedy algorithm that contains the item "A".

3.4   User Defined Virtual Views

The framework is flexible enough to allow queries with constraints on metrics that were not included in the virtual view from the beginning. The user can create his own virtual mining view using information such as the support of the itemsets. We illustrate this by providing definitions for "accuracy" and "size".

The accuracy of a specific leaf in the tree can be computed from the support of the itemsets that belong to the leaf [19] as follows (for the tree in Figure 2):

$$acc(L_1) = \frac{support(+AB)}{support(AB)}, \; acc(L_2) = \frac{support(-A\overline{B})}{support(A\overline{B})}, \; \ldots$$

The global accuracy of the tree is the weighted mean of the accuracies of its leaves. This can be computed without any information on the actual structure of the tree as follows:

$$acc(T) = acc(L1) \cdot \frac{support(AB)}{support(\emptyset)} + acc(L2) \cdot \frac{support(A\overline{B})}{support(\emptyset)} + \ldots$$

$$= \frac{support(+AB) + support(-A\overline{B}) + \ldots}{support(\emptyset)}.$$

Itemsets that include a "negative" item usually do not have their support computed. In this case, formulas based on the inclusion-exclusion principle, such as:

$$support(A\overline{B}-) = support(A-) - support(AB-)$$

can be used to compute the support of all itemsets from the support of the "positive" itemsets [3].

These formulas can be translated into the SQL language to compute all the characteristics in the *Tree_charac* table. As in Section 2, we assume that we have a *Supports* table that contains the support of all itemsets.

```
acc(T1)= SELECT SUM(Supports.support) /
                (SELECT Supports.support
                 FROM Supports WHERE Supports.isid = ''I0'')
                as accuracy
         FROM Supports, all_trees
         WHERE Supports.isid = all_trees.isid
         AND all_trees.treeID = T1
         GROUP BY all_trees.treeID

size(T1) = SELECT COUNT(*) FROM all_trees
           WHERE all_trees.treeID = T1
```

4   Implementation

The ADReM group connected an Apriori-like algorithm for association rule mining to a standard Oracle database. The resulting system can answer inductive queries for association rules including constraints on the support of the itemsets and the confidence of the rules, and constraints requiring the presence or absence of some item in the resulting rules. We extend this system by interfacing it to a decision tree learner named CLUS[2]. CLUS is a predictive clustering tree [2] learner that uses a standard greedy recursive top-down induction algorithm to construct decision trees. CLUS can be used to answer queries with regard to the *Greedy_trees* table. To support queries on the *All_trees* table, we propose a new algorithm CLUS-EX that performs an exhaustive search for all decision trees satisfying a set of constraints. We first discuss in more detail queries on the *Greedy_trees* table (Section 4.1) and then present CLUS-EX (Section 4.2).

4.1   Greedy Tree Learning

For this task we use the standard implementation of CLUS, which is a greedy recursive top-down induction algorithm similar to C4.5 [20]. First, a large tree is built based on the training data and subsequently this tree is pruned such that the constraints in the query are satisfied. Following the precursor work by Garofalakis et al. [8], a number of constraints were implemented in CLUS [21]. It currently supports constraints on the size of the tree (i.e., an upper-bound on the number of nodes), on the error of the tree, and on the syntax of the tree. The error measure used for classification tree learning is the proportion of misclassified examples (i.e., 1.0-*accuracy*). The syntactic constraints allow the user to introduce expert knowledge in the tree. This expert knowledge takes the form of a partial tree (including the root) that must appear in the resulting tree. Essentially, this subtree specifies the important attributes in the domain. Other constraints discussed in Section 3.1 still have to be implemented in the system. Currently, queries such as the following can be used:

---

[2] http://www.cs.kuleuven.be/~dtai/clus/

```
SQL> select * from trees_charac c, greedy_trees g
where c.tree_id = g.tree_id and c.err <= 0.2 and c.sz= 9;


TREE_ID SZ ERROR ACCURACY
------- -- ----- --------
     0  9  0.02     0.98  1 rows selected.


SQL> select * from trees_charac c, greedy_trees g
where c.tree_id = g.tree_id and c.err <= 0.2 and c.sz <= 8;


TREE_ID SZ ERROR ACCURACY
------- -- ----- --------
     1  7 0.027    0.973  1 rows selected.


SQL> select * from trees_charac c, greedy_trees g
where c.tree_id = g.tree_id and c.sz< 4;


TREE_ID SZ ERROR ACCURACY
------- -- ----- --------
     2  3 0.333    0.667  1 rows selected.
```

The trees computed for the different queries can be stored in a "log" table that can be queried just as easily. After the session above, this table would contain:

```
TREE_ID    SZ    ERROR    ACCURACY
-------  -----  -------  ----------
     0     9     0.02        0.98
     1     7     0.027       0.973
     2     3     0.333       0.667
```

4.2 Exhaustive Tree Learning

This section proposes the exhaustive tree learner Clus-EX. Clus-EX searches the space of all possible trees of at most MaxSize (a parameter) nodes in a depth-first fashion. Basically, a queue of trees is kept; a search step consists of taking the first tree of the queue, computing refinements for it, and adding those refinements to the front of the queue. A refinement consists of splitting one leaf of the tree according to one of the available attributes. Generating all possible refinements in this way is not optimal because the same tree may be generated multiple times. To avoid identical trees from being generated, it is sufficient to restrict the refinements as follows. Clus-EX only splits leaves that are below or to the right of the last node on the right-most path of internal nodes (or, more formally: the ones that come after the last internal node of the current tree when it is written in pre-order notation). In the example in Figure 3.a, only the dark gray leaves will be refined. The completeness and optimality of this method follow from a result by Chi et al. [5].
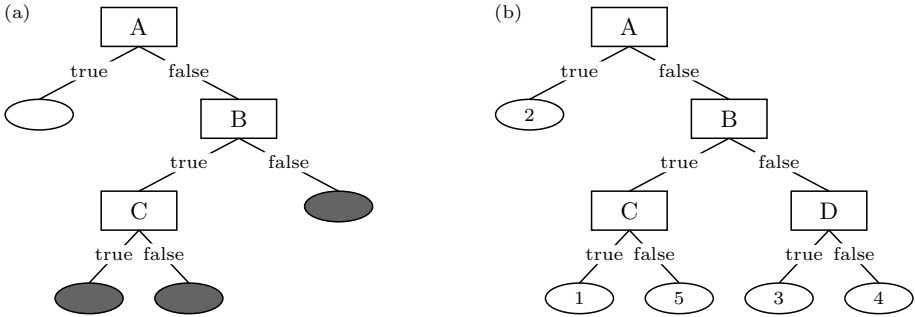
Fig. 3. (a) Example tree being refined by CLUS-EX. The right-most path of internal nodes is A, B, C. Only the leaves below or to the right of C are refined (indicated in dark gray). (b) An illustration of exploiting constraints on both the size and the error of the tree.

As discussed in Section 3.1, finding all possible trees that can be constructed on a data set is intractable in the general case. However, if the user specifies sufficiently restrictive constraints, and these constraints can be used for pruning, then the search becomes practically feasible. (This is similar to itemset mining, where a sufficiently high support threshold must be set so that the search becomes tractable.) In our case, including a size constraint in the inductive query is crucial to prune the search, that is, the search can stop as soon as the given size limit is reached. But a combination of size and error constraints can also be exploited for pruning. Assume that the constraints state that there can be at most E_{max} misclassified examples and that the size of the tree (internal nodes plus leaves) can be at most S. Take a tree of size S'. Splitting a leaf increases the size of the tree by at least 2, so at most $m = (S - S')/2$ (integer division) leaves of the current tree can be split before the maximum size is reached. Observe that the greatest error reduction occurs when the leaves with the largest number of errors (i.e., number of examples not belonging to the majority class in the leaf) are split, and all these splits yield pure leaves. Assume that the current tree has E_0 errors in leaves that cannot be split anymore by the optimal refinement operator, and k leaves L_i that still can be split (the gray leaves in Figure 3.a). Assume further that the L_i are sorted such that $e_1 \geq e_2 \geq \ldots \geq e_k$, with e_i the number of errors of L_i. Then any valid extension of this tree has a total error of at least $E_0 + \sum_{i=m+1}^{k} e_i$. If this sum is greater than E_{max}, then we can safely prune the search at this point.

Consider the example in Figure 3.b, and assume that the five leaves have (from left to right) $2, 1, 5, 3$, and 4 errors. Let MaxSize $= 11$. The current size is 9, so we can do at most one more split. Given the refinement strategy described above, we know that only the two right-most leaves can be split in the current situation (the two leaves below node D). Therefore, $E_0 = 2 + 1 + 5 = 8$ errors, and the best possible split occurs when the leaf with 4 errors is replaced by two pure leaves. The lower bound on the error is therefore $3 + 8 = 11$.

Table 1. The performance of CLUS-EX on UCI data sets. We consider for each data set different values for the MaxSize and MaxError constraints. For each constraint pair, the size of the search space (number of trees searched) without and with error based pruning is reported together with the reduction factor due to pruning (Red.). The last column is the number of trees that satisfy the constraints.

Data set	MaxSize	MaxError	Search space (# trees searched)		Red.	# Result
			No pruning	Pruning		
soybean	7	0.3	177977	86821	2.0	0
soybean	9	0.3	8143901	4053299	2.0	0
soybean	7	0.6	177977	125055	1.4	848
zoo	7	0.2	13776	9218	1.5	214
zoo	7	0.3	13776	11908	1.2	2342
zoo	9	0.3	345388	276190	1.3	95299
zoo	11	0.2	7871768	4296549	1.8	708026
zoo	11	0.1	7871768	1637934	4.8	16636
audiology	7	0.3	415704	380739	1.1	0
audiology	7	0.5	415704	406290	1.0	2326

Table 1 presents the results of CLUS-EX on different symbolic UCI [18] data sets. The third and the fourth columns show the number of nodes evaluated during the search, i.e, they give an idea of the size of the search space and of the efficiency of the pruning method. The table shows that when the maximum *error* (resp. minimum *accuracy*) is sufficiently low (resp. high) compared to the size constraint, the combination of the constraints can be used to efficiently prune the search space. The table shows that restrictive constraints are crucial to restrict the search space and the number of resulting trees. The soybean and the audiology data sets contain many classes, therefore, the combination of strict error and size constraints can easily lead to an empty result set.

Currently, queries such as the following can be used with the prototype:

```
SQL> select c.tree-id,sz,accuracy,err from trees_charac c, all_trees a
where c.tree_id = a.tree_id and sz <= 7 and accuracy > 0.8');


 TREE_ID          SZ         ERR    ACCURACY
---------- ---------- ---------- ----------
        0           7       0.125       0.875
        1           7       0.125       0.875
        2           7       0.125       0.875
        3           7       0.125       0.875
        4           7       0.125       0.875
        5           7       0.125       0.875
        6           7       0.125       0.875
        7           7       0.125       0.875
        8           5           0           1
        9           5           0           1
```

```
SQL> select * from all_trees a, trees_sets t
where a.set_id = t.set_id and tree_id = 0;

   TREE_ID      SET_ID       NODE      SET_ID ITEM        SIGN
---------- ---------- ---------- ---------- -------- ----
         0          0          1          0 null           1
         0          1          1          1 null           1
         0          1          1          1 B = F          1
         0          2          1          2 null           1
         0          2          1          2 B = F          1
         0          2          1          2 A = F          1
         0          3          0          3 null           1
      ....       ....       ....       .... ....         ...
```

5   Perspectives

There are many open problems related to the proposed approach. For instance, for efficiency reasons, the system should be able to look at the "log" table that contains the previously computed trees to check if the answer of the current query has not already been computed, before triggering a data mining algorithm. If the user asks for all trees of size smaller than 8, and later for all trees of size smaller than 6, the results computed from the first query should be reusable for the second query. The "log" table should also contain the previous queries together with the computed trees, which raises the question of how to store the queries themselves in the database. This entire problem, called *interactive mining* because it refers to the reutilisation of queries posed within the same working session, has been investigated for association rules [9], but not for decision tree learning.

If the database is modified between two queries, then it might be interesting to reuse the previously computed predictive models to more efficiently compute new predictive models for the modified database. This problem known has *incremental learning* has already been studied for decision trees [23] when a new example is added to the database.

These functionalities have to be integrated into the prototype along with the extension of the framework to multi-valued attributes.

Because predictive models ultimately aim at predicting the class of new examples, it would be interesting to include that possibility in the IDB. This is currently non-trivial in our approach; it requires complicated queries. More generally, the limitations of our approach with respect to what can be expressed, and how difficult it is to express it, require further investigation.

Another perspective is the integration of other predictive models such as *Bayesian Networks* in the same IDB framework already designed for association rule and decision tree mining. The user might be interested in queries such as "find the Bayesian network of size 10 with maximal likelihood". Again, a structure to store Bayesian networks has to be designed and an algorithm that can build Bayesian networks under constraints has to be implemented.

6 Conclusion

In this paper we have studied how decision tree induction can be integrated in inductive databases following the ADReM approach. Considering only Boolean attributes, the representation of trees in a relational database is quite similar to that of association rules, with this difference that the conjunctions describing nodes may have negated literals whereas itemsets only contain positive literals. A more important difference is that a decision tree learner typically returns one tree that is "optimal" in some "not very precisely" defined way, whereas the IDB approach lends itself more easily to mining approaches that return all results fulfilling certain well-defined conditions. It is therefore useful to introduce both a *Greedy_trees* table and an *All_trees* table, where queries to *Greedy_trees* trigger the execution of a standard tree learner and queries to *All_trees* trigger the execution of an exhaustive tree learner. We have described a number of example queries that could be used, and proposed a new algorithm that is able to exhaustively search for decision trees under constraints. We presented a preliminary implementation of this algorithm, and discussed open questions and perspectives of this work.

The approach presented in this paper focused on the representation of the models, the querying mechanism and the constrained based mining of the models. We believe that the simplicity of this approach makes it easier to be included as a brick in a larger system able to support the whole KDD process, which is the ultimate aim of an inductive database.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proc. of the 20th Int. Conf. on Very Large Databases VLDB, pp. 487–499 (1994)
2. Blockeel, H., De Raedt, L., Ramon, J.: Top-down induction of clustering trees. In: 15th Int'l Conf. on Machine Learning, pp. 55–63 (1998)
3. Calders, T., Goethals, B.: Mining all non-derivable frequent itemsets. In: Elomaa, T., Mannila, H., Toivonen, H. (eds.) PKDD 2002. LNCS (LNAI), vol. 2431, pp. 74–85. Springer, Heidelberg (2002)
4. Calders, T., Goethals, B., Prado, A.: Integrating pattern mining in relational databases. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) PKDD 2006. LNCS (LNAI), vol. 4213, pp. 454–461. Springer, Heidelberg (2006)
5. Chi, Y., Muntz, R.R., Nijssen, S., Kok, J.N.: Frequent subtree mining - An overview. Fundamenta Informaticae 66(1-2), 161–198 (2005)
6. De Raedt, L.: A logical database mining query language. In: Cussens, J., Frisch, A.M. (eds.) ILP 2000. LNCS (LNAI), vol. 1866, pp. 78–92. Springer, Heidelberg (2000)

7. Domingos, P.: MetaCost: A general method for making classifiers cost-sensitive. In: Knowledge Discovery and Data Mining, pp. 155–164 (1999)
8. Garofalakis, M., Hyun, D., Rastogi, R., Shim, K.: Building decision trees with constraints. Data Mining and Knowledge Discovery 7(2), 187–214 (2003)
9. Goethals, B., Van den Bussche, J.: On supporting interactive association rule mining. In: Kambayashi, Y., Mohania, M.K., Tjoa, A.M. (eds.) DaWaK 2000. LNCS, vol. 1874, pp. 307–316. Springer, Heidelberg (2000)
10. Han, J., Fu, Y., Wang, W., Koperski, K., Zaiane, O.: DMQL: A data mining query language for relational databases. In: SIGMOD 1996 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD 1996) (1996)
11. Imielinski, T., Mannila, H.: A database perspective on knowledge discovery. Comm. of the ACM 39, 58–64 (1996)
12. Imielinski, T., Virmani, A.: MSQL: A query language for database mining. Data Mining and Knowledge Discovery 3(4), 373–408 (1999)
13. Kocev, D., Džeroski, S., Struyf, J.: Similarity constraints in beam-search induction of predictive clustering trees. In: Proc. of the Conf. on Data Mining and Data Warehouses (SiKDD 2006) at the 9th Int'l Multi-conf. on Information Society (IS-2006), pp. 267–270 (2006)
14. Kramer, S., Aufschild, V., Hapfelmeier, A., Jarasch, A., Kessler, K., Reckow, S., Wicker, J., Richter, L.: Inductive databases in the relational model: The data as the bridge. In: Bonchi, F., Boulicaut, J.-F. (eds.) KDID 2005. LNCS, vol. 3933, pp. 124–138. Springer, Heidelberg (2006)
15. Meo, R., Psaila, G., Ceri, S.: An extension to SQL for mining association rules. Data Mining and Knowledge Discovery 2(2), 195–224 (1998)
16. Michalski, R.S., Kaufman, K.A.: Building knowledge scouts using KGL metalanguage. Fundamenta Informaticae 41(4), 433–447 (2000)
17. Mitchell, T.M.: Machine Learning. McGraw-Hill, New York (1997)
18. Newman, D.J., Hettich, S., Blake, C.L., Merz, C.J.: UCI repository of machine learning databases (1998)
19. Pance, P., Dzeroski, S., Blockeel, H., Loskovska, S.: Predictive data mining using itemset frequencies. In: Zbornik 8. mednarodne multikonference Informacijska druzba. Ljubljana: Institut "Jožef Stefan", pp. 224–227. Informacijska Druzba (2005)
20. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann, San Francisco (1993)
21. Struyf, J., Džeroski, S.: Constraint based induction of multi-objective regression trees. In: Bonchi, F., Boulicaut, J.-F. (eds.) KDID 2005. LNCS, vol. 3933, pp. 222–233. Springer, Heidelberg (2006)
22. Turney, P.: Cost-sensitive classification: Empirical evaluation of a hybrid genetic decision tree induction algorithm. Journal of Artificial Intelligence Research 2, 369–409 (1995)
23. Utgoff, P.E.: Incremental induction of decision trees. Machine Learning 4, 161–186 (1989)
24. Wang, H., Zaniolo, C.: ATLaS: A native extension of SQL for data mining. In: SIAM Int'l Conf. Data Mining, pp. 130–144 (2003)
25. Xiaobing, W.: Knowledge representation and inductive learning with XML. In: Proc. of the IEEE/WIC/ACM Int'l Conf. on Web Intelligence (WI 2004), pp. 491–494. IEEE Computer Society Press, Los Alamitos (2004)