HAL
open science

An inductive database prototype based on virtual mining views

Hendrik Blockeel, Toon Calders, Elisa Fromont, Bart Goethals, Adriana Prado, Céline Robardet

To cite this version:

Hendrik Blockeel, Toon Calders, Elisa Fromont, Bart Goethals, Adriana Prado, et al.. An inductive database prototype based on virtual mining views. SIGKDD international conference on Knowledge discovery and data mining, Aug 2008, Las Vegas, United States. pp.1061-1064, 10.1145/1401890.1402019 . hal-00372017

HAL Id: hal-00372017

https://hal.science/hal-00372017v1

Submitted on 31 Mar 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An Inductive Database Prototype Based on Virtual Mining Views

Hendrik Blockeel
K.U. Leuven
Leuven, Belgium

Toon Calders
T.U. Eindhoven
Eindhoven, The Netherlands

Elisa Fromont
K.U. Leuven
Leuven, Belgium

Bart Goethals
Universiteit Antwerpen
Antwerp, Belgium

Adriana Prado
Universiteit Antwerpen
Antwerp, Belgium

Céline Robardet
LIRIS, UMR 5208 INSA-LYON
Lyon, France

ABSTRACT

We present a prototype of an inductive database. Our system enables the user to query not only the data stored in the database but also generalizations (e.g. rules or trees) over these data through the use of virtual mining views. The mining views are relational tables that virtually contain the complete output of data mining algorithms executed over a given dataset. The prototype implemented into PostgreSQL currently integrates frequent itemset, association rule and decision tree mining. We illustrate the interactive and iterative capabilities of our system with a description of a complete data mining scenario.

Categories and Subject Descriptors: H.2.4 [Database Management]: System.

General Terms: Algorithms, Experimentation.

Keywords: Data Mining, Inductive Databases.

1. INTRODUCTION

Data mining is not a one-shot activity, but rather an iterative and interactive process. During the whole discovery process, typically, many different data mining tasks are performed, their results are combined, and possibly used as input for other data mining tasks. To support this knowledge discovery process, there is a need for integrating data mining with data storage and management. The concept of inductive databases (IDBs) has been proposed as a means of achieving such integration [7]. In [3, 4, 5, 6], we describe how such an inductive database can be designed in practice, by using *virtual mining views*.

In an IDB, one can not only query the data stored in the database, but also the patterns that are implicitly present in these data. One of the main advantages of our system is the flexibility of ad-hoc querying, that is, the user can specify new types of constraints and query the patterns and models in combination with the data itself and so forth. In this paper, we illustrate this feature with a data mining scenario in which we first learn a classifier over a given dataset and, afterwards, we look for correct association rules, which describes the misclassified examples w.r.t this classifier. Notice that the functionality of an inductive database goes far beyond that of data mining suites such as, Weka [10] and Yale [8]. These systems have the advantage of imposing one uniform data format for a group of algorithms. On the other hand, they do not allow ad-hoc queries.

The rest of the paper is organized as follows. Section 2 addresses the idea behind the development of our prototype. Section 3 presents the virtual mining views framework. In Section 4, we describe how the system is implemented. Finally, Section 5 describes a complete database scenario, which illustrates the interactive and iterative capabilities of our system.

2. DESCRIPTION OF THE SYSTEM

The system presented in this paper builds upon our preliminary work in [3, 5, 6]. In contrast to the numerous proposals for data mining query languages, we propose to integrate data mining into database systems without extending the query language. Instead, we extend the database schema with new tables containing, for instance, association rules, decision trees, or other descriptive or predictive models. As far as the user is concerned, these tables contain all possible patterns, trees, and models that can be learned over the data. Of course, such tables would in most cases be huge. Therefore, they are in fact implemented as views, called virtual mining views.

Whenever a query is formulated, selecting for instance association rules from these tables, a run of a data mining algorithm is triggered (e.g., Apriori [1]) to compute the result of the query, in exactly the same way that normal views in databases are only computed at query time, and only to the extent necessary for answering the query.

When the user formulates his or her mining query, the parser is invoked by the DBMS, creating an equivalent relational algebra expression. At this point, the expression is processed by the *Mining Extension*, which extracts from the query the constraints that can be pushed into the data mining algorithms. The output of these algorithms is then materialized in the virtual mining views. After the material-

Copyright is held by the author/owner(s).
KDD'08, August 24–27, 2008, Las Vegas, Nevada, USA.
ACM 978-1-60558-193-4/08/08.

ization, the work-flow of the DBMS continues as usual and, as a result, the query is executed as if all patterns and models were stored in the database. Observe that this system can possibly cover every mining technique whose output can be completely stored in relational tables.

This approach also integrates constraint-based mining in a natural way. Within a query, one can impose conditions on the kind of patterns or models that one wants to find. In many cases, these constraints can be pushed into the mining process. In [5], Calders et al. present an algorithm that extracts from a query a set of constraints relevant for association rules to be pushed into the mining algorithm. In this way, not all possible patterns or models need to be generated, but only those required to evaluate the query correctly as if all possible patterns or models were stored. We have extended this constraint extraction algorithm to extract constraints from queries over decision trees. The reader can refer to [3] for more details on the algorithm.

3. THE VIRTUAL MINING VIEWS

The virtual mining views framework consists of a set of relational tables that virtually contain the complete output of data mining algorithms executed over a given dataset. Every time a dataset D is created in the system, all virtual mining views associated with D are automatically created. Figure 1 illustrates the virtual mining views for the dataset Playtennis [9]. They are the following:

- *Concepts*: Virtually contains all conjunctive concepts with conditions of the form "Attribute=value" that exist in the domain of the dataset. We represent them as tuples, using $'?'$ as the *wildcard value* and assume it does not exist in the domain of any attribute. The attribute *cid* identifies every concept.
- *Sets*: As itemsets in a relational database are conjunctions of the form "Attribute=value", they can be represented as concepts. Thus, *Sets* represents all itemsets that can be mined over the dataset along with their characteristics, such as support (supp) and size (sz).
- *Rules*: Represents all association rules that can be mined over the dataset. The attribute *rid* is the rule identifier, *cida* and *cidc* are identifiers of the concepts repesenting, respectively, the antecedent and consequent of the rule, *cid* is their union, and $conf$ is the confidence of the corresponding rule.
- *Trees_Attr*: Represents all decison trees that can be learned for one specific target attribute *Attr*. A unique identifier *treeid* is associated to every decison tree and each of the decision trees is described as a set of concepts. Each concept represents one path from the root to a leaf of the tree.
- *Treescharac_Attr*: Represents the characteristics of all decision trees that can be learned for the target attribute *Attr*. For every decision tree, there is a tuple with the decision tree identifier *treeid*, *acc* its accuracy and *sz* its size (in number of nodes).

In Section 5, we give some concrete examples of common data mining tasks and well-known constraints (such as minimum confidence and minimum accuracy) that can be expressed quite naturally with SQL queries over the mining views. For more examples, we refer the reader to [3, 4].

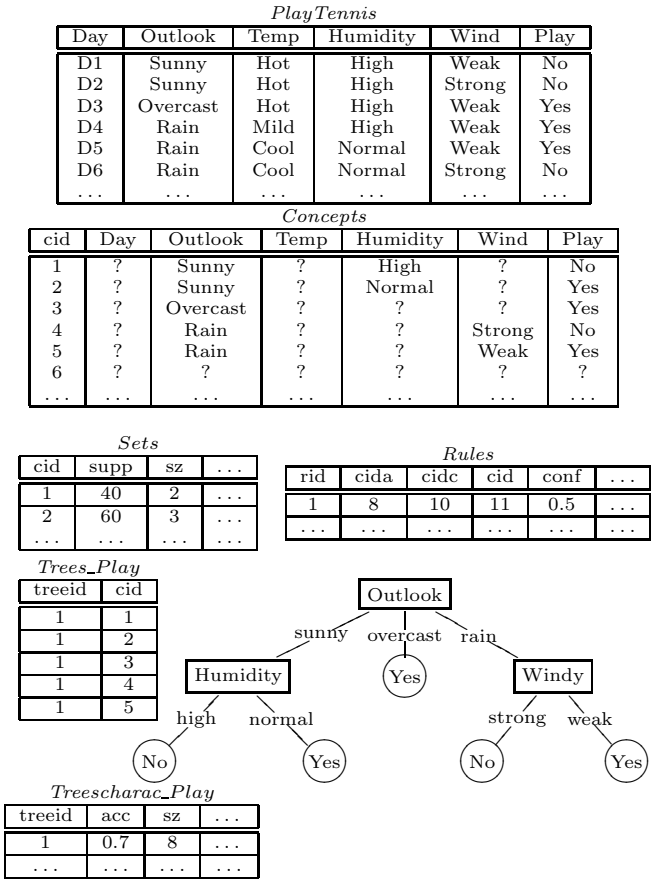

Figure 1: The Virtual Mining Views Framework.

4. IMPLEMENTATION

The system was developed into PostgreSQL[1] (written in C). We adapted the web-based administration tool PhpPgAdmin[2], in order to have a friendly interface to the system.

When the user writes a query, PostgreSQL generates a data structure representing its corresponding relational algebra expression. After this data structure is generated, our Mining Extension is called. Here, we process the relational algebra structure, extract the constraints, trigger the data mining algorithms and materialize the results in the virtual mining views. Just after the materialization, the work-flow of the DBMS continues and the query is executed as if the patterns or models were there all the time.

The system is currently linked to algorithms for frequent itemset mining, association rule discovery and exhaustive decision tree learning [6]. The constraints represented as attributes of the mining views (size, accuracy, support, confidence) can be extracted and efficiently exploited by the integrated data mining algorithms.

Experiments published in [4] showed that the execution times of the queries are rather low which support our claim that the virtual mining views provide an elegant way to incorporate data mining capacities to database systems without changing the query language.

[1] http://www.postgresql.org/
[2] http://phppgadmin.sourceforge.net/

```
create table simple_mushroom
as select cap_color, odor, gill_size,
          gill_color, spore_print_color, class
from mushroom;

alter table simple_mushroom add column id serial;
```

Figure 2: Pre-processing.

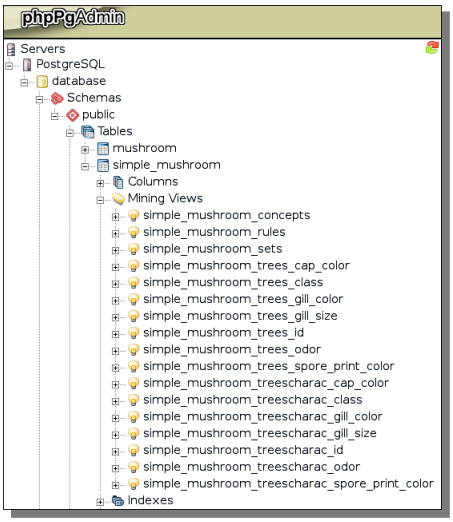

Figure 3: Mining views for table simple_mushroom.

5. AN ILLUSTRATIVE SCENARIO

In this section, we describe an extended scenario that explores the interactive and iterative capabilities of our system. The scenario consists of mining decision trees over the mushroom dataset [2] and also association rules over intermediate query results. We assume that table MUSHROOM is already stored in our system. It contains 8,124 tuples and 23 categorical attributes. The attribute *class* discriminates mushrooms from being poisonous or edible, while the other attributes describe features of the mushrooms, such as the color of the cap and odor.

Step 1: Pre-processing.

To illustrate how it is possible to preprocess data using our system, we create a new table called SIMPLE_MUSHROOM selecting only a subset of the attributes of the original table MUSHROOM.

Figure 2 shows the corresponding pre-procecessing query, followed by a query that adds an identifier to every example in the new table (it is worth noticing that the chosen attributes are known to be important for classification). The mining views automatically created for table SIMPLE_MUSHROOM can be visualized in the screenshot in Figure 3.

Step 2: Mining over decision trees.

In this step, we look for decision trees over table SIMPLE_MUSHROOM, targetting the attribute *class* and with maximum accuracy among those trees of size ≤ 5.

The query is shown in Figure 4. The subquery selects the maximum accuracy achieved by the trees with size ≤ 5 and accuracy $\geq 90\%$ (the latter constraint is added in order to

```
create table best_tree
as select t.treeid, c.*, d.acc, d.sz
    from simple_mushroom_trees_class t,
            simple_mushroom_treescharac_class d,
            simple_mushroom_concepts c
    where t.cid = c.cid
              and t.treeid = d.treeid
              and d.sz <=5
              and d.acc >= 90
              and d.acc = ( select max(acc)
                    from simple_mushroom_treescharac_class
                    where sz<=5 and acc >= 90)
    order by t.treeid, c.cid
```

Figure 4: Query selecting trees with maximum accuracy.

prune the search space of possible trees). The main query creates a table containing the trees with the characteristics mentioned above, having the pre-selected maximum accuracy. We use views *Concepts* and *Treescharac* in order to retrieve the concepts of the trees along with their characteristics. In the end, 3 trees are stored in table BEST_TREE. All of them have an accuracy of 95% and 5 nodes.

Step 3: Post-processing 1.

Having learned the trees with maximum accuracy in the previous step, we now want to explore the predictive capacity of these trees. Since all trees have the same accuracy, we choose the most balanced one, presented in Figure 5. This tree has *treeid* equal to 6.

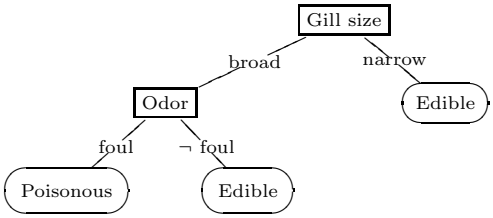

Figure 5: Decision tree (¬ foul = {spicy, pungent, none, musty, fishy, creosote, anise, almond})

Step 4: Post-processing 2.

Next, we want to store in the database the examples in SIMPLE_MUSHROOM that are misclassified w.r.t. the selected tree. To classify a new example using that tree, one simply looks up the concept that covers the new example. More generally, if we have a test set S, all predictions of the examples in S are obtained by equi-joining S with the semantic representation of the tree given in the view *Concepts*. We join S to *Concepts* using a variant of the equi-join that requires that either the values are equal or there is a wildcard value '?'.

Figure 6 shows the corresponding query. To suit our purposes, the equi-join is made between tables SIMPLE_MUSHROOM and BEST_TREE, from which the tree with $treeid = 6$ is selected. The misclassified examples are those for which the prediction is different from the real class (stated in the last line of the query).

Step 5: Post-processing 3.

In this final step, we are interested in describing the mis-

```
create table misclassified_mushroom
as select test.*
    from simple_mushroom test, best_tree tree
     where (test.cap_color= tree.cap_color
                or tree.cap_color = '?')
            and (test.odor = tree.odor
                or tree.odor = '?')
           and (test.gill_size = tree.gill_size
                or tree.gill_size = '?')
           and (test.gill_color = tree.gill_color
                or tree.gill_color = '?')
           and (test.spore_print_color
                  = tree.spore_print_color
                or tree.spore_print_color = '?')
           and tree.treeid = 6
           and test.class <> tree.class
```

Figure 6: Query selecting the misclassified mushrooms.

classified examples obtained in the former step. To this end, first we create table MUSHROOM_STATUS in which every example in table SIMPLE_MUSHROOM is labeled as well classified ($status = 1$) or misclassified ($status = 0$). The query is shown in Figure 7.

```
create table mushroom_status
as select s.*, '0' as "status"
    from simple_mushroom s
    where exists
            (select *
            from misclassified_mushroom m
            where s.id = m.id)
union
    select s.*, '1' as "status"
    from simple_mushroom s
    where not exists
            (select *
            from misclassified_mushroom m
            where s.id = m.id)
```

Figure 7: Query labeling the examples w.r.t. the selected decision tree.

Second, we mine table MUSHROOM_STATUS for correct (100% confidence) class association rules (the class is the attribute *status*), having one attribute-value pair in the antecedent, i.e. the most general rules. The query is shown in Figure 8.

```
select S.supp, S.sz, R.conf,
     C1.spore_print_color, C1.cap_color,
     C1.gill_size, C1.gill_color, C1.odor,
     '=>' as "=>", C2.status
from mushroom_status_sets S,
       mushroom_status_sets S1,
       mushroom_status_rules R,
       mushroom_status_concepts C1,
       mushroom_status_concepts C2
where R.cid = S.cid
     and R.cidc = S1.cid
     and C1.cid = R.cida
     and C2.cid = R.cidc
     and S.supp >= 15
     and R.conf >= 100
     and S.sz = 2 //total size of the rules
     and S1.sz = 1 //size of the consequent
     and C2.status <> '?'
order by C2.status
```

Figure 8: Query over association rules.

Figure 9 shows a screenshot with the rules output by the query in Figure 8. As we are interested in describing the misclassified examples, we focus on the rules having consequent "*status*=0". Clearly, these rules reveal *odor*="musty", *gill_color*="green", *cap_color*={"green","purple"}, and *spore_print_color*={"green","purple"} as discriminative features to explain the misclassifications.

Query Results

supp	sz	conf	spore_print_color	cap_color	gill_size	gill_color	odor	=>	status
36	2	100	?	?	?	?	musty	=>	0
16	2	100	?	green	?	?	?	=>	0
48	2	100	purple	?	?	?	?	=>	0
24	2	100	?	?	?	green	?	=>	0
72	2	100	green	?	?	?	?	=>	0
16	2	100	?	purple	?	?	?	=>	0
96	2	100	?	?	?	red	?	=>	1
64	2	100	?	?	?	orange	?	=>	1
1728	2	100	?	?	?	buff	?	=>	1
48	2	100	yellow	?	?	?	?	=>	1
48	2	100	orange	?	?	?	?	=>	1
48	2	100	buff	?	?	?	?	=>	1
576	2	100	?	?	?	?	spicy	=>	1
256	2	100	?	?	?	?	pungent	=>	1
2160	2	100	?	?	?	?	foul	=>	1
576	2	100	?	?	?	?	fishy	=>	1
192	2	100	?	?	?	?	creosote	=>	1

17 row(s)

Total runtime: 5,096.875 ms

Figure 9: Association rules describing the well classified and misclassified mushrooms.

This ends the data mining scenario. In conclusion, we have developed a prototype of an inductive database based on virtual mining views. The advantages of our system are threefold: Firstly, the data are mined where they are located: in the database. Secondly, the user can specify in a declarative way (SQL queries) the patterns or models in which he or she is interested. Finally, thanks to the flexibility of ad-hoc querying of our system, the output of some queries can be used as input for subsequent queries.

Acknowledgment

Hendrik Blockeel is a post-doctoral fellow from the Research Foundation – Flanders (FWO-Vlaanderen). This research was funded through K.U.Leuven GOA project 2003/8 "Inductive Knowledge bases", FWO project "Foundations for inductive databases", and the EU project "Inductive Queries for Mining Patterns and Models".

6. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994.

[2] A. Asuncion and D. Newman. UCI machine learning repository, 2007. http://www.ics.uci.edu/∼mlearn/MLRepository.html.

[3] H. Blockeel, T. Calders, E. Fromont, B. Goethals, and A. Prado. Mining views: Database views for data mining. In *CMILE workshop at PKDD*, 2007.

[4] H. Blockeel, T. Calders, E. Fromont, B. Goethals, and A. Prado. Mining views: Database views for data mining. In *IEEE ICDE*, 2008.

[5] T. Calders, B. Goethals, and A. Prado. Integrating pattern mining in relational databases. In *PKDD*, 2006.

[6] E. Fromont, H. Blockeel, and J. Struyf. Integrating decision tree learning into inductive databases. In *KDID workshop at ECML/PKDD*, 2007.

[7] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of the ACM*, 39(11):58–64, 1996.

[8] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. Yale: Rapid prototyping for complex data mining tasks. In *KDD*, pages 935–940, 2006.

[9] T. M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[10] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.