Direct Access of an ILP Algorithm to a Database Management System

Peter Brockhausen and Katharina Morik
Univ. Dortmund, Computer Science Department, LS VIII
D–44221 Dortmund
{brockh, morik}@ls8.informatik.uni-dortmund.de

Abstract

When learning from very large databases, the reduction of complexity is of highest importance. Two extremes of making knowledge discovery in databases (KDD) feasible have been put forward. One extreme is to choose a most simple hypothesis language and so to be capable of very fast learning on real-world databases. The opposite extreme is to select a small data set and be capable of learning very expressive (first-order logic) hypotheses. We have combined inductive logic programming (ILP) directly with a relational database. The tool exploits a declarative specification of the syntactic form of hypotheses. We indicate the impact of different mappings from the learner's representation to the one of the database on the complexity of learning. We demonstrate, how background knowledge can be structured and integrated into our learning framework. We conclude with discussing results from first tests.

1 Introduction

Knowledge discovery in databases (KDD) is an application challenging machine learning because it has high efficiency requirements with yet high understandability and reliability requirements. First, the learning task is to find all valid and non–redundant rules (rule learning). This learning task is more complex than the concept learning task as was shown by Uwe Kietz [11]. To make it even worse, second, the data sets for learning are very large.

Two extremes of making KDD feasible have been put forward. One extreme is to choose a most simple hypothesis language and to be capable of very fast learning on real-world databases. Fast algorithms have been developed that generalize attribute values and find dependencies between attributes. These algorithms are capable of directly accessing a database, i.e. the representation language $\mathcal{L}_{\mathcal{E}}$ is the language of the database. The Apriori and AprioriTid algorithms find *association rules* that determine subsets of correlated attribute values. Attribute values are represented such that each attribute value becomes a Boolean attribute, indicating whether the value is true or false for a certain entity [1]. Rules are formed that state: *If a set of attributes is true, then also another set of attributes is true.*

As all combinations of Boolean attributes have to be considered, the time complexity of the algorithm is exponential in the number of attributes. However, in practice the algorithm takes only 20 seconds for 100 000 tuples[1].

[1] In [1] the authors present a series of experiments with the algorithms and give a lower bound for finding an association rule.

Other fast learning algorithms exploit given hierarchies and generalize attribute values by climbing the hierarchy [15], merging tuples that become identical, and drop attributes with too many distinct values that cannot be generalized. The result are rules that characterize all tuples that have a certain value of attribute A in terms of generalized values of other attributes [5]. Similarly, the KID3 algorithm discovers dependencies between values of two attributes using hierarchies from background knowledge [19]. The result is a set of rules of the form

$$A = a' \rightarrow cond(B)$$

where a' is a generalized attribute value (i.e,. it covers a set of attribute values) of attribute A.

It is easy to see that more complex dependencies between several attributes cannot be expressed (and, hence, cannot be learned) by these fast algorithms. In particular, relations between attribute values of different attributes cannot be learned. For instance, learning a rule stating that: *If the value of $A \leq$ the value of B then the value of $C = c$* requires the capability of handling relations. Hence, these fast learning algorithms trade in expressiveness for the capability of dealing with very large data sets.

The other extreme of how to make KDD feasible is to select a small subset from the data set and learn complex rules. This option is chosen by most algorithms of inductive logic programming (ILP), which are applied to the KDD problem. The rule learning task has been stated within the ILP paradigm by Nicolas Helft [8] using the logic notion of minimal models of a theory $\mathcal{M}^+(Th) \subseteq \mathcal{M}(Th)$:

Given observations $\mathcal{E}$ in a representation language $\mathcal{L}_\mathcal{E}$ and background knowledge $\mathcal{B}$ in a representation language $\mathcal{L}_\mathcal{B}$,

find the set of hypotheses $\mathcal{H}$ in $\mathcal{L}_\mathcal{H}$, which is a (restricted) first-order logic, such that

(1) $\mathcal{M}^+(\mathcal{B} \cup \mathcal{E}) \subseteq \mathcal{M}(H)$

(2) for each $h \in \mathcal{H}$ there exists $e \in \mathcal{E}$ such that $\mathcal{B}, \mathcal{E} - \{e\} \not\models e$ and $\mathcal{B}, \mathcal{E} - \{e\}, h \models e$ (necessity of h)

(3) for each $h \in \mathcal{L}_\mathcal{H}$ satisfying (1) and (2), it is true that $\mathcal{H} \models h$ (completeness of $\mathcal{H}$)

(4) $\mathcal{H}$ is minimal.

This learning task is solved by some inductive logic programming (ILP) systems (e.g., RDT [12], CLAUDIEN [6]), LINUS [13] and INDEX [7]). The learning task itself is not restricted to learning from databases. However, for the application to databases the selected tuples have to be re–represented as (Prolog) ground facts. This time consuming work has to be redone, whenever the chosen representation turns out to be inadequate. One advantage of our chosen approach is, that we only have to re–represent the mapping of predicates to database relations, and the data stays unchanged in the database. In general, ILP algorithms trade in the capability to handle large data sets for increased expressiveness of the learning result. We aim at learning from existing databases directly.

The paper is organized as follows. First, the RDT algorithm is summarized and it is shown how we enhanced it to become RDT/DB which directly accesses relational databases via SQL. The time complexity of RDT/DB is presented in terms of the size of its hypothesis space. The analysis of the hypothesis space indicates, where to further structure the hypothesis space in order to make learning from very large data sets feasible. Second, we present the integration of background knowledge into the learning process and its explicit representation in the database. Third, we discuss our approach with respect to related work and applications. We argue that relational learning with a direct database access is important and feasible for KDD applications.

2 Applying ILP to Databases

ILP rule learning algorithms are of particular interest in the framework of KDD because they allow the detection of more complex rules than the ones presented above. Until now, however, they have not been applied to commonly used relational database systems. Since the demand of KDD is to analyze the databases that are in use, we have now enhanced RDT to become RDT/DB, the first ILP rule learner that directly interacts with a database management system.

The Interaction Model

For RDT/DB we have developed an interaction model between the learning tool and the ORACLE V7 database management system.

The data dictionary of the database system informs about the relations, attributes and their data types, and the keys of the database relations. This information is used in order to map database relations and attributes to predicates of RDT/DB's hypothesis language. Note, that only predicate names and arity are stored in RDT/DB as predicate declarations, not a transformed version of the database entries! Hypothesis generation is then performed by the learning tool, instantiating rule schemata top-down breadth-first.

For hypothesis testing, SQL queries are generated by the learning tool and sent to the database system. The coupling is realised by a TCP/IP connection between ORACLE and the learning tool. An interface between Prolog, the programming language in which RDT/DB is implemented, and ORACLE has been written in C. This approach of loosely coupling both systems has several advantages. It allows us to use two machines, one dedicated database server and one machine for the learning tool. Since we only transmit SQL queries, whose answer set consists of only one number, the network connection will not tend to be a bottleneck. The use of SQL as a query language gives us the opportunity to access almost any commercial database systems, without changing the learning tool.

RDT/DB

RDT/DB uses the same declarative specification of the hypothesis language as RDT does in order to restrict the hypothesis space, see for details [12]. The specification is given by the user in terms of rule schemata. A rule schema is a rule with predicate variables (instead of predicate symbols). In addition, arguments of the literals can be designated for learning constant values. A simple example of a rule schema is:

$$mp\_two\_c(P1, P2, P3, C): \quad P1(X1, C) \& P2(Y, X1) \& P3(Y, X2) \rightarrow P1(X2, C)$$

Here, the second argument of the conclusion and the second argument of the first premise literal is a particular constant value that is to be learned.

```
RDT/DB(Q)
set RS and LEAVES to ∅
for all known rule models R do
    if the conclusion C is unifiable with Q then push RΣ, Σ = {C/Q}, on RS
endfor
while RS ≠ ∅ do
    pop a most general (wrt. ≥_RS) rule model R from RS
    instantiate-and-test(R, TOO-GENERAL)
    for all X ∈ RS, X ≤_θ R do
        pop X from RS
        for all Y ∈ TOO-GENERAL do
            for all really different Σ : Yσ ≥_RS X Σ do
                push XΣ on RS
            endfor
        endfor
    endfor
endwhile
```

Figure 1: RDT/DB top–level loop

For hypothesis generation, RDT/DB instantiates the predicate variables and the arguments that are marked for constant learning. A fully instantiated rule schema is a rule. An instantiation is, for instance,

$$region(X1, europe)\&licensed(Y, X1)\&produced(Y, X2) \rightarrow region(X2, europe)$$

In the example, it was found that the cars which are licensed within Europe have also been produced within Europe.

The rule schemata are ordered by generality: for every instantiation of a more general rule schema there exist more special rules as instantiations of a more special rule schema, if the more special rule schema can be instantiated at all. Hence, the ordering reflects the generality ordering of sets of rules. This structure of the hypothesis space is used when doing breadth-first search for learning. Breadth-first search allows to safely prune branches of sets of hypotheses that already have too few support in order to be accepted.

Figures 1 and 2 show the pseudocode skeleton of the RDT/DB algorithm, which remained the same as in RDT (cf. [17][Chap. 6]). RS and LEAVES represent the search status. At the beginning, RS contains all rule models, which conclusions are unifiable with Q. These rule models will be instantiated and tested breadth first according to $\leq_{RS}$ (θ-subsumption defined on rule models). The data structure LEAVES stores all hypotheses, which are either accepted or too special. Predicate variables P in rule models will be instantiated according to $\leq_P$, an ordering which reflects the linkage of the variables of that model. Predicates which are unifiable with the predicate variable P have to be compatible with the defined topology (if any is defined), and more important they have to be sort compatible with instantiated predicates of that hypothesis.

In our database setting sort compatible is equivalent with type compatible, if we do not have any other information. Types are e.g. NUMBER, DATE, or VARCHAR2. But if we have a more fine grained sort or type hierarchy as additional background knowledge, it will be used instead.

```
instantiate-and-test(HYPO, TOO-GENERAL)
while HYPO ≠ ∅ do
    pop a hypothesis H from HYPO;
    if there are premises with uninstantiated predicates in H
    then let PREM be the smallest (wrt. ≤_P) premise and P its predicate;
        for all predicates p which are arity compatible with P and topology
                compatible with the conclusion of H and sort compatible with H do
            set H to HΣ, Σ ={P/p};
            test(H);
        endfor
    else select a constant T to learn with smallest δ(T);
        for all terms t suitable for T do
            set H to Hσ, σ ={T/t};
            test(H)
        endfor
    endif
endwhile

test(H)
if the instantiated part of H is not a specialization (≥_θ) of an element of LEAVES
then test the instantiated part of H on the database
     if H is not too special (i.e. is acceptable)
     then if all premises of H are instantiated
          then if H is acceptable wrt. all criteria
               then push H onto LEAVES;
                    assert H in the KB
               else push H onto TOO-GENERAL
          else push H onto HYPO
     else push H onto LEAVES
endif
```

Figure 2: RDT/DB instantiate-and-test subroutine

In the next step, the (partly) instantiated rule model will be tested on the database, i.e. we compute the acceptance criterion. If the hypothesis is too special, it will be inserted into LEAVES. Otherwise not fully instantiated hypotheses will be inserted into HYPO for further instantiations later on. Fully instantiated and accepted hypotheses will be inserted into LEAVES, rejected hypotheses wrt. the acceptance criterion are too general and will be inserted into TOO-GENERAL.

A rule is tested by SQL SELECT–statements. For instance, the number of supporting tuples, $pos(H)$, for the rule above is determined by the following statement:

```
SELECT COUNT (*)
       FROM vehicles veh1, vehicles veh2,
            regions reg1, regions reg2
       WHERE reg1.place = veh1.produced_at
             and veh1.ID = veh2.ID
             and veh2.licensed = reg2.place
             and reg1.region = 'europe'
             and reg2.region = 'europe';
```

The number of contradicting tuples, $neg(H)$, is determined by the negation of the condition which correspond to the rule's conclusion:

```
SELECT COUNT (*)
       FROM vehicles veh1, vehicles veh2,
            regions reg1, regions reg2
       WHERE reg1.place = veh1.produced_at
             and veh1.ID = veh2.ID
             and veh2.licensed = reg2.place
             and reg2.region = 'europe'
             and not reg1.region = 'europe';
```

The database with two relations being:

vehicles:

ID	produced_at	licensed
fin_123	stuttgart	ulm
fin_456	kyoto	stuttgart
...	...	...

regions:

place	region
stuttgart	europe
ulm	europe
kyoto	asia

The counts for $pos(H)$, the number of supporting tuples, $neg(H)$, the number of contradicting tuples, and $concl(H)$, the number of all tuples for which the conclusion predicate of the hypothesis holds, are used for calculating the acceptance criterion for fully instantiated rule schemata.

As this example implies, RDT/DB is able to handle negative examples. These are either explicitly stored in the database or implicitly, computed by the SQL view statement. In our vehicle application, "negative" facts are for instance those cars, which do not have a fault. They will be constructed by a view, which computes the difference from the table with information about all cars, and the table with information about cars, which have a fault. Since the result of every view is a "virtual table", they underlie the same mapping procedure as every table, see below. Using mapping 1, we have for example $faulty(ID)$ and $notfaulty(ID)$. Then it is up to the user to state that $notfaulty(ID) = not(faulty(ID))$. Obviously, negative facts are not restricted to appear only in the conclusion, but they can also appear in the premise of rules.

No matter how we build negative examples, we have to generate two SQL queries, one for counting $pos(H)$ and one for $neg(H)$. As is the case in RDT, the user can either specify a pruning criterion and an acceptance criterion or only one of them. In the latter case the other criterion will be computed automatically. However, as long as only partially instantiated rule schemata will be tested and if a separate pruning criterion is defined, which is advisable, we can save one SQL query, namely computing $neg(H)$. A typical pruning criterion would be $pos(H) \leq 100$, saying do not accept hypotheses which do not cover at least 100 examples. Only if we test a fully instantiated rule schema which is not to special, according to the pruning criterion, we have to compute the acceptance criterion, i.e. compute $neg(H)$. Here, a typical example for an acceptance criterion is $(pos(H)/concl(H) - neg(H)/concl(H)) \geq 0.8$.

The acceptance criterion enables the user to enforce different degrees of reliability of the learning result, or, to put it the other way around, tolerate different degrees of noise.

Furthermore and like in RDT the user can use the figures predicted and total in his acceptance and pruning criteria. Predicted counts the number of conclusion instances, which are neither true nor false, and total equals pos + neg + predicted. But he must be aware of one very important consequence of doing this. The figure which will be counted on the database is always total, and predicted will be computed as in the formula above. This means, that only tables, which correspond to the premise literals will be joined for computing total. However this is not always possible, and then the Cartesian product has to be used, which is a very bad idea. Thus computing these figures is disabled by default. Let us consider the rule $p(A, C)$ & $q(B, D) \rightarrow r(A, B)$. For counting total, we cannot use a join, because table r is not part of the query and therefore the join condition $r.A = p.A$ and $r.B = q.B$ is missing. If these two measures predicted and total are needed, then the user has to ensure that the variables in the conclusion predicates are still linked.

On the other hand, this is not a serious limitation, because we consider the rule learning task, and not the classification and prediction learning problem.

Analysis of the Hypothesis Space

The size of the hypothesis space of RDT/DB does not depend on the number of tuples, but on the number of rule schemata, r, the number of predicates that are available for instantiations, p, the maximal number of literals of a rule schema, k. For each literal, all predicates have to be tried. Without constant learning, the number of hypotheses is $r \cdot p^k$ in the worst case. As k is usually a small number in order to obtain understandable results, this polynomial is acceptable. Constants to be learned are very similar to predicates. For each argument marked for constant learning, all possible values of the argument (the respective database attribute) must be tried. Let i be the maximal number of possible values of an argument marked for constant learning, and let c be the number of different constants to be learned. Then the hypothesis space is limited by $r \cdot (p \cdot i^c)^k$.

The size of the hypothesis space determines the cost of hypothesis generation. For each hypothesis, (at most) two SQL statements have to be executed by the database system, see above. These determine the cost of hypothesis testing.

Here, the size of the hypothesis space is described in terms of the representation RDT/DB uses for hypothesis generation. The particular figures for given databases depend on the mapping from RDT/DB's representation to relations and attributes of the database.

An immediate mapping would be to let each database relation become a predicate, the attributes of the relation becoming the predicate's arguments.

Mapping 1: For each relation R with attributes $A_1, \ldots, A_n$, a predicate $r(A_1, \ldots, A_n)$ is formed, r being the string of R's name.

Learning would then be constant learning, because it is very unlikely that a complete database relation determines another one. Hence, p would be the number of relations in the database. This is often a quite small number. However, c would be the maximal number of attributes of a relation! All constants in all combinations would have to be tried. Hence, this first mapping is in general not a good idea.

If we map each attribute of each relation to a predicate, we enlarge the number of predicates, but we reduce constant learning.

Mapping 2: For each relation R with attributes $A_1, \ldots, A_n$, where the attributes $A_j, \ldots, A_l$ are the primary key, for each $x \in [1, \ldots, n] \backslash [j, \ldots, l]$ a predicate $r\_AX(A_j, \ldots, A_l, A_x)$ is formed, where AX is the string of the attribute name.

If the primary key of the relation is a single attribute, we get two–place predicates. The number of predicates is bounded by the number of relations times the maximal number of attributes of a relation (subtracting the number of key attributes). Since the number of constants to be learned cannot exceed the arity of predicates, and because we never mark a key attribute for constant learning, c will be at most 1. This second mapping reduces the learning task to learning k–place combinations of constant values.

A third mapping achieves the same effect. Attribute values of the database are mapped to predicates of RDT/DB.

Mapping 3: For each attribute A_i which is not a primary key and has the values $a_1, \ldots, a_n$ a set of predicates $r\_AI\_ai(A_j, \ldots, A_l)$ are formed, $A_j, \ldots, A_l$ being the primary key.

Using the results of NUM_INT[2], a fourth mapping can be applied, which has turned out to be quite powerful. In fact, mapping 3 can be seen as the special case, where all intervals consist of only one value.

Mapping 4: For each attribute A_i which is not a primary key and for which intervals of values have been computed, $< a_{1p}, a_{1q} >, \ldots, < a_{np}, a_{nq} >$, a set of predicates $r\_AI < a\_ip, a\_iq > (A_j, \ldots, A_l)$ is formed, $A_j, \ldots, A_l$ being the primary key.

A predicate $table\_costs\_ < 10, 100 > (ID)$ would be mapped on the table *Table*, having the attributes *ID* and *Costs*. It is true, if the values for the attribute *Costs* are in the range of 10 to 100. This predicate corresponds to a disjunction of values. And even if we have this type of predicates in the premise of a rule — a conjunction of predicates, one or more incorporating disjunctions of values — we only increase the computational complexity slightly. The reason why this is the case is quite simple. In principal, every predicate in a rule will be mapped on one table, and all tables will be joined using an equi–join. Having this kind of a predicate, we use instead of an equijoin a θ–join with θ being the expression $Costs \geq 10$ and $Costs \leq 100$.

Instead of using one comparison, e.g. $a = b$, we, i.e. the database system, uses two comparisons for predicates of the fourth mapping type. This is only a small increase in computational time, because the most time consuming part is the join operation itself. Furthermore, we are able to use every valid SQL–expression for θ-joins in a mapping. And instead of NUM_INT we can use any other algorithm, whose clusters can be described with these expressions.

Moreover, the user is not obliged to use only these four mappings. If he wants to, he can augment the second to fourth mapping with additional attributes in the predicate, e.g. $r\_AX(A_j, \ldots, A_l, A_x, A_y)$ or $r\_AI\_ai(A_j, \ldots, A_l, A_y)$.

One further mapping which can be applied, consists of leaving out a part of or the whole key in any of the above mappings. Having a table with three attributes A, B, and C, A is the key, the user might be only interested in the different combinations of

[2]The learning algorithm NUM_INT discovers intervals of numerical values, based on a gap approach, for details cf. [18].

B and C. Then he can specify a predicate $combinations(B, C)$. Using this predicate in a hypothesis, the SQL generator takes care of three different things. First, a *group by B, C* statement will be inserted into the query. Second, a projection onto B and C will be build, and third, only distinct tuples will be counted, otherwise the $pos(H)$ and $neg(H)$ figures would be incorrect, i.e. tuples would be counted twice.

Having presented these different mappings, it is quite possible, that the premise of a rule to be tested on the database will contain two predicates, which will be mapped on the same table, key, and attribute with identical variable bindings concerning the key, e.g. $r\_A < 1, 10 > (Key, B)$ and $r\_A < 11, 100 > (Key, C)$ are two predicates, which will be mapped on table r with attributes KEY, A, B, and C. Let $r\_A < 1, 10 > (Key, B)$ & $r\_A < 11, 100 > (Key, C)$ & $p(C) \rightarrow q(Key, B, C)$ be the rule to be tested.

It is obvious, that it is impossible that both predicates can be true at the same time, i.e. in the premise of a rule. Since the attribute *Key* is a key, and the intervals $< 1, 10 >$ and $< 11, 100 >$, defined on attribute A, do not overlap, all tuples of relation r will be divided into two disjoint sets, regardless of the additional attributes B and C in the predicates. All these cases will be detected by RDT/DB, and the hypothesis will be discarded as too special, because $pos(H)$ will be zero.

The general principle, which lays behind this observation, are functional dependencies between attributes. There are some more cases, where the exploitation of data dependencies from database theory will yield the result, without actually sending the query to the database. One fast and simple, but nevertheless effective approach, is to plug a semantic query optimizer into our interaction model, e.g. the one described by Bell [2]. Its task would be to take the generated query from RDT/DB, optimize it, and then send it to the database or to return immediately the result, if it can already be deduced, without consulting the database.[3]

One last aspect of these different mappings we should mention is that we make always use of the sorts, or in database terminology types, of the attributes. That means that the predicate definitions encompass argument sort declarations, which will be exploited by RDT/DB as does RDT with sorts. If we have no further background knowledge about these types, we use the information from the data dictionary, for instance *Costs* is of type NUMBER or *ID* is of type VARCHAR2. But without any change, type hierarchies can be exploited[4].

To summerize this section, it becomes clear that the restriction of the hypothesis space as it is given by RDT/DB's hypothesis generation can lead to very different hypothesis spaces depending on the mapping from database attributes to predicates. Only when reducing the learning task from finding all valid combinations of attribute values to finding k–place combinations, learning becomes feasible on large databases.

3 The Necessity of Background Knowledge

In the course of an on–going project at Daimler–Benz AG on the analysis of their data about vehicles, we have applied our learning tool. The database with all vehicles of a certain type of Mercedes — among them some cases of warranty — is about 2.6 Gigabytes large. It consists of 40 relations with about 40 attributes each. The main topic of interest for the users is to find rules that characterize warranty cases and structure them into meaningful classes.

[3] This integration work still has to be done.

[4] The implementation of RDT/DB always works with sort taxonomies, but Oracle delivers only "flat" hierarchies.

In a first approach, two systems, CLAUDIEN and RDT/DB, have been applied, which both could well find rules. The first results of both were better than guessing on the basis of frequencies. For car variants with a certain probability of being claimed to be faulty by the customer, properties could be found which select a set of cars with a higher probability of being claimed as a case of warranty. That is, some factors that indicate warranty cases to a certain degree could be found. An example of an uninteresting (although almost 100% correct) set of rules found by RDT/DB looks like the following one:

$$engine\_variant(VIN, 123) \rightarrow engine\_type(VIN, 456)$$

A rule learned by CLAUDIEN looked like the following[5]

warranty (_VIN) IF
_Car_variant = 987 *[accuracy 0.159]*
_Engine_type = 654 *[accuracy 0.161]*

However, both types of rules were not interesting, since they expressed what is known to all mechanics, because certain combinations have been sold more often than others. Moreover, these factors were far away from hints to causal relationships. Whereas the former rules present a kind of knowledge, which is too general to be interesting, the latter ones are not interesting too, because the hypothesis language is too special. The reason is that the hypothesis language does not offer sets of attribute values, but only the single values.

The problem is how to find the appropriate level of detail that allows to learn interesting rules. We have to introduce descriptors into the hypothesis language that are more general than constant values. The classical approaches to overcome this problem are hardly applicable when learning from real databases. First, taxonomies of descriptors have served this purpose, for instance, in conceptual clustering [14] or the versions space approach [16]. There, however, the taxonomies were given by the user of the learning system. Here, we do not know of such hierarchies of sets of attribute values but have to find them. Second, methods for constructive induction exist that introduce more abstract descriptors into the hypothesis language, but they presuppose classified examples — which is also not given in our application. Third, unsupervised clustering of attribute values is not feasible. Marcel Holsheimer and Arno Siebes have shown that the complexity of finding what they call set–descriptions (i.e. disjunctions of constants) is exponential in the number of all attribute values of all relations as well as in the number of tuples of the database [9]. Hence, there are only a few options left.

One alternative, which we only mention here briefly, consists of extending RDT/-DB into a multistrategy learning system. It then uses two algorithms, which we have developed, that structure attributes and attribute values, respectively: FDD (Functional Dependency Detection) [3] finds a generality order of attributes by detecting functional dependencies in the database. NUM_INT finds a hierarchy of intervals in numerical (linear) attributes without reference to a classification. This approach is discussed in detail in [18].

In the following we concentrate on another alternative, namely the use of background knowledge. We will demonstrate both, that we can use it in an appropriate way, and that we need it to learn more interesting rules.

[5]Note that neither the rule nor the percentages involved are the true ones so that no conclusions on Mercedes cars can be made.

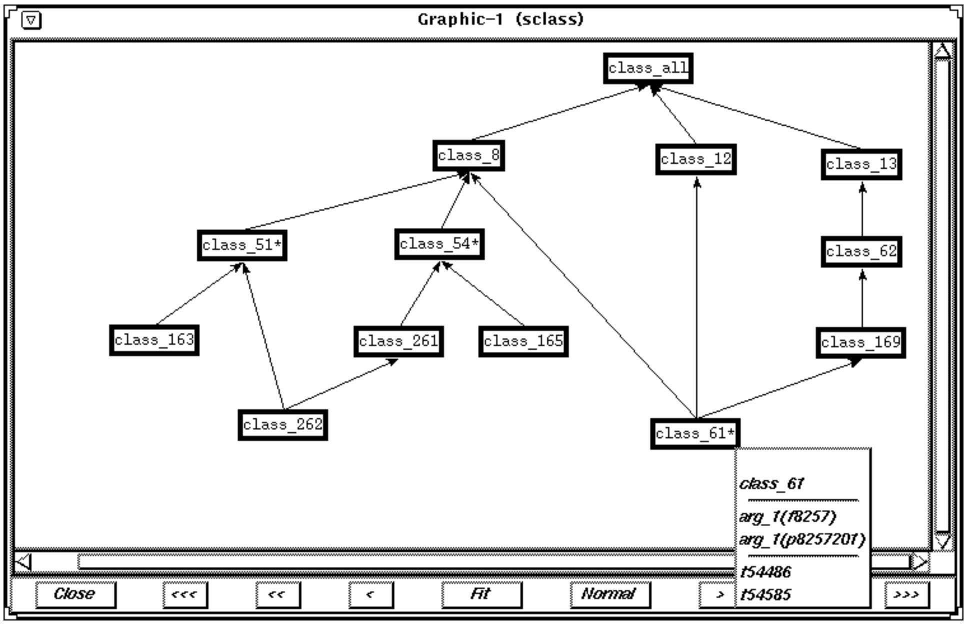

Figure 3: Part of the sort lattice computed by STT.

As pointed out above and to stress the importance of the second D in KDD, it is essential to enable algorithms to deal with great amounts of data which are stored in databases. This leads to taking complexity issues seriously. Given the declarative syntactic bias in terms of rule schemata, the hypothesis space in a real-world application can still be very high, because the number of attributes (determining p) and attribute values i is usually high. This leads to the demand of restricting the number of attributes and attribute values used for learning. Reducing the number of attribute values of an attribute can be done by climbing a hierarchy of more and more abstract attribute values. If this background knowledge is not available, it has to be learned.

Forming a hierarchy of nominal attribute values – STT

Background knowledge is most often used in a KDD framework in order to structure sets of attribute values, that is, the background knowledge offers a hierarchy of more and more abstract attribute values. However, background material is often unstructured. In this case, it needs some structuring before it can be used for learning from the database. For this task, we use STT, a tool for acquiring taxonomies from facts [10][6]. For all predicates it forms sets for each argument position consisting of the constant values which occur at that position. The subset relations between the sets is computed. It may turn out, for instance, that all values that occur as second argument of a predicate p_1 also occur as first argument of predicate p_2, but not the other way around. In this case, the first set of values is a subset of the second one. We omit the presentation of other features of STT. Here, we apply it as a fast tool for finding sets of entities that are described by several predicates in background knowledge.

[6] A detailed description can be found in [17].

We have represented textual background material as one–ary ground facts. The predicates express independent aspects of attribute values of an attribute of the database. These attribute values are at argument position. Different predicates hold for the same attribute value. For 738 predicates and 14 828 facts, STT delivered a graph with 273 nodes[7]. The graph combines the different aspects. Since the sets of attribute values are meaningful for the user, he can select a level of the graph. Selecting a layer with 4 nodes, we introduced 4 new Boolean predicates into the database. The new attributes replace the original database attribute A in $\mathcal{L}_{\mathcal{H}_i}$. This increases p by 3, but decreases i by almost 10 000, since we disregard the original database attribute in $\mathcal{L}_{\mathcal{H}}$.

Also the introduction of the new attributes on the basis of STT's output led to learning more interesting rules. The background material is the mechanic's workbook of vehicle parts, classified by functional (causal) groups of parts, spatial groupings (a part is close to another part, though possibly belonging to another functional group), and possible faults or damages of a part. The vehicle parts are numbered. *t54486*, for instance, is a certain electric switch within the automatic locking device. The functional groups of parts are also numerically encoded. *f8257*, for instance, refers to the locking device of the vehicle. The fact *f8257(t54486)* tells that the switch *t54486* belongs to the locking device. The spatial grouping is given by pictures that show closely related parts. The pictures are, again, numerically encoded. *p8257201*, for instance, refers to a picture with parts of the electronic locking device that are closely related to the injection complex. *p8257201(t54486)* tells that the switch belongs to the spatial group of the injection. Possible damages or faults depend, of course, rather on the material of the part than its functional group. All different types of damages are denoted by different predicates (e.g., *s04* indicates that the part might leak). These three aspects are each represented by several classes and subclasses. Each part belongs at least to three groups (to a functional one, to a spatial one, and a type of possible error), frequently to several subclasses of the same group. The combination of these three aspects has led to surprising classes found by STT. Looking at Figure 3, class_61 comprises two switches, *t54486* and *t54585*. They are the intersection of three meaningful classes:

class_169 : here, several parts of the injection device are clustered. These are parts such as tubes or gasoline tank. Up in the hierarchy, parts of the functional group of injection and then (class_13) parts of gasoline supply in general are clustered.

class_12 : here, parts of the dashboard are clustered, among them the display of the locking device[8] (protection from theft).

class_8 : here, operating parts of the engine are clustered that serve the injection function.

The illustration shows that mechanics can easily interpret the clusters of parts, and the hierarchy learned by STT is meaningful. The intersection classes are very selective where classes such as, e.g., class_13 cover all parts of a functional group (here: gasoline supply).

[7] Since STT took about 8 hours, it cannot be subsumed under the fast algorithms. However, its result is computed only once from the background material which otherwise would have been ignored.

[8] The locking device interrupts the injection function.

4 Discussion

Complexity reduction is of highest importance when learning from databases. As the costs of hypothesis testing are beyond our control, but up to the database system,[9] the costs of hypothesis generation must be reduced.

The rule learning algorithm RDT is particularly well suited for KDD tasks, because its complexity is not bound with reference to the number of tuples but with reference to the representation of hypotheses.

On the one hand side this claim is also true, if we consider e.g. FOIL in its present state, as a stand alone program which handles all data itself, and interestingly, which uses database join operations internally. But on the other hand there are many reasons, why a hypothetical FOIL/DB seems not feasible. First, FOIL needs negative examples in order to learn. Since nobody will try to generate negative examples by means of the CWA in this database setting, they have to be given. Second, in the inner loop of FOIL, the training set grows, if the literal introduces new variables. In a database setting, where we have many examples just in the beginning, this effect can be quite dramatic. And third, how do we realize the covering?

The straight forward ideas of deleting covered tuples or copying uncovered tuples are not feasible. In our database, to copy or to recreate one big table with an index on the key takes three hours, one join of this table with itself 2 minutes. So you will neither recreate your database for each learning run nor wait hours before the learning of the next clause starts during one learning run. The covered examples of one learned clause of this FOIL/DB can be represented by a view, let us say VIEW_A. The reduced training set (in the outer loop of FOIL) would be VIEW_B = TRAINING_EXAMPLES – VIEW_A. But by doing this, FOIL/DB would always work on the original table, loosing the main advantage which is responsible for the well known efficiency of FOIL!

To summerize these remarks, the main advantage of FOIL, its explicit manipulation of tuples, would become its disadvantage, if ported on databases. Even one can say that FOIL on databases would become very close to RDT/DB concerning the testing of hypotheses, i.e. joining tables and counting. But RDT/DB has the advantages, that it can learn without negative examples and that the example set does not grow.

Furthermore, by its top-down, breadth-first search RDT/DB allows to safely prune large parts of the hypothesis space. The declarative syntactic bias is extremely useful in order to restrict the hypothesis space in case of learning from very large data sets. In order to directly access database management systems, RDT was enhanced such that hypothesis testing is executed via SQL queries. However, the syntactic language bias alone is not enough for applying RDT to real-world databases without reducing it to the expressiveness of an algorithm such as KID3, for instance. If we want to keep the capability of relational learning but also want to learn from all tuples of a large database, we need more restrictions. They should lead to a reduction of the number p of predicates or the maximal number i of attribute values for an attribute. The restriction should be domain–dependent, for which our different mappings deliver the needed means.

We have applied several methods (different mappings and background knowledge) for reducing the number of hypotheses in concert. Because we have chosen an algorithm with a user given declarative bias, we could investigate how to restrict and structure the hypothesis space.

[9] This is not the complete truth, as one could imagine to prefer hypotheses that do not require a JOIN in the test to those that do.

- A syntactic restriction of the form of hypotheses is effective because it cuts what is in the exponent of the formula defining its worst–case size: the maximal number of literals. Also the number of constant values to learn about is indicated by the user — another figure in the exponent of the formula.

- The learning task is also restricted by the chosen mapping from predicates of the learning tool to database attributes. Instead of learning all combinations of all attribute values, only a k–place combination is learned.

Using these restrictions, we successfully learned 76 rules about 6 different conclusion predicates from an excerpt of the vehicles database.

It is an issue for discussion, whether the user should select appropriate levels from the learned hierarchies of the "service algorithms", here STT, or not. We have adopted the position of [4] that the user should be involved in the KDD process. On the one hand, the selection of one layer as opposed to trying all combinations of all hierarchies makes learning feasible also on very large databases. On the other hand, the user is interested in preliminary results and wants to have control of the data mining process. The user is interested in some classes of hypotheses and does not want to specify this interest precisely as yet another declarative bias.

There are two main differences between our framework for KDD and the KEPLER KDD workbench, which offers a variety of ILP algorithms together with a set–oriented layer for data management [20]. KEPLER allows the user to call various learning tools and use the results of one as input to another one. In our learning environment, the user is involved as an oracle for selecting a part of the background knowledge, see above. And more importantly, we have moved to directly accessing databases via SQL.

Experiments on the data are an ongoing effort [10]. We are planning systematic tests that compare quality and time spent on learning for the various possibilities of applying different mappings of predicates on different tables and selecting different numbers of classes from the background knowledge and convert them into binary attributes in the database. Since the tables in the vehicle database heavily vary in their numbers of tuples, it is still possible although unlikely, that under certain conditions even mapping 1 has some advantages.

Right now, we can state that without using various methods in concert, we achieved valid but not interesting results. Some types of relations could not at all be learned without background knowledge. Hence, the advantage of our approach is not an enhancement of a method that works already, but is that of making relational learning work on real-world databases at all. Since this break-through has been achieved, we can now enhance the algorithms.

Acknowledgments

Work presented in this paper has been partially funded by Daimler–Benz AG, Research Center Ulm, Contract No.: 094 965 129 7/0191. We thank G. Nakhaeizadeh and R. Wirth for indicating practical needs and encouraging us to fulfill these needs using relational learning. Christian Franzel has developed the NUM_INT algorithm. Mark Siebert has acquired background knowledge about vehicle parts and has applied STT to it. H. Blokeels and L. Dehaspe have made the learning runs with CLAUDIEN. We also thank J.–U. Kietz for fruitful discussions.

[10] Since the data are strictly confidential, we cannot illustrate the increase of interestingness here.

References

[1] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, AAAI Press Series in Computer Science, chapter 12, pages 277–296. A Bradford Book, The MIT Press, Cambridge Massachusetts, London England, 1996.

[2] Siegfried Bell. Deciding distinctness of query results by discovered constraints. In Mark Wallace, editor, *Proceedings of the Second International Conference on the Practical Application of Constraint Technology*, pages 399–417, Blackpool, Lancashire, FY2 9UN, UK, 1996. The Practical Application Company Ltd.

[3] Siegfried Bell and Peter Brockhausen. Discovery of constraints and data dependencies in databases (extended abstract). In Nada Lavrac and Stefan Wrobel, editors, *Machine Learning: ECML-95 (Proc. European Conf. on Machine Learning, 1995)*, Lecture Notes in Artificial Intelligence 914, pages 267 – 270, Berlin, Heidelberg, New York, 1995. Springer Verlag.

[4] Ronald J. Brachman and Tej Anand. The process of knowledge discovery in databases: A human-centered approach. In Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, AAAI Press Series in Computer Science, chapter 2, pages 33–51. A Bradford Book, The MIT Press, Cambridge Massachusetts, London England, 1996.

[5] Y. Cai, N. Cercone, and J. Han. Attribute–oriented induction in relational databases. In G. Piatetsky-Shapiro and W. Frawley, editors, *Knowledge Discovery in Databases*, pages 213 –228. AAAI/MIT Press, Cambridge, Mass., 1991.

[6] Luc De Raedt and Maurice Bruynooghe. A theory of clausal discovery. In Stephen Muggleton, editor, *Procs. of the 3rd International Workshop on Inductive Logic Programming*, number IJS-DP-6707 in J. Stefan Institute Technical Reports, pages 25–40, 1993.

[7] Peter Flach. Predicate invention in inductive data engineering. In Pavel Brazdil, editor, *Machine Learning - ECML'93*, volume 667 of *Lecture Notes in Artificial Intelligence*, pages 83–94, 1993.

[8] Nicolas Helft. Inductive generalisation: A logical framework. In *Procs. of the 2nd European Working Session on Learning*, 1987.

[9] M. Holsheimer and A. Siebes. Data mining: The search for knowledge in databases. Technical report, CWI Amsterdam, 1993.

[10] Jörg-Uwe Kietz. Incremental and reversible acquisition of taxonomies. In *Proceedings of EKAW–88*, chapter 24, pages 1–11. 1988. Also as KIT-Report 66, Technical University Berlin.

[11] Jörg Uwe Kietz. *Induktive Analyse relationaler Daten*. PhD thesis, 1996. to appear, in german.

[12] Jörg-Uwe Kietz and Stefan Wrobel. Controlling the complexity of learning in logic through syntactic and task–oriented models. In Stephen Muggleton, editor, *Inductive Logic Programming*, chapter 16, pages 335–360. Academic Press, London, 1992.

[13] Nada Lavrac and Saso Dzeroski. *Inductive Logic Programming - Techniques and Applications.* Ellis Horwood, New York, 1994.

[14] R.S. Michalski and R.E. Stepp. Conceptual clustering: Inventing goal-oriented classifications of structured objects. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning - An Artificial Intelligence Approach Vol II*, pages 471–498. Tioga Publishing Company, Los Altos, 1986.

[15] Ryszard S. Michalski. A theory and methodology of inductive learning. In *Machine Learning — An Artificial Intelligence Approach*, pages 83–134. Morgan Kaufman, Los Altos, CA, 1983.

[16] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982.

[17] K. Morik, S. Wrobel, J.-U. Kietz, and W. Emde. *Knowledge Acquisition and Machine Learning - Theory, Methods, and Applications.* Academic Press, London, 1993.

[18] Katharina Morik and Peter Brockhausen. A multistrategy approach to relational knowledge discovery in databases. In Ryszard S. Michalski and Janusz Wnek, editors, *Proceedings of the Third International Workshop on Multistrategy Learning (MSL-96)*, Palo Alto, May 1996. AAAI Press.

[19] G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. In G. Piatetsky-Shapiro and W. Frawley, editors, *Knowledge Discovery in Databases*, pages 229 –248. AAAI/MIT Press, Cambridge, Mass., 1991.

[20] S. Wrobel, D. Wettscherek, E. Sommer, and W. Emde. Extensibility in data mining systems. In Evangelos Simoudis and Jia Wei Han, editors, *2nd Int. Conference on Knowledge Discovery and Data Mining*, Menlo Park, August 1996. AAAI Press. submitted paper, available at http://nathan.gmd.de/projects/ml/home.html.