A Multistrategy Approach to Relational Knowledge Discovery in Databases

Katharina Morik and **Peter Brockhausen**
Univ. Dortmund, Computer Science Department, LS VIII
D-44221 Dortmund
{morik, brockh}@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. A multistrategy approach allows to combine most of the advantages and exclude most of the disadvantages. More simple learning algorithms detect hierarchies that are used in order to structure the hypothesis space for a more complex learning algorithm. The better structured the hypothesis space is, the better can learning prune away uninteresting or losing hypotheses and the faster it becomes.

We have combined inductive logic programming (ILP) directly with a relational database. The ILP algorithm is controlled in a model-driven way by the user and in a data-driven way by structures that are induced by three simple learning algorithms.

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 (Kietz 1996). 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 (Agrawal *et al.* 1996). 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].

Other fast learning algorithms exploit given hierarchies and generalize attribute values by climbing the hierarchy (Michalski 1983), 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 (Cai, Cercone, & Han 1991). Similarly, the KID3 algorithm discovers dependencies between values of two attributes using hierarchies from background knowledge (Piatetsky-Shapiro 1991). 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 $C = c$ (1)

[1] In (Agrawal *et al.* 1996) the authors present a series of experiments with the algorithms and give a lower bound for finding an association rule.

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 (Helft 1987) 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 systems (e.g., RDT (Kietz & Wrobel 1992), CLAUDIEN (De Raedt & Bruynooghe 1993)), LINUS (Lavrac & Dzeroski 1994) and INDEX (Flach 1993)). For the application to databases the selected tuples are re-represented as (Prolog) ground facts. In general, ILP algorithms trade in the capability to handle large data sets for increased expressiveness of the learning result.

Given the trade-off between feasibility and expressiveness, we propose a multistrategy approach. The idea is to combine different computational strategies for the same inferential strategy (here: induction) [2]. The overall learning task is decomposed into a sequence of learning tasks. Simpler subtasks of learning can then be performed by simpler (and faster) learning methods. The simpler algorithms induce hierarchies of attributes and attribute values that structure the hypothesis space for the ILP learner. The ILP learning algorithm uses this structure for its level-wise refinement strategy. The architecture of our MSL-system for KDD is shown in figure 1.

The learning algorithms and their transmutations (Michalski 1994) are:

[2]According to (Michalski 1994) the *computational strategy* means the type of knowledge representation and the associated methods for modifying it in the process of learning. The *inferential strategy* means the primary type of inference underlying a learning process.

FDD : learning functional dependencies between database attributes by an association transmutation. The functional dependencies constitute a *more general* relationship of database attributes. This relationship determines a sequence of more and more detailed hypothesis languages.

NUM_INT : learning a hierarchy of intervals of linear (numerical) attribute values by an agglomeration transmutation. From this hierarchy, the user selects the most interesting intervals which are then introduced as more abstract attribute values into the hypothesis language.

STT : learning a hierarchy of nominal attribute values from background knowledge by an agglomeration transmutation. More abstract attribute values are introduced into the database (database aggregation).

RDT/DB : learning rules in a restricted first-order logic by a generalization transmutation. RDT/DB searches in a hypothesis space that is tailored to the application by the user and the preceding three algorithms.

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, the algorithms that preprocess data for RDT/DB are characterized, namely FDD, NUM_INT, STT. Third, we discuss our approach with respect to related work and applications. We argue that multistrategy learning is important for KDD applications.

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 such as the one 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.

RDT/DB

RDT/DB uses the same declarative specification of the hypothesis language as RDT does in order to restrict

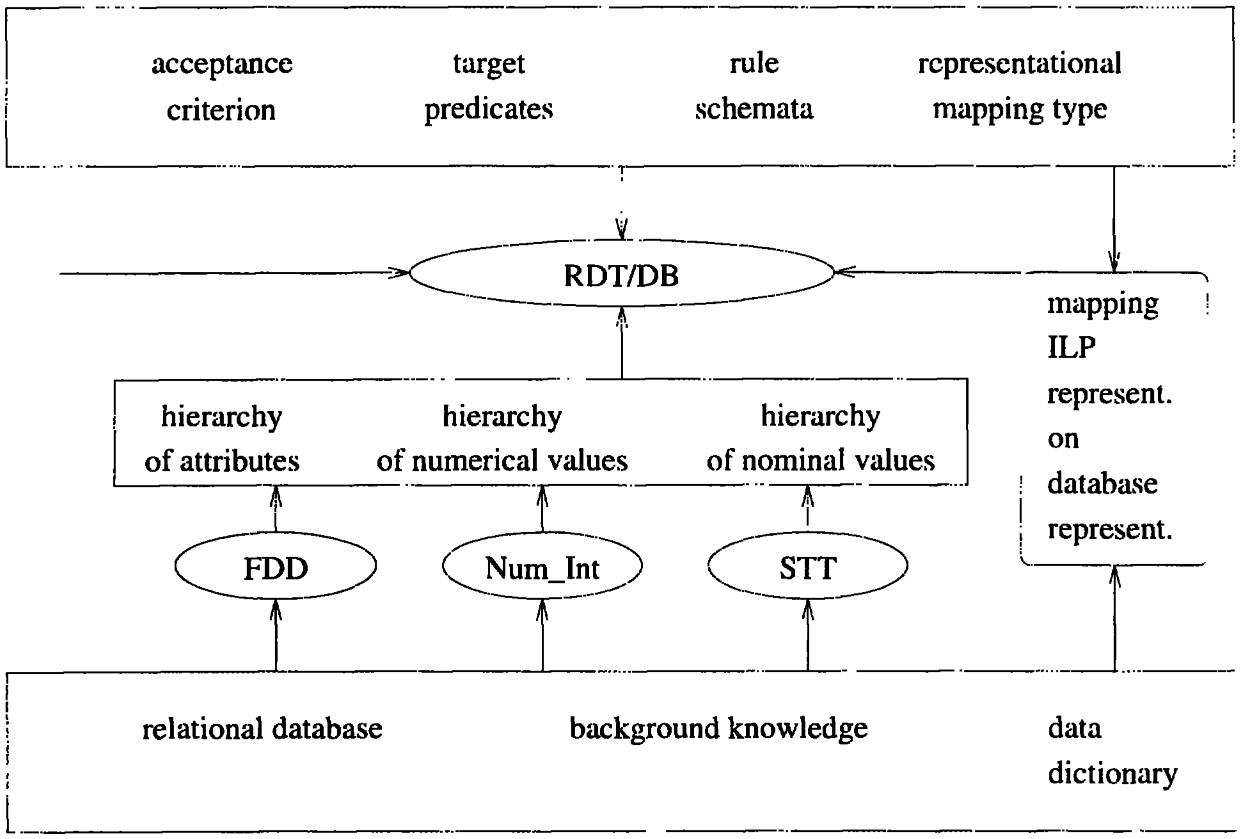

Figure 1: A multistrategy framework for KDD

the hypothesis space (see for details (Kietz & Wrobel 1992)). 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:
$mp\_two\_c(C, P1, P2, P3)$:
$P1(X1, C) \& P2(Y, X1) \& P3(Y, X2) \rightarrow P1(X2, C)$

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.

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.

Another user-given control knowledge is the acceptance criterion. The user composes it of $pos(H)$, 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. The user can enforce different degrees of reliability of the learning result, or, to put it the other way around, tolerate different degrees of noise. A typical acceptance criterion is, for instance, $(pos(H)/concl(H) - (neg(H)/concl(H)) \geq 0.8$.

For RDT/DB we have developed an interaction model between the learning tool and the ORACLE database system. The data dictionary of the database system informs about relations and attributes of the database. This information is used in order to automatically map database relations and attributes to predicates of

RDT'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 are sent to the database system. 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 $pos(H), neg(H)$, and $concl(H)$ are used for calculating the acceptance criterion for fully instantiated rule schemata.

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 polynom 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, two SQL statements have to be executed by the database system. 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 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.

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.

The analysis of the hypothesis space gives good hints for how to write rule schemata: the most general ones should have only one premise literal and the most special ones not more than 3 as this keeps k small; there should be as few schemata as possible in order to keep r small; at most one constant should be marked for learning in order to keep c small.

Further Control of Complexity

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.

Some attributes are of particular interest for the user of the KDD system. For instance, in an analysis of warranty cases, the attribute that expresses, whether an item was a warranty case or not, is the most relevant one. The user can specify this attribute as the target for learning. However, the attributes that determine a warranty case cannot be specified by the user. It is the task of learning to identify them! This is the point where FDD comes into play. FDD learns a partial generality ordering on attributes. A sequence of learning passes of RDT/DB is started, each pass using only the most general and not yet explored attributes of the attribute hierarchy for characterizing the user-given target attributes. The sequence is stopped as soon as hypotheses are found that obey the acceptance criterion. That means, after successfully learning in language $\mathcal{L}_{\mathcal{H}_i}$, no new learning pass with $\mathcal{L}_{\mathcal{H}_{i+1}}$ is started. Of course, the user may start RDT/DB with the next language and continue the sequence of learning passes. Most important is, however, that the user gets an output of learned rules, in time, because p decreases remarkably. Note, that the representational bias in terms of the sequence of $\mathcal{L}_{\mathcal{H}_i}$ is a *semantic* declarative bias (as opposed to the syntactic rule schemata or the language sequences of CLINT (De Raedt 1992)). It is domain-dependent. Using the output of FDD, RDT/DB adapts its behavior to a new data set.

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. For numerical values, NUM_INT finds a hierarchy of intervals. Since this is a learning result in its own right, it is presented to the user who selects relevant intervals. These are transformed by RDT/DB into predicates that are used instead of the ones that would have been formed on the basis of the original database attribute values (according to the third mapping). Hence, p slightly increases, but i decreases a lot.

For nominal values, any fast learning algorithm that is capable of finding clusters within the values of one attribute (i.e. finds sets of attribute values) could be plugged into our multistrategy framework. The clusters are named and these names become more abstract attribute values replacing the original values. In our current application (see below), we have chosen a different approach. We have background knowledge about three different aspects of the attribute values of a given attribute A in the database. The background knowledge is used for learning a graph, where the nodes contain attribute values of the given attributes and the links establish a subset ordering. Note, that the graph forms clusters that combine two different aspects. Since the sets of attribute values are meaningful for the user, he can select a level of the graph. Each node of the chosen level becomes a binary attribute of the database. The new attributes replace the original database attribute A in $\mathcal{L}_{\mathcal{H}_i}$. Again, p slightly increases, but i decreases a lot.

Detecting Functional Dependencies — FDD

In the following we assume some familiarity with definitions of relational database theory (see, e.g., (Kanellakis 1990)). Capital letters like A, B, C denote attributes and X, Y, Z sets of attributes. A functional dependency (FD) $X \rightarrow Y$ is valid, if every pair of tuples, which agrees in its X values, also agrees in its Y values. According to Armstrongs's Axioms (Ullman 1988) and without loss of generality we only regard FDs with one attribute on the right hand side. The discovery of FDs may be visualized as a search in semi lattices. The nodes are labeled with data dependencies and the edges correspond to the *more general than* relationship as in (Savnik & Flach 1993), which implies the partial ordering.

Definition 1 *(More general FD) Let X and Y be sets of attributes such that $X \subseteq Y$, then the FD $X \to A$ is more general than the dependency $Y \to A$, or $Y \to A$ is more specific than $X \to A$.*

In contrast to the notion of a minimal cover in database theory, the algorithm FDD computes a *most general cover.* The difference is shown by the following example: The set $\{A \to B, B \to C, A \to C\}$ is most general in our sense, but not minimal according to database theory, since the transitivity rule is applicable.

Definition 2 *(Most general cover) The set of functional dependencies F of relation R ($R \models F$) is a most general cover, if for every dependency $X \to A \in F$, there does not exist any Y with $Y \subset X$ and $Y \to A \in F$.*

The hypothesis generation is a top-down, breadth-first search through the semi lattice imposed by the *more general than* relationship as in (Mannila & Räihä 1994). In order to speed up computation, we use a subset of the complete axiomatization of functional and unary inclusion dependencies and independencies (Bell 1995). Furthermore, FDD computes a partition of all attributes into three classes. The first class contains candidate key attributes, the second contains attributes with null values and only the attributes of the third class (the rest) are needed for discovery. Before actually starting the top-down search, the most specific hypothesis will be tested. If this hypothesis does not hold, then the whole semi lattice can be discarded.

FDD uses the interaction model that was described above, i.e. FDD generates SQL queries (i.e. hypotheses) and the database system computes the answer (i.e. tests the hypothesis). Figure 2 lists this kind of statements and the condition which must hold. The clue is the GROUP BY instruction. The computational costs of this operation are dependent on the database system, but it can be done in time $O(m * \log m)$.

We define a hierarchy on the involved attributes of one-place FDs in the following way:

Definition 3 *(More general attribute) Given a set of one-place functional dependencies F. The attribute C is more general than A, if $A \to C$ is an element of the transitive closure of F and $C \to A$ is not an element of the transitive closure of F.*

-
  ```
  SELECT MAX (COUNT (DISTINCT B))
  FROM R
  GROUP BY A1, ..., An                =:a1
  ```
- $a_1 = 1 \Rightarrow A_1 \ldots A_n \to B$

Figure 2: A SQL query for the computation of functional dependencies, ($B \notin \{A_1 \ldots A_n\}$)

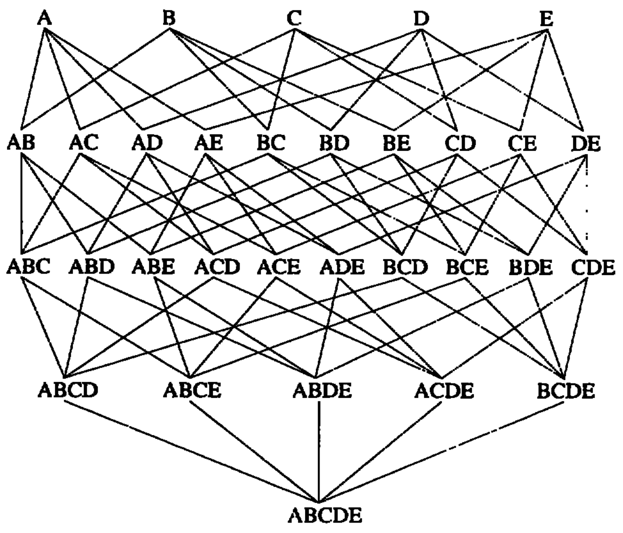

Figure 3: Search lattice for FDD.

We present a simple example to illustrate this. Let the only valid FDs in R be the following: $\{A \to B, B \to C\}$. Then we will get a hierarchy of attributes, where C is more general than B, and B more general than A. Since the three attributes are from the same relation and the FDs hold, there must be more tuples with the same value for C than for B. This follows immediately from the definition of FDs. The same is true for B and A. Furthermore, if there are cycles in the transitive closure of these one-place FDs, then all attributes within the cycle are of equal generality. Attributes entering the cycle are more general than the ones within it. Attributes leaving the cycle are more specific than attributes within it.

Although the time complexity of FDD is exponential in the worst case (Beeri *et al.* 1984), in practice, FDD successfully learned from several databases of the size of 65 000 tuples, up to 18 relations, and up to 7 attributes each. Even without any depth restriction although most of the FDs actually were one-place dependencies – FDD takes 3.5 minutes for learning 113 FDs from 18 relations with up to 6 attributes each.

We exploit this *more general* relationship on attributes for the development of a sequence of hypothesis languages. Each stays within the same syntactical structure as given by the rule schemata, but has only a subset of the attributes. Given, for instance, the FDs $\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$, the first set of predicates in $\mathcal{L}_{\mathcal{H}_1}$ includes C and neither A nor B, $\mathcal{L}_{\mathcal{H}_2}$ includes B and neither A nor C, and $\mathcal{L}_{\mathcal{H}_3}$ includes A and neither B nor C. As a result, we have a level-wise refinement strategy as in (Mannila & Toivonen 1996), that means, we start the search with hypotheses consisting of most general attributes. If these hypotheses are too general, continue with more specific attributes only.

Discretization of Numerical Attributes — NUM_INT

Numerical values offer an ordering that can be exploited. Hence, these values can be grouped with less complexity than nominal values, even if no classification is available. The idea behind NUM_INT is to order the numbers and to search for "gaps" in the stream of values. The biggest gap is supposed to be the best point for splitting up the initial interval [min, max]. The next gaps are taken to continue splitting in a top down fashion. The result is a tree with the initial interval [min, max] as the root, split intervals as inner nodes and unsplit intervals as leaves. The depth of this tree is determined by a parameter (d) which is set by the user.

The result is obtained by three conceptual steps: First, order the numbers (using the statement "select ...order by ..." via embedded SQL); second, fetch tuple by tuple (via embedded SQL) gathering all information needed for splitting; and third, build up the tree of intervals. The complexity of the three steps is as follows: Step (1) should be $O(N \log N)$, N being the number of tuples, because we select only one attribute (this is done by the database system and therefore beyond our control). In step (2) each gap has to be sorted into an array of depth d which leads to $O(N \cdot d)$. Finally in step(3) we have to insert $O(d)$ values into an ordered array of depth $O(d)$ resulting in complexity of $O(d^2)$.

Most time is consumed by simply fetching the tuples one by one via SQL. We tested NUM_INT on a database containing about 750.000 tuples: the algorithm ran 35 minutes on a SUN SPARC for a depth of 100: 2 minutes were taken for ordering, 8 minutes for internal processing and about 25 minutes for waiting on the database system to deliver the tuples. This also shows that it is essential that we touch each tuple only once and collect all information (min, max, found intervals, number of tuples in each interval) "on the fly".

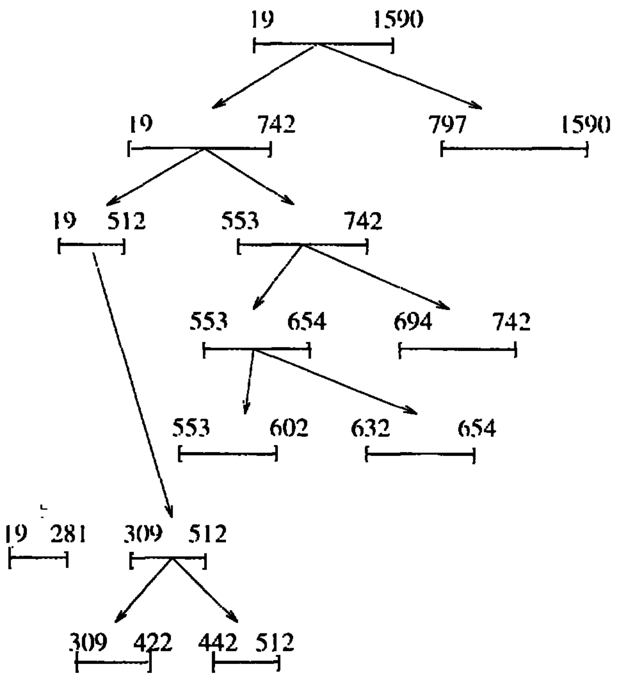

Figure 4: Hierarchy of intervals found by NUM_INT

Of course we are aware of the fact that this "gap"-approach is a quite simple one and that more sophisticated approaches for learning intervals are known (e.g., (Wettscherek & Diettorich 1995), (Pazzani 1995)). However, these algorithms are either more complex (in particular, clustering algorithms although being much more elegant are too complex to be applicable), or require a classification.

Pazzani, for instance, presented an iterative improvement approach for finding discretizations, i.e. intervals, of numeric attributes. His algorithm starts with a partition into a small number of seed intervals with equal size, and then iteratively uses split or merge operations on the intervals based on error or misclassification costs. In most cases, an appropriate number of intervals is unknown in advance, resulting in some loops of the algorithm and therefore he has to reconsider all values of the attribute. This is unfeasible in large databases. However, these algorithms are either more complex (in particular, clustering algorithms although being much more elegant are too complex to be applicable), or require a classification.

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 (Kietz 1988)[3]. 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.

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[4]. The graph combines the different aspects. Selecting a layer with 4 nodes, we introduced 4 new Boolean predicates into the database. This increases p by 3, but decreases i by almost 10 000, since we disregard the original database attribute in $\mathcal{L}_{\mathcal{H}}$.

Discussion

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. Its top-down, breadth-first search 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 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. The task of structuring the set of attributes as well as the task of structuring sets of attribute values is performed by more specialized learning algorithms. While we are convinced that this work-share between specialized and a more general learning algorithm is a powerful idea, we do not claim that the algorithms for structuring attribute values are in general the best choice. However, our architecture allows to plug in other (better) algorithms, if available.

It is an issue for discussion, whether the user should select appropriate levels from the learned hierarchies of the "service algorithms", or not. We have adopted the position of (Brachman & Anand 1996) 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. Note, however, that the user in our framework does not need to implement the interaction between the learning result of one algorithm and its impact on the other algorithm. Our multistrategy framework for KDD is a closed-loop learning approach with the user being involved as an oracle. This is a particular difference regarding the KEPLER KDD workbench, which offers a variety of ILP algorithms together with a set–oriented layer for data management (Wrobel *et al.* 1996). KEPLER allows the user to call various learning tools and use the results of one as input to another one. It is a loosely coupled system, where our framework is a tightly coupled one. Another difference is that we have moved to directly accessing databases via SQL.

In the course of an on- going project at Daimler Benz AG on the analysis of their data about vehicles, we have applied our multistrategy learning. 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. In a monostrategy approach, RDT/DB could well find rules, among them ones that are about 100% correct. However, these rules were not interesting, since they expressed what is known to all mechanics.

[3] A detailed description can be found in (Morik *et al.* 1993).

[4] 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.

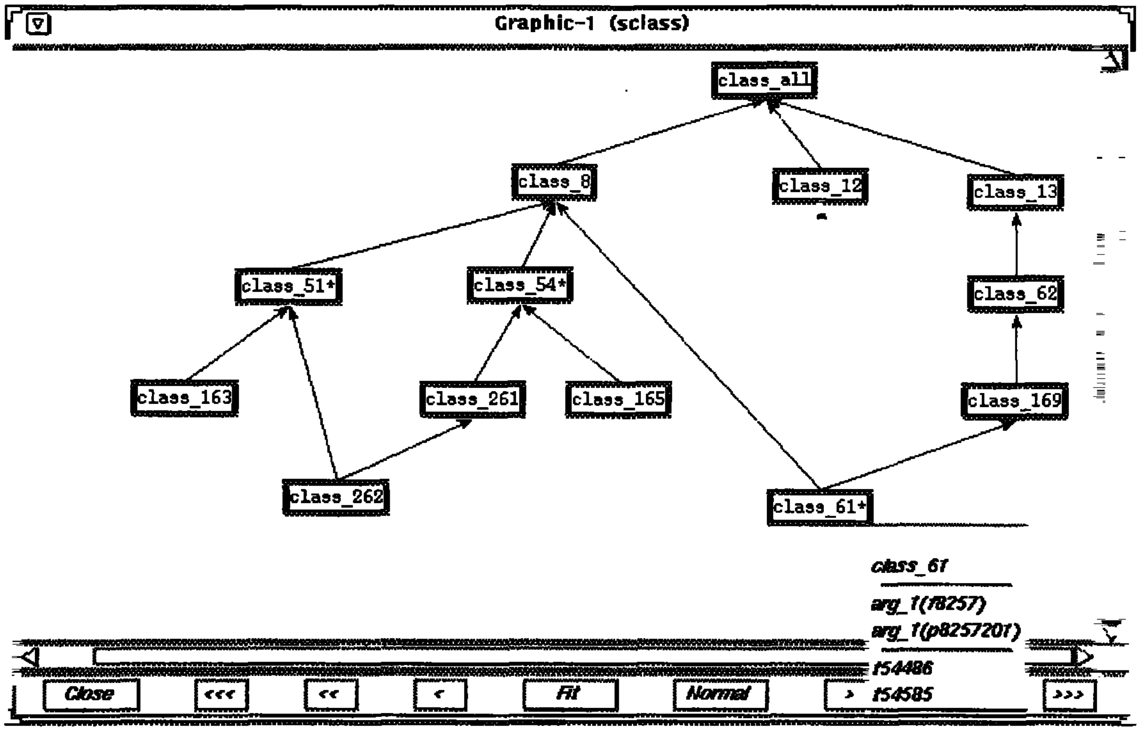

Figure 5: Part of the sort lattice computed by STT.

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

More interesting rules could only be found, when preprocessing the data and so focusing RDT/DB on interesting parts of the hypothesis space. Preprocessing achieved even results that were interesting in their own right. FDD found, for instance, that not all variants determine the corresponding type, but only the transmission type determines the transmission variant[5]. Other interesting results are those where the vehicle identification determines a particular vehicle part. Of course, we all know that a car has only one engine. However, for other parts (especially outfit or small technical parts) it is interesting to see, whether they are determined by the vehicle variant, or not.

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 5, 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

[5]As FDs do not allow one single value of the determining attribute to have two values in the determined attribute, the rule found by RDT/DB and shown above is not a FD.

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 (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).

The intervals found by NUM_INT in the cost attribute of warranty cases allowed RDT/DB to find 23 rules with an accuracy within the range of 79% to 84%.

Experiments on the data are an ongoing effort [6]. We are planning systematic tests that compare quality and time spent on learning for the various combinations of learning algorithms. 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 preprocessing. For instance, no relations with costs of warranty cases could be found before NUM_INT delivered the intervals. Moreover, without the further restrictions by learning results of other learning algorithms, RDT/DB could not use all the tuples. Hence, the advantage of applying the multistrategy 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. We also thank J.-U. Kietz for fruitful discussions.

References

Agrawal, R.; Mannila, H.; Srikant, R.; Toivonen, H.; and Verkamo, A. I. 1996. Fast discovery of association rules. In Fayyad, U. M.; Piatetsky-Shapiro, G.; Smyth, P.; and Uthurusamy, R., eds., *Advances in Knowledge Discovery and Data Mining*, AAAI Press Series in Computer Science. Cambridge Massachusetts, London England: A Bradford Book, The MIT Press. chapter 12, 277–296.

Beeri, C.; Dowd, M.; Fagin, R.; and Statman, R. 1984. On the structure of Armstrong relations for functional dependencies. *Journal of the ACM* 31(1):30–46.

Bell, S. 1995. Discovery and maintenance of functional dependencies by independencies. In *First Int. Conference on Knowledge Discovery in Databases.*

Brachman, R. J., and Anand, T. 1996. The process of knowledge discovery in databases: A human-centered approach. In Fayyad, U. M.; Piatetsky-Shapiro, G.; Smyth, P.; and Uthurusamy, R., eds., *Advances in Knowledge Discovery and Data Mining*, AAAI Press Series in Computer Science. Cambridge Massachusetts, London England: A Bradford Book, The MIT Press. chapter 2, 33–51.

Cai, Y.; Cercone, N.; and Han, J. 1991. Attribute-oriented induction in relational databases. In Piatetsky-Shapiro, G., and Frawley, W., eds., *Knowledge Discovery in Databases.* Cambridge, Mass.: AAAI/MIT Press. 213 - 228.

De Raedt, L., and Bruynooghe, M. 1993. A theory of clausal discovery. In Muggleton, S., ed., *Procs. of the 3rd International Workshop on Inductive Logic Programming*, number IJS-DP-6707 in J. Stefan Institute Technical Reports, 25–40.

De Raedt, L. 1992. *Interactive Theory Revision: an Inductive Logic Programming Approach.* Academic Press.

Flach, P. 1993. Predicate invention in inductive data engineering. In Brazdil, P., ed., *Machine Learning - ECML'93*, volume 667 of *Lecture Notes in Artificial Intelligence*, 83–94.

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

Kanellakis, P. 1990. *Elements of Relational Database Theory.* Formal Models and Semantics, Handbook of Theoretical Computer Science. Elsevier. chapter 12, 1074–1156.

Kietz, J.-U., and Wrobel, S. 1992. Controlling the complexity of learning in logic through syntactic and task-oriented models. In Muggleton, S., ed., *Inductive Logic Programming.* London: Academic Press. chapter 16, 335–360.

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

Kietz, J.-U. 1988. Incremental and reversible acquisition of taxonomies. In *Proceedings of EKAW-88*. chapter 24, 1–11. Also as KIT-Report 66, Technical University Berlin.

Kietz, J. U. 1996. *Induktive Analyse relationaler Daten*. Ph.D. Dissertation. to appear, in german.

Lavrac, N., and Dzeroski, S. 1994. *Inductive Logic Programming - Techniques and Applications*. New York: Ellis Horwood.

Mannila, H., and Räihä, K.-J. 1994. Algorithms for inferring functional dependencies from relations. *Data and Knowledge Engineering* 12:83–99.

Mannila, H., and Toivonen, H. 1996. On an algorithm for finding all interesting sentences. In Trappl, R., ed., *Cybernetics and Systems '96 (EMCSR 1996)*.

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

Michalski, R. 1994. Inferential theory of learning: Developing foundations for multistrategy learning. In Michalski, R., and Tecuci, G., eds., *Machine Learning A Multistrategy Approach (IV)*. San Francisco: Morgan Kaufmann.

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

Pazzani, M. J. 1995. An iterative improvement approach for the discretization of numeric attributes in Bayesian classifiers. In Fayyad, U. M., and Uthurusamy, R., eds., *The First International Conference on Knowledge Discovery and Data Mining*, 228–233. AAAI Press.

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

Savnik, I., and Flach, P. A. 1993. Bottom-up induction of functional dependencies from relations. In Piatetsky-Shapiro, G., ed., *Proceedings of the AAAI-93 Workshop on Knowledge Discovery in Databases*, 174–185. Menlo Park, California: The American Association for Artificial Intelligence.

Ullman, J. D. 1988. *Principles of database and knowledge–base systems*, volume 1. Rockville, MD: Computer Science Press.

Wettscherek, D., and Dietterich, T. G. 1995. An experimental comparison of the nearest-neighbour and nearest-hyperrectangle algorithms. *Machine Learning* 19(1):5 – 27.

Wrobel, S.; Wettscherek, D.; Sommer, E.; and Emde, W. 1996. Extensibility in data mining systems. submitted paper, available at http://nathan.gmd.de/projects/ml/home.html.