Enabling End-User Datawarehouse Mining
Contract No. IST-1999-11993
Deliverable No. D15

Mining Multi-Relational Data
Deliverable D15

Nico Brandt[1], Peter Brockhausen[2,3], Marc de Haas[1], Jörg-Uwe Kietz[2], Arno Knobbe[1], Olaf Rem[1], and Regina Zücker[2]

[1] Perot Systems
Dept. Knowledge Engineering
NL-3821 AE, Netherlands
{Nico.Brandt, Marc.DeHaas, Olaf.Rem}@ps.net

[2] Swiss Life
IT Research & Development
CH-8022 Zurich, Switzerland
{Uwe.Kietz, Peter.Brockhausen, Regina.Zuecker}@swisslife.ch

[3] University of Dortmund
Dept. of Computer Science
D-44221 Dortmund, Germany
{morik}@ls8.cs.uni-dortmund.de

June 22, 2001

Abstract

The fact that data is scattered over many tables causes many problems in the practice of data mining. To deal with this problem, one either constructs a single table by hand, or one uses a Multi-Relational Data Mining algorithm. In this work package some specific operations needed for dealing with multi-relational data have been investigated. One approach is that in which the single table is constructed automatically using aggregate functions, which repeatedly summarise information from different tables over associations in the data model. Following the construction of the single table, traditional data mining algorithms can be applied. The other approach is that in which a combination of description logic and first-order horn-logic is used.

Contents

Chapter 1

Introduction

An important practical problem in data mining is that we often want to find models and patterns over data that resides in multiple tables. Traditional attribute-value learning systems, such as C4.5 (Quinan 1993), can discover patterns that can be expressed in attribute-value languages which have the expressive power of propositional logic. These languages are limited and do not allow for representing complex structured objects and relations among objects or their components. In most relational data models these structured objects exists, and are interesting for learning purposes.

We investigated into several possibilities to handle learning in multi-relational databases.

ILP basically relies on joining of 1:N related tables, e.g.[1] join policies and partner based on foreign-keys in the partner-role, with the following positive and negative features:

+ very easy

- change the analyzed population

- propositionalization produces the difficult (NP-complete) multiple-instance learning problem.

- no overview features

- exploding number of records (in the depth of the relation chains involved.)

In chapter 3 we developed a preprocessing method based on aggregated features of the 1:N tables, e.g.: total yearly premium for all tariffs of all insurances, there exists a death-case-tariff, all insurances are pension-insurances, ...

+ unique join based on summarization

[1] The examples are based on the data base sample provided in [Mining Mart Deliverable 6.1a]

+ no change of the (distribution of the) analyzed population

+ propositionalization produces a normal attribute-value learning problem.

- computational more expensive (SQL: group by)

- danger of aggregating "apples" and "pears"

- no individual features and no information about co-occurrence of features on depended objects.

Another possibility is the specialization of the (indeterminate) 1:N relation into (determinate) 1:1 relations, e.g.: split partner-role: insurance-owner, insured person, premium-payer, ... order the tariffs of an policy based on function: death, saving, disabled, ...

+ unique joins

+ no change of the (distribution of the) analyzed population

+ propositionalization produces a normal attribute-value learning problem with only a polynomial (in the number of determinate relations) larger number of features (e.g. DINUS).

- computational more expensive

- not always possible to find meaningful determinate specializations

- exploding number of features with many unknowns

This has been executed as a manual preprocessing operation in preparing the determinate version of the ILP MESH-design dataset for GOLEM. The description logic learning system KLUSTER tried to automatize that by learning useful sub-roles. In chapter 4 we developed the learning theory to combine ILP and DL into a single (CARIN-) rule learning method.

In WP13 we will investigate the combination of the second and third method with clustering for segmentation:

1. specialize 1:N relations whenever this is natural

2. build structural prototypes from the data

3. segment the data based on these structural prototypes

4. join now determinate (1:1) relations directly

5. group the aggregations of still indeterminate relations

Chapter 2

Data preparation for MRDM

The topic of this chapter is data preparation for multi-relational data (MRD). We have developed a software demonstrator Tolkien with a GUI for creation and manipulation of data models (Section 2.1). For modeling on multi-relaitonal data, it is important to know about key and relationships, which are the glue that keeps the tables in the model together (section 2.2). The developed model will then be used as a basis for data mining. If one use ILP tools for learning on these models, most ofthem will not be able to learn on the whole data set, but only on a sample. Nevertheless, the results of data mining should be evaluated on the whole data. In section 2.1.2 we describe the rule tester, a software which evaluates ILP rules on a data bases directly, and wich uses the modeling results as input for the requiered mapping from concepts, roles, and features in M4 to predicates in logic (i.e. ILP).

2.1 Pre-processing on a relational data model

Within the data mining process considerable time is spend for pre-processing the data. Practical experiences have shown that the time spend on pre-processing can take from 50% up to 80% of the entire data mining process when using the traditional attribute value-learners. In the multi-relational approach the pre-processing step that transforms the available data in a propositional representation has become obsolete. This may save considerable time, or the focus of pre-processing may shift into adding/coding domain knowledge that can be used in the learning process. Usage tests where data mining consultants used the Tolkien software[1] for pre-processing data showed that taking a relational data model as basis for data mining algorithm speeds up the pre-processing task. The consultants also reported

[1] Tolkien is a software demonstrator for this work package. It consists of an implementation of the aggregation approach described in chapter 3 and a graphical user interface in which a user can do some simple manipulations on a data model like table and attribute selection and specification of relations between tables.

that they had a better overview of the pre-processing task, because feature selection and feature construction are specified on the concepts (tables) in which they conceptually fit.

2.1.1 Tolkien and M4

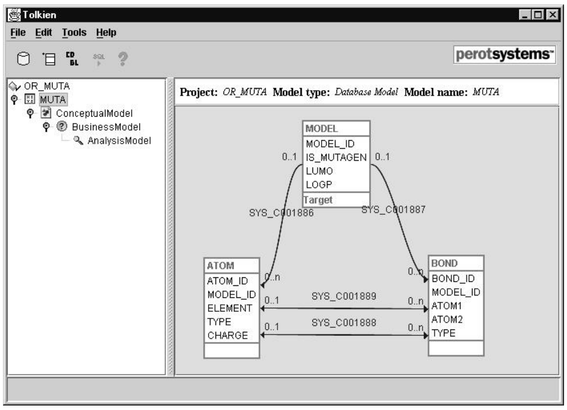

Figure 2.1: Tolkien screenshot showing a data model

Tolkien is a demonstrator program for this workpackage. The program provides the following functionality:

- An import function for importing (part of) a database schema. Thus a database model can easily be created.
- The creation and manipulation of data models. One can focus on parts of a data model by creating conceptual models, business models or analysis models. These are easily managed through a tree-like display of the models.
- A propositionalisation operator for automatic propositionalisation of multi-relational data. This operator generates a tree of pre-processing actions (see figure 2.2). The definitions (SQL code) of these actions can be viewed and the tree can be executed on the database.
- The generation and export of Common Declarative Bias Language code. This code describes the data model and can be used as input for an ILP engine.

Figure 2.1 shows a database model for the Mutagenesis database (see also chapter 3).

Chapter 3 describes how Tolkien has been used in a few mining experiments using the propositionalisation operator. The Tolkien user guide (see

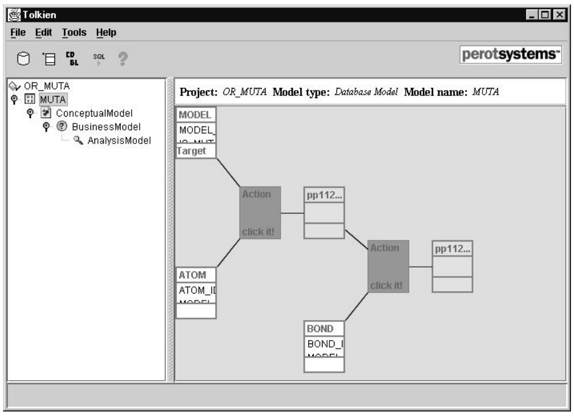

Figure 2.2: Pre-processing tree of actions for the propositionalisation operator

appendix) describes the functionality of the Tolkien demonstrator program in more detail. This section focuses further on the architecture of Tolkien.

When starting the development of this demonstrator program the implementation of the gui part of KDDSE was not started and the M4 model was not yet defined. Therefore the choice of implementation for Tolkien was based on experience and important trends in the market. The Tolkien demonstrator will be integrated with M4 by using M4 as a persistent store. In that way other programs that use M4 can benefit from the operations that Tolkien can do on the M4 model and vice versa. Integration of the demonstrator with KDDSE is an important issue on which discussions have allready started. We expect to solve this issue in an early stage of WP12 in which the graphical user interface for the specification of an entire knowledge discovery task is build.

Tolkien consists of a client part and a server part. The client has a graphical user interface and has been developed using the Java programming language. Java was chosen, because it allows for fast development of especially user interfaces and the developed programs can be run easily on various platforms.

The server side consists of Enterprise Java Beans (EJB) components running in Sun's reference implementation of their Java 2 Platform Enterprise Edition (J2EE) application server[2]. When using the Java programming language on the server side, Enterprise Java Beans have become the standard to use. EJB components provide good reusability, enable rapid development and can provide their services to different types of clients. In the last two years J2EE has become an increasingly popular platform to develop appli-

[2] At *http://java.sun.com/j2ee/j2sdkee/* Sun provides a reference implementation of J2EE that may be used freely under certain conditions.

cations for. The EJB components that have been developed for Tolkien conform to the EJB specifications. This ensures that these components can also be easily deployed to application servers of other vendors.

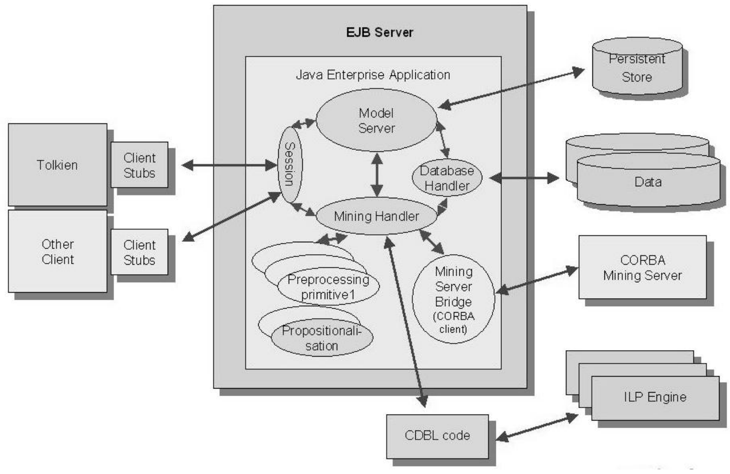

Figure 2.3: Architecture of Tolkien demonstrator program

Figure 2.3 shows a schematic overview of the architecture of Tolkien. The most important components and their relations are shown. Components that reside in the application server are:

- The Session component. It receives requests from clients and processes these itself or delegates them to the Model Server or the Mining Handler.

- The Model Server component. The Model Server stores meta data about data models. It has relations with the Session, Mining Handler, Database Handler and Persistent Store components. The current implementation depends mainly on container managed persistence for storing Model components. It can use an Oracle database or the Cloudscape database, which is provided with the J2EE application server.

- The Mining Handler component. The Mining Handler is responsible for the Mining process. It can invoke pre-processing primitives and communicate with the Session, Model Server, Database Handler or Mining Server Bridge components when needed. At this point the only task of the Mining Handler is to guide the propositionalisation operator and to generate CDBL code. The CDBL code describes the data model and can be used as input for an ILP engine.

- The Propositionalisation component. The Propositionalisation component handles the propositionalisation algorithm described in chapter 3. It is the only pre-processing primitive at this time in the demonstrator and is invoked by the Mining Handler.
- The Database Handler component. The Database Handler is responsible for processing requests for storing, retrieving or changing data in the data store. The Model Server and the Mining Handler use it. It supports connection pooling.

The figure also shows some possible extensions. Other type of clients than the Tolkien GUI may communicate with the Session component. This could for example also be a web client. Other types of pre-processing primitives are possible and can be added as components to the application server. Also CORBA can be used to connect to the EJB components. This makes it possible for example to have a Mining Server Bridge component that interacts between the Mining Handler and an external Mining Server.

The Tolkien demonstrator will be integrated with M4 by using M4 as a persistent store for the Model Server. In that way other programs that use M4 can benefit from the operations that Tolkien can do on the M4 model and vice versa.

2.1.2 Rule testing and M4

In the Mining Mart architecture, machine learning algorithms can be used as pre-processing operators. For multi-relational data sets, it is advantageous to use ILP algorithms, which are able to handle several relations at once. Whereas some ILP algorithms are able to learn on databases directly [Morik and Brockhausen, 1997], others are unable to make use of a relational database. Looking at new developments in ILP like CILGG, which provides a very rich and expressive representation language (a combination of horn logic with description logic, see chapter 4), it is rather unlikely that these kinds of algorithms will ever be able to learn on the whole database directly. Therefore in Mining Mart, we provide the possibility that machine learning algorithms can learn on a data sample, extracted from the database. But that means too, that we need a mechanism to test and evaluate the results of these algorithms on the whole database, which is the topic of this section.

In the development of the *rule tester*[3] we pursued two goals. First, this tool should be able to test rules in a restricted first–order logic, i.e. function free horn logic with explicit negation. Second, it should profit as much as possible from the SQL compiler from WP7. And in addition to the original work package description, we want to provision for the "generalisation" of the input language, i.e. from "pure" horn logic to horn logic with description logic enhancements.

[3] The description of the input formats will be in the appendix.

To fulfil our first goal, to test rules on a relational database, three kinds of input are required: obviously the rules to be tested, the "physical" names of the database tables and attributes involved, and a mapping from the predicates used in the rules onto the "physical" names. By "physical names", we mean the actual names of columns in tables, views, snapshots etc. in the database. We need this information for the generation of SQL code.

The case designer however, will not be confronted with the physical objects mentioned above. He deals with concepts, relations, and features. For a detailed description see deliverable D8/D9. Concepts and relations correspond to predicates in the logical representation used in ILP, feature values correspond to arguments in predicates.

Whatever pre-processing the user does on single tables, e.g. row or feature selection or construction, the result will be a "new" concept. This concept will be mapped one-to-one on a horn logic predicate and given a user specified name e.g. the name of the concept. By a one-to-one mapping we map a concept with n features on a n-ary predicate, where every argument place represents one feature in a defined order, and where the concept name is uniquely mapped on the predicate name. These predicates are the ones which appear in the rules, which are learned by ILP algorithms on excerpts of the data. Both kinds of information, the low level information about physical objects and the high level information about concepts, relations, and features is stored in M^4 and can be read automatically.

To reach our second goal and in contrast to RDT/DB [Morik and Brockhausen, 1997], where the user has to specify a mapping directly, here, we have chosen an approach where the user specifies the mapping rather indirectly by carrying out pre-processing on tables. This gives us a much larger flexibility and at the same time simplifies the mapping itself a lot. Since we have an optimising SQL compiler (WP7) which is responsible for the pre-processing, we may even get performance improvements in some situations, since the conditions in the SQL queries will be less complex. This in turn makes the task for the SQL optimiser in ORACLE easier.

As a side effect of this approach, we ease the workload for the case designer of a new Mining Mart case, since she can concentrate on the pre-processing and doesn't have to keep in mind the possible effects of many different mappings on the hypothesis space and performance.

2.2 Use of dependencies

For multi–relational pre-processing it is important to know about all the primary and foreign key relations in the database. These are the attributes, which are most often used for joining tables or building aggregates across several tables. But in practice, one encounters several problems. First, not all keys are marked in the data dictionary. This is due to the fact that most databases only allow one primary key per table, although several might be possible. For foreign keys, the referenced attribute must be marked as a key. Second, possible keys are not marked, because they might not hold in every instance of the database according to the defined schema. And third, some key relations might be unknown to the designers of the database schema.

To overcome these problems, one can use a tool like FDD [Bell and Brockhausen, 1995], which automatically detects all possible primary and foreign keys in a relational database directly via SQL queries. Discovered keys will be inserted in the M^4 model on the implementation level and relationships on the conceptual level.

The tool FDD is based on database theory about functional dependencies (FD) and unary inclusion dependencies[4]. The output is the most general cover of all existing functional dependencies and unary inclusion dependencies in the database. By restricting the inclusion dependencies to those dependencies, where keys are involved i.e. the attributes are part of FDs, one gets all foreign key relations.

Another use of functional dependencies is to structure the set of attributes of tables such that learning only regards attributes of one level of generality. Having computed the most general cover of one–place functional dependencies [Bell and Brockhausen, 1995], we define a hierarchy on the involved attributes in the following way:

Definition 2.1 (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.*

We give 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 affected attributes are of equal generality and more general or specific than attributes, which enter or leave this cycle, respectively.

This more general relationship on attributes can be exploited for the development of a sequence of hypothesis languages with different subsets of attributes. Given, for instance, the FDs $\{A \to B, B \to C, A \to C\}$, the first set of concepts in L_{H_1} includes C and neither A nor B, L_{H_2} includes B and neither A nor C, and L_{H_3} includes A and neither B nor C. This reduces

[4] The restriction to unary inclusion dependencies should be made, since the (general) implication problem of functional and inclusion dependencies is undecidable [Chandra and Vardi, 1985].

the number of concepts within each learning pass. As a result, we have a level–wise refinement strategy as e.g. in [Mannila and Toivonen, 1996], that means, start the sequence of learning experiments with hypotheses consisting of most general attributes. If this experiment is not successful according to the defined success criteria, i.e. the results are still too general, continue with more specific attributes only.

Chapter 3

Aggregation approach

The motivation for the use of aggregates stems from the observation that the difficult case in constructing a single table is when there are one-to-many relationships between tables. The traditional way to summarise such relationships in Statistics and OLAP is through aggregates that are based on histograms, such as *count*, *sum*, *min*, *max*, and *avg*. We limit ourselves to these aggregates, but note that they can be applied recursively over a collection of relationships. The idea of propositionalisation (the construction of one table) is not new. Several relatively successful algorithms have been proposed in the context of Inductive Logic Programming (ILP) (see [Kramer et al., 2000], [Kramer, 1999], [Srinivasan et al., 1999], [Kramer et al., 1998], [Alphonse and Toivonen, 1999], [Dehaspe et al., 1999]). A common aspect of these algorithms is that the derived table consists solely of binary features, each corresponding to a (promising) clause discovered by an ILP-algorithm. Especially for numerical attributes, our approach leads to a markedly different search space.

We illustrate our approach on three well-known data sets. The aim of these experiments is twofold. Firstly, to demonstrate the accuracy in a range of domains. Secondly, to illustrate the radically different way our approach models structured data, compared to ILP or MRDM approaches.

The paper is organised as follows. First we discuss propositionalisation and aggregates in more detail. In particular we introduce the notion of depth, to illustrate the complexity of the search space. Next we introduce the RollUp algorithm that constructs the single table. Then we present the results of our experiments and the paper ends with a discussion and conclusions.

3.1 Propositionalisation

In this section we describe the basic concepts involved in propositionalisation, and provide some definitions. In this paper, we define propositionalisation as the process of transforming a multi-relational dataset, containing structured examples, into a propositional dataset with derived attribute-value features, describing the structural properties of the examples. The process can thus be thought of as summarising data stored in multiple tables in a single table containing one record per example. The aim of this process, of course, is to pre-process multi-relational data for subsequent analysis by attribute-value learners.

We will be using this definition in the broadest sense. We will make no assumptions about the datatype of the derived attribute (binary, nominal, numeric, etc.) nor do we specify what language will be used to specify the propositional features. Traditionally, propositionalisation has been approached from an ILP standpoint with only binary features, expressed in first-order logic (FOL)(see [Kramer et al., 2000], [Kramer, 1999], [Kramer et al., 1998], [Alphonse and Toivonen, 1999], [Dehaspe et al., 1999]). To our knowledge, the use of other aggregates than existence has been limited. One example is given in [Deroski et al., 1999], which describes a propositionalisation-step where numeric attributes were defined for counts of different substructures. It is our aim to analyse the applicability of a broader range of aggregates.

With a growing availability of algorithms from the fields of ILP and Multi-Relational Data Mining (MRDM), one might wonder why such a cumbersome pre-processing step is desirable in the first place, instead of applying one of these algorithms to the multi-relational data directly. The following is a (possibly incomplete) list of reasons:

- Pragmatic choice for specific propositional techniques. People may wish to apply their favourite attribute-value learner, or only have access to commercial of-the-shelf Data Mining tools. Good examples can be found in the contributions to the financial dataset challenge at PKDD conferences [Workshop PKDD, 1999].

- Superiority of propositional algorithms with respect to certain Machine Learning parameters. Although extra facilities are quickly being added to existing ILP engines, propositional algorithms still have a head start where it concerns handling of numeric values, regression, distance measures, cumulativity etc.

- Greater speed of propositional algorithms. This advantage of course only holds if the preceding work for propositionalisation was limited, or performed only once and then reused during multiple attribute-value learning sessions.

- Advantages related to multiple consecutive learning steps. Because we are applying two learning steps, we are effectively combining two search strategies. The first step essentially transforms a multi-relational search space into a propositional one. The second step then uses these complex patterns to search deeper than either step could achieve when applied in isolation. This issue is investigated in more detail in the remainder of this section.

The term propositionalisation leads to some confusion because, although it pertains to the initial step of flattening a multi-relational database, it is often used to indicate the whole approach, including the subsequent propositional learning step. Because we are mostly interested in the two steps in unison, and for the sake of discussion, we introduce the following generic algorithm. The name is taken from Czech, and indicates a two-step dance.

Polka (DB D; DM M; int r, p)
  $P :=$ MRDM (D, M, r);
  $R :=$ PDM (P, p);

The algorithm takes a database D and data model M (acting as declarative bias), and first applies a Multi-Relational Data Mining algorithm MRDM. The resulting propositional features P are then fed to a propositional Data Mining algorithm PDM, producing result R. We use the integers r and p very informally to identify the extent of the multi-relational and propositional search, respectively.

In order to characterise more formally the extent of the search, we introduce three measures that are functions of the patterns that are considered. The values of the measures for the most complex patterns in the search space are then measures for the extent of the search algorithm. The definition is based on the graphical pattern language of Selection Graphs, introduced in [Knobbe et al., 1999], but can be re-written in terms of other languages such as FOL, relational algebra or SQL. We first repeat our basic definition of Selection Graphs.

definition A *selection graph* G is a pair (N, E), where N is a set of pairs (t, C), t is a table in the data model and C is a, possibly empty, set of conditions on attributes in t of type $t.a$ *operator* c; the operator is one of the usual selection operators, $=$, $>$, etc. E is a set of triples (p, q, a) called selection edges, where p and q are selection nodes and a is an association between $p.t$ and $q.t$ in the data model. The selection graph contains at least one node n_0 that corresponds to the target table t_0.

Now assume G is a Selection Graph.

definition variable-depth: $d_v(G)$ equals the length of the longest path in G.

definition clause-depth: $d_c(G)$ equals the sum of the number of nonroot nodes, edges and conditions in G.

definition variable-width: $w_v(G)$ equals the largest sum of the number of conditions and children per node, not including the root-node.

The intuition of these definitions is as follows. An algorithm searches *variable-deep*, if pieces of discovered substructure are refined by adding more substructure, resulting in chains of variables (edges in Selection Graphs). With each new variable, information from a new table is involved. An algorithm searches *clause-deep*, if it considers very specific patterns, regardless of the number of tables involved. Even propositional algorithms may produce clause-deep patterns that contain many conditions at the root-node and no other nodes. Rather than long chains of variables, *variable-wide* algorithms are concerned with the frequent reuse of a single variable. If information from a new table is included, it will be further refined by extra restrictions, either through conditions on this information, or through further substructure.

example The following Selection Graph, which refers to a 3-table database introduced in [Knobbe et al., 1999], identifies parents above 40 who have a child and bought a toy. The measures produce the following complexity characteristics:

$$d_v(G) = 1, d_c(G) = 5, w_v(G) = 0$$

The complexity measures can now be used to relate the search depth of Polka to the propositional and multi-relational algorithm it is made up of.

lemma 1 d_v (Polka) $=$ d_v (MRDM)
lemma 2 d_c (Polka) $=$ d_{vc} (MRDM) d_c (PDM)
lemma 3 w_v (Polka) $=$ w_v (MRDM)

Not surprisingly, the complexity of Polka depends largely on the complexity of the actual propositionalisation step. However, lemma 2 demonstrates that Polka considers very clause-deep patterns, in fact deeper than a multi-relational algorithm would consider in isolation. This is due to the combining of search spaces mentioned earlier. Later on we will examine the search restrictions that the use of aggregates have on the propositionalisation step and thus on Polka.

3.2 Aggregates

In the previous section we observed that an essential element of propositionalisation is the ability to summarise information distributed over several tables in the target table. We require functions that can reduce pieces of substructure to a single value, which describes some aspects of this substructure. Such functions are called aggregates. Having a set of well-chosen aggregates will allow us to describe the essence of the structural information over a wide variety of structures.

We define an aggregate as a function that takes as input a set of records in a database, related through the associations in the data model, and produces a single value as output. We will be using aggregates to project information stored in several tables on one of these tables, essentially adding virtual attributes to this table. In the case where the information is projected on the target table, and structural information belonging to an example is summarised as a new feature of that example, aggregates can be thought of as a form of feature construction.

Our broad definition includes aggregates of a great variety of complexity. An important aspect of the complexity of an aggregate is the number of (associations between) tables it involves. As each aggregate essentially considers a subset of the data model, we can use our 3 previously defined complexity-measures for data models to characterise aggregates. Specifically variable-depth is useful to classify aggregates. An aggregate of variable-depth 0 involves just one table, and is hence a case of propositional feature construction. Standard aggregates found in SQL (count, min, sum, etc.) have a variable-depth of 1, whereas variable-deeper aggregates represent some form of Multi-Relational pattern (benzene rings in molecules, etc.). Using this classification of variable-depth we give some examples to illustrate the range of possibilities.

$d_v(A) = 0$:

- Propositions (adult == (age $\geq$ 18))
- Arithmetic functions (area == widthlength)

$d_v(A) = 1$:

- Count, count with condition
- Count distinct
- Min, max, sum, avg
- Exists, exists with condition
- Select record (eldest son, first contract)

- Predominant value

$d_v(A) > 1$:

- Exists substructure
- Count substructure
- Conjunction of aggregates (maximum count of children)

Clearly the list of possible classes of aggregates is long, and the number of instances is infinite. In order to arrive at a practical and manageable solution for propositionalisation we will have to drastically limit the range of classes and instances. Apart from deterministic and heuristic rules to select good candidates, pragmatic limitations to a small set of aggregate classes are unavoidable. In this paper we have chosen to restrict ourselves to the classes available in SQL, and combinations thereof. The remainder of this paper further investigates the choice of instances.

3.3 Summarisation

We will be viewing the propositionalisation process as a series of steps in which information in one table is projected onto records in another table successively. Each association in the data model gives rise to one such step. The specifics of such a step, which we will refer to as summarisation, are the subject of this section.

Let us consider two tables P and Q, neither of which needs to be the target table, that are joined by an association A. By summarising over A, information can be added to P about the structural properties of A, as well as the data within Q. To summarise Q, a set of aggregates of variable-depth 1 are needed. For Data Mining purposes these aggregates need to be informative, i.e. they need to add discriminative power to P. If an aggregate produces a constant value, it fails to capture useful variation in the substructure provided by A and Q. This is regardless of whether P is the target table or a secondary table, which needs to be further summarised. Discriminative power of a (virtual) attribute is expressed by the entropy measure.

As was demonstrated before in [Knobbe et al., 1999], the multiplicity of association A influences the search space of multi-relational patterns involving A. The same is true for summarisation over A using aggregates. Our choice of aggregates depends on the multiplicity of A. In particular if we summarise Q over A only the multiplicity on the side of Q is relevant. This is because an association in general describes two relationships between the records in both tables, one for each direction. The following four options exist:

1 For every record in P there is but a single record in Q. This is basically a look-up over a foreign key relation and no aggregates are required. A simple join will add all non-key attributes of Q to P.

0..1 Similar to the 1 case, but now a look-up may fail because a record in P may not have a corresponding record in Q. An outer join is necessary, which fills in NULL values for missing records.

1..n For every record in P, there is at least one record in Q. Aggregates are required in order to capture the information in the group of records belonging to a single record in P.

0..n Similar to the 1..n case, but now the value of certain aggregates may be undefined due to empty groups. Special care will need to be taken to deal with the resulting NULL values.

Let us now consider the 1..n case in more detail. A imposes a grouping on the records in Q. For m records in P there will be m groups of records in Q. Because of the set-semantics of relational databases every group can be described by a collection of histograms or data-cubes. We can now view an aggregate instance as a function of one of these types of histograms. For example the predominant aggregate for an attribute $Q.a$ simply returns the value corresponding to the highest count in the histogram of $Q.a$. Note that m groups will produce m histograms and thus m values for one aggregate instance, one for each record in P. The notion of functions of histograms helps us to define relevant aggregate classes.

count The count aggregate is the most obvious aggregate through its direct relation to histograms. The most basic instance without conditions simply returns the single value in the 0-dimensional histogram. Adding a single condition requires a 1-dimensional histogram of the attribute involved in the condition. For example the number of sons in a family can be computed from a histogram of gender of that family. An attribute with a cardinality c will produce c aggregate instances of count with one condition. It is clear that the number of instances will explode if we allow even more conditions. As our final propositional dataset will then become impractically large we will have to restrict the number of instances. We will only consider counts with no condition and counts with one condition on nominal attributes. This implies that for the *count* aggregate $w_v \leq 1$.
There is some overlap in the patterns that can be expressed by using the count aggregate and those expressed in FOL. Testing for a count greater than zero obviously corresponds to existence. Testing for a count greater than some threshold t however, requires a clause-depth of $O(t^2)$ in FOL. With the less-than operator things become even worse for FOL representations as it requires the use of negation in a way that the language bias of

many ILP algorithms does not cater for. The use of the count aggregate is clearly more powerful in these respects.

example Selecting the set of molecules containing at least three carbon atoms requires one count aggregate, compared to the following FOL statement:

molecule(M, ...), atom(M, A, c, ...), atom(M, B, c, ...),
atom(M, C, c, ...), A $\neq$ B, A $\neq$ C, B $\neq$ C.

min and max The two obvious aggregates for numeric attributes, min and max, exhibit similar behaviour. Again there is a trivial way of computing min and max from the histogram; the smallest and largest value for which there is a non-zero count, respectively. The min and max aggregates support another type of constraint commonly used in FOL-based algorithms, existence with a numeric constraint. The following proposition describes the correspondence between the minimum and maximum of a group of numbers, and the occurrence of particular values in the group.

proposition Let B be a bag of real numbers, and t some real, then

max $(B) > t$ iff $\exists v \in B : v > t$,
min $(B) < t$ iff $\exists v \in B : v < t$.

This simply states that testing whether the maximum is greater than some threshold is equivalent to testing whether any value is greater than t. Analogous for min. It is important to note the combination of max and ¿, and min and ¡ respectively. If max were to be used in combination with ¡ or = then the FOL equivalent would again require the use of negation. Such use of the min and max aggregate gives us a natural means of introducing the universal quantor ”: all values are required to be above the minimum, or below the maximum. Another advantage of the min and max aggregate is that they each replace a set of binary existence aggregate instances (one for each threshold), making the propositional representation a lot more compact.

In short we can conclude that on the level of summarisation ($d_v = 1$) aggregates can express many of the concepts used in FOL. They can even express concepts that are hard or impossible to express in FOL. The most important limitation of our choice of aggregate instances is the number of attributes involved: $w_v \leq 1$.

3.4 The RollUp Algorithm

With the basic operations provided in the previous sections we can now define a basic propositionalisation algorithm. The algorithm will traverse the data model graph and repeatedly use the summarisation operation to project data from one table onto another, until all information has been aggregated at the target table. Although this repeated summarisation can be done in several ways, we will describe a basic algorithm, called RollUp.

The RollUp algorithm performs a depth-first search (DFS) through the data model, up to a specified depth. Whenever the recursive algorithm reaches its maximum depth or a leaf in the graph, it will "roll up" the relevant table by summarising it on the parent in the DFS tree. Internal nodes in the tree will again be summarised after all its children have been summarised. This means that attributes considered deep in the tree may be aggregated multiple times. The process continues until all tables are summarised on the target table. In combination with a propositional learner we have an instance of Polka. The following pseudo code describes RollUp more formally:

RollUp (Table T, Datamodel M, int d)
  $V := T$;
  if $d < 0$
    for all associations A from T in M
      $W :=$ RollUp(T.getTable(A), M, $d-1$);
      $S :=$ Summarise(W, A);
      V.add(S);
  return V;

The effect of RollUp is that each attribute appearing in a table other than the target table will appear several times in aggregated form in the resulting view. This multiple occurrence happens for two reasons. The first reason is that tables may occur multiple times in the DFS tree because they can be reached through multiple paths in the Datamodel. Each path will produce a different aggregation of the available attributes. The second reason is related to the choices of aggregate class at each summarisation along a path in the Datamodel. This choice, and the fact that aggregates may be combined in longer paths produces multiple occurrences of an attribute per path.

The variable-depth of the deepest feature is equal to the parameter d. Each feature corresponds to at most one attribute aggregated along a path of depth dv. The clause-depth is therefore a linear function of the variable-depth. As each feature involves at most one attribute, and is aggregated along a path with no branches, the variable-width will always be either 0 or 1. This produces the following characteristics for RollUp. Use lemmas 1 to 3 to characterise Polka instantiated with RollUp.

lemma 4 d_v (RollUp) $= d$
lemma 5 d_c (RollUp) $= 2d_v$ (RollUp) $+ 1$
lemma 6 w_v (RollUp) $= 1$

3.5 Experiments

In order to acquire empirical knowledge about the effectiveness of our approach, we have tested RollUp on three well-known multi-relational datasets. These datasets were chosen because they show a variety of datamodels that occur frequently in many multi-relational problems. They are Musk [Dietterich et al., 1997], Mutagenesis [Srinivasan et al., 1996], and Financial [Workshop PKDD, 1999]. Each dataset was loaded in the RDBMS Oracle. The data was modelled in UML using the multi-relational modelling tool Tolkien (see [Knobbe et al., 2000] and fig. 1) and subsequently translated to CDBL. Based on this declarative bias, the RollUp module produced one database view for each dataset, containing the propositionalised data. This was then taken as input for the common Machine Learning procedure C5.0.

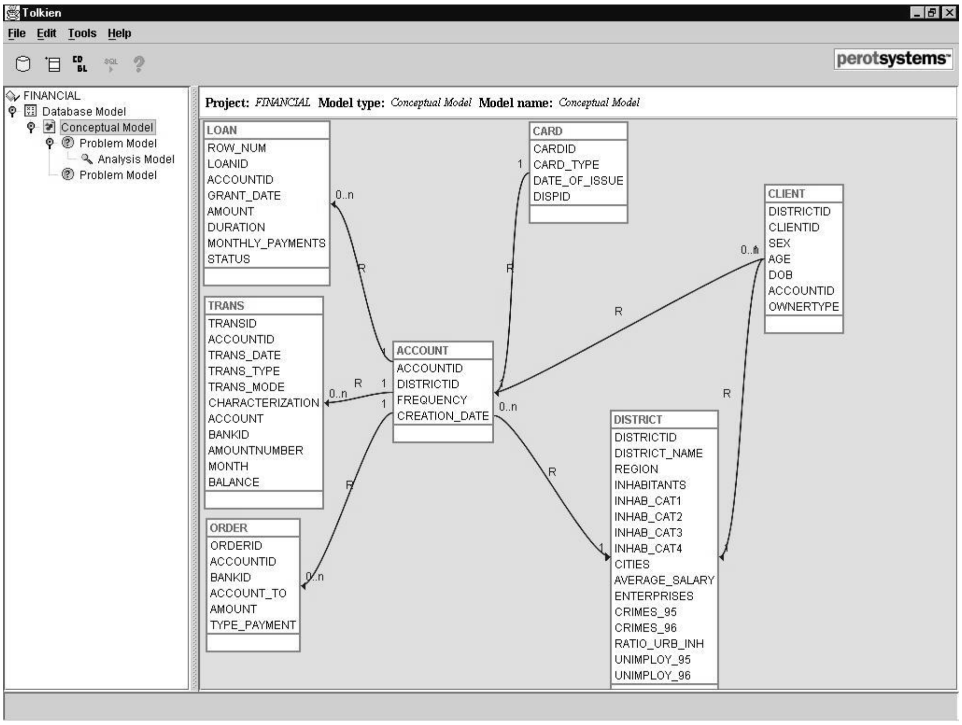

Figure 3.1: Screenshot of Tolkien

For quantitative comparison with other techniques, we have computed the average accuracy by leave-one-out cross-validation for Musk and Mutagenesis, and by 10-fold cross-validation for Financial. A single tree was built on the complete set of data, in order to acquire performance figures and qualitative results about our approach. Based on the occurrence of aggregate classes in the derived models, we have analysed the usefulness of different classes.

3.5.1 Musk

The Musk database [Dietterich et al., 1997] describes molecules occurring in different conformations. Each molecule is either *musk* or *non-musk* and one of the conformations determines this property. Such a problem is known as a multiple-instance problem, and will be modelled by two tables **molecule** and **conformation**, joined by a one-to-many association. **Confirmation** contains a molecule identifier plus 166 continuous features. **Molecule** just contains the identifier and the class. We have analysed two datasets, MuskSmall, containing 92 molecules and 476 confirmations, and MuskLarge, containing 102 molecules and 6598 confirmations.

Table 3.1 shows the results of RollUp compared to other, previously published results. The performance of RollUp is comparable to Tilde, but below that of special-purpose algorithms.

Algorithm	MuskSmall	MuskLarge
Iterated-discrim APR	92.4%	89.2%
GFS elim kde APR	91.3%	80.4%
RollUp	89.1%	77.5%
Tilde	87.0%	79.4%
Back-propagation	75.0%	67.7%
C4.5	68.5%	58.8%

Table 3.1: Results on Musk

3.5.2 Mutagenesis

Similar to the Musk database, the Mutagenesis database describes molecules falling in two classes, *mutagenic* and *non-mutagenic*. However, this time structural information about the atoms and bonds that make up the compound are provided. As chemical compounds are essentially annotated graphs, this database is a good test case for how well our approach deals with graph-data. The dataset we have analysed is known as the 'regression-friendly' dataset, and consists of 188 molecules. The database consists of 26 tables, of which three tables directly describe the graphical structure of the molecule (**molecule**, **atom** and **bond**). The remaining 23 tables describe the occurrence of predefined functional groups, such as benzene rings.

Four different experiments will be performed, using different settings, or so-called backgrounds. They will be referred to as experiment B1 to B4:

- B1: the atoms in the molecule are given, as well as the bonds between them; the type of each bond is given as well as the element and type of each atom.

- B2: as B1, but continuous values about the charge of atoms are added.

- B3: as B2, but two continuous values describing each molecule are added.
- B4: as B3, but knowledge about functional groups is added.

Table 3.2 shows the results of RollUp compared to other, previously published results. Clearly RollUp outperforms the other methods on all backgrounds, except B4. Most surprisingly, RollUp already performs well on B1, whereas the ILP methods seem to benefit from the propositional information provided in B3.

	Progol	FOIL	Tilde	RollUp
B1	76%	61%	75%	86%
B2	81%	61%	79%	85%
B3	83%	83%	85%	89%
B4	88%	82%	86%	84%

Table 3.2: Results on Mutagenesis

example The following tree of the B3 experiment illustrates the use of aggregates for structural descriptions.

```
CNT_BOND =< 26
    PREDOMINANT_TYPE_ATOM [21 27] -> F
    PREDOMINANT_TYPE_ATOM 22 -> F
    PREDOMINANT_TYPE_ATOM 3
        MAX_CHARGE_ATOM =< 0.0
            PREDOMINANT_TYPE_BOND 7 -> F
            PREDOMINANT_TYPE_BOND 1 -> T
        MAX_CHARGE_ATOM  > 0.0 -> F
CNT_BOND  > 26
    LUMO =< -1.102
        LOGP =< 6.26 -> T
        LOGP  > 6.26 -> F
    LUMO  > -1.102 -> F
```

3.5.3 Financial

Our third database is taken from the Discovery Challenge organised at PKDD '99 and PKDD 2000 [Workshop PKDD, 1999]. The database is based on data from a Czech bank. It describes the operations of 5369 clients holding 4500 accounts. The data is stored in 8 tables, 4 of which describe the usage of products, such as credit cards and loans. Three tables describe client and account information, and the remaining table contains demographic information about 77 Czech districts. We have chosen the account table as our target table (see fig. 1). Although we thus have 4500 examples, the dataset contains a total of 1079680 records. Our aim was to determine the loan-quality of an account.

A near perfect score of 99.9% was achieved on the Financial dataset. Due to the great variety of problem definitions described in the literature, quantitative comparisons with previous results are impossible. Similar (descriptive) analyses of loan-quality however never produced the pattern responsible for RollUp's performance. The aggregation approach proved particularly successful on the large transaction table (1056320 records). This table has sometimes been left out of other experiments due to scalability problems.

3.6 Discussion

The experimental results in the previous section demonstrate that our approach is at least competitive with existing multi-relational techniques, such as Progol and Tilde. Our approach has two major differences with these techniques, which may be the source of the good performance: the use of aggregates and the use of propositionalisation. Let us consider the contribution of each of these in turn.

Aggregates There is an essential difference in the way a group of records is characterised by FOL and by aggregates. FOL characterisations are based on the occurrence of one or more records in the group with certain properties. Aggregates on the other hand typically describe the group as a whole; each record has some influence on the value of the aggregate. The result of this difference is that FOL and aggregates provide two unique feature-spaces to the learning procedure. Each feature-space has its advantages and disadvantages, and may be more or less suitable for the problem at hand.

Although the feature-spaces produced by FOL and aggregates have entirely different characteristics, there is still some overlap. As was shown in section 4, some aggregates are very similar in behaviour to FOL expressions. The common features in the two spaces typically

- select one or a few records in a group (min and $<$, max and $>$, count > 0 for some condition).
- involve a single attribute: $w_v \leq 1$
- have a relatively low variable-depth.

If these properties hold, aggregate-based learning procedures will generally perform better, as they can dispose of the common selective aggregates, as well as the complete aggregates such as sum and avg.

Datamodels with a low variable depth are quite common in database design, and are called star-shaped (dv = 1) or snowflake schemata ($d_v > 1$). The Musk dataset is the most simple example of a star-shaped model. The Datamodel of Mutagenesis consists for a large part of a star-shaped model, and Financial is essentially a snowflake schema. Many real-world datasets described in the literature as ILP applications essentially have such a manageable structure. Moreover, results on these datasets frequently exhibit the extra condition of $w_v \leq 1$. Some illustrative examples are given in [Deroski et al., 1999] and [Todorovski et al., 1999].

example The following regression tree is taken (with kind permission from the authors) from ILP-experiments with biodegradability [Deroski et al., 1999], which involved a dataset very similar to the Mutagenesis dataset. Careful examination of the variable-structure shows that although very complex patterns are produced ($d_c = 13$), these patterns are neither variable-deep ($d_v = 1$) nor variable-wide ($w_v = 0$). Although the regression algorithm will have considered more elaborate patterns, the resulting tree completely falls into the set of patterns shared by ILP algorithms and our approach based on aggregates.

```
activ(A,B)
carbon_5_ar_ring(A,C,D) ?
+--yes: [9.10211]
+--no:  aldehyde(A,E,F) ?
  +--yes: [4.93332]
  +--no:  atm(A,G,h,H,I) ?
    +--yes: mweight(A,J) , J =< 80 ?
    |  +--yes: [5.52184]
    |  +--no:  ester(A,K,L) ?
    |    +--yes: mweight(A,M) , M =< 140 ?
    |    |  +--yes: [4.93332]
    |    |  +--no:  [5.88207]
    |    +--no:  mweight(A,N) , N =< 340 ?
    |       +--yes: carboxylic_acid(A,O,P) ?
    |       |  +--yes: [5.52288]
    |       |  +--no:  ar_halide(A,Q,R) ?
    |       |    +--yes: alkyl_halide(A,S,T) ?
    |       |    |  +--yes: [11.2742]
    |       |    |  +--no:  [7.81235]
    |       |    +--no:  phenol(A,U,V) ?
    |       |       +--yes: mweight(A,W) , W =< 180 ?
    |       |       |  +--yes: [4.66378]
    |       |       |  +--no:  [7.29547]
    |       |       +--no:  [6.86852]
```

```
|       +--no:  [8.28685]
+--no:  mweight(A,X) , X =< 100 ?
  +--yes: [6.04025]
  +--no:  [8.55286]
```

Propositionalisation According to lemma 2, Polka has the ability to discovery patterns that have a bigger clause-depth than either of its steps has. This is demonstrated by the experiments with our particular instance of Polka. RollUp produces variable-deep and thus clause-deep features. These clause-deep features are combined in the decision tree. Some leafs represent very clause-deep patterns, even though their support is still sufficient. This is an advantage of Polka (propositionalisation + propositional learning) over multi-relational algorithms in general.

Next to advantages related to expressivity, there are more practical reasons for using Polka. Once the propositionalisation-stage is finished, a large part of the computationally expensive work is done, and the derived view can be analysed multiple times. This not only provides a greater efficiency, but gives the analyst more flexibility in choosing the right modelling technique from a large range of well-developed commercially available set of tools. The analyst can vary the style of analysis (trees, rules, neural, instance-based) as well as the paradigm (classification, regression).

3.7 Conclusion

We have presented a method that uses aggregates to propositionalise a multi-relational database, such that the resulting view can be analysed by existing propositional methods. The method uses information from the datamodel to guide a process of repeated summarisation of tables on other tables. The method has shown good performance on three well-known datasets, both in terms of accuracy as well as in terms of speed and flexibility.

The experimental findings are supported by theoretical results, which indicate the strength of this approach on so-called star-shaped or snowflake Datamodels. We have also given evidence for why propositionalisation approaches in general may outperform ILP or MRDM systems, as was suggested before in the literature [Deroski et al., 1999], [Srinivasan et al., 1999].

Chapter 4

Learning CARIN-ALN rules with ILP-methods

4.1 Learning Carin-$\mathcal{ALN}$as a pre-processing method

Carin was proposed by [Levy and Rouset, 1998] as a combination of the two main approaches to represent and reason about relational knowledge, namely description logic and first-order horn-logic. In Inductive Logic Programming (ILP) learning first-order horn-logic is investigated in depth, for learning description logics there exist first approaches [Kietz and Morik, 1994; Cohen and Hirsh, 1994b] and theoretical learnability results [Cohen and Hirsh, 1994a; Frazier and Pitt, 1994].

Recently, it was proposed to use Carin-$\mathcal{ALN}$[1] as a framework for learning [Rouveirol and Ventos, 2000]. This is a very interesting extension of ILP as $\mathcal{ALN}$provides a new bias orthogonal to the one used in ILP, i.e. it allows all quantified descriptions of body-variables, instead of the existential quantified ones in ILP. This allows to handle the very difficult indeterminate relations efficiently by abstracting them into a determinate summary, i.e. in that respect it is a very similar to the aggregation operation described in the previous chapter. It also has atleast and atmost restrictions, which allow to quantify the amount of indeterminsm of these relations. In the paper provided in Appendix II we investigated into the polynomial learnability of Carin-$\mathcal{ALN}$and we defined a polynomial multi-relational pre-processing method which enables ILP-systems to learn rules with mixed existential and aggregated descriptions, or to learn pure $\mathcal{ALN}$-rules as determinate tree-structured clauses, a subset of horn logic, which can be learned by ILP-systems very efficiently [Kietz, 1996; Muggleton and Feng, 1992].

The description logic $\mathcal{ALN}$is also the base of the conceptual data model of M4 [Mining Mart Deliverable 7&8]. By being able to learn interesting $\mathcal{ALN}$concepts directly from the data base are able to discover interesting

[1] $\mathcal{ALN}$stands for a specific description logic described later.

parts (especially how indeterminate are the relations in the conceptual model and how are they restricted to a segment of the original concept) of the meta data from the data to be analysed. The work done here lays the complexity theoretic base to investigate into the discovery of structural prototypes in depth within the work-package WP13: "Clustering & Description Logic".

4.2 Testing CARIN rules on large databases

This section will present a short outline of the problems encountered, when testing CARIN rules on relational databases with the *rule tester* (cf. chapter 2.1.2). Let us first recapitulate testing of "pure" horn rules. The head of a horn rule is all quantified and the body of a horn rule existential quantified. Let us assume a one–to–one mapping from predicates onto tables. If we want to know if a rule is true (and how many examples are covered), it is sufficient to join all tables together. That means counting the number of tuples for those attributes, which correspond to argument places of the head predicate. This situation changes drastically, when it comes to testing of CARIN rules!

If we only have primitive concepts and roles, a DL rule is indistinguishable from a HL rule with unary and binary predicates. But using concept terms — DL terms which contain all, atleast, and atmost restrictions — enhances the expressive power of the language[2], i.e. the rules. All restrictions can be thought of as locally introducing all quantifications into the body of the rule. In a first try, we translated a rule which contains a role with atleast, atmost and all restrictions into three sub-queries, one for each restriction. The idea is to convert e.g. an *atleast 3* restriction into *not exists* ≤ 2, or *atmost 5* into *not exists* ≥ 6.

But, this approach turned out to be not scalable to large databases. The *rule tester* was developed for ILP systems relying on the k-literal restrictions to ensure polynomial learnability. To be scalable for data mining k should be in the range $3 - 8$. For the *rule tester* this means that one can just translate predicates to tables and variable-cooccurances to join-conditions, and the result will be a join of atmost k tables, i.e. a query modern RDBMS are able to perform efficiently.

Newer ILP-systems however rely on more general restrictions, which still ensure polynomial learnability, e.g. CILGG[Kietz, 1996] relies on the ij-determinate k-literal-local restriction [Kietz and Lübbe, 1994]. The complexity of learning k-literal-local clauses is similar to that of k-literal clauses, i.e. CILGGis scalable to data mining for a k in the range of $1 - 3$. But, a k-literal-local clause over p predicates with arity j can contain up to $k*(p*(j*k)^j)^k$ [Cohen, 1993] different literals. Translating that directly into

[2]Borgida showed that DL needs full first–order logic and is not expressible in HL [Borgida, 1996].

a join produces a query far beyond the capabilities of current RDBMS even for very small tables. Therefore, we plan to investigate into a structured translation approach as follows:

- Head and determinate literals (1-1 joins) are translated into one query.
- Each local will be translated into one query.
- each DL-literal will be translated into one query.
- The whole rule will be a PL/SQL script, where a cursor is running over the head/determinate query, and for each instance of the cursor truth of all the other queries will be tested.

This approach[3] is feasible due to several reasons. As pointed out in the paper (Kietz, 2001, see Appendix II), it makes only sense to learn rules with DL terms over deterministic variables. Thus, there is only one instantiation (of head and body predicates) for which we have to check the DL queries. Moreover, since we have to check truth and we will use cursors, we can always stop as soon as the first solution is found, i.e. we do not need to compute the result set as in a pure SQL solution.

[3]Testing HL rules will also benefit from this approach. Using cursors and doing a part of a otherwise large join "by hand", i.e. a PL/SQL script, will yield an even better performance.

Chapter 5

Conclusion and Outlook

In this workpackage two interesting pre-processing approaches have been presented for dealing with multi-relational data. There is a working software implementation of the aggregation approach that has been used to test its applicability and effectiveness. The results of the exepriments are promising. This propositionalisation operator uses meta data that is also available in the M4 model [Mining Mart Deliverable 7&8] and is suitable to incorporate in the KDDSE. The work in chapter 4, which enables ILP-systems to learn rules with mixed existential and aggregated descriptions, lays the complexity theoretic base to investigate into the discovery of structural prototypes.

Bibliography

[Alphonse and Toivonen, 1999] Alphonse, ., Rouveirol, C. Selective Propositionalisation for Relational Learning, In Proceedings of PKDD '99, 1999.

[Bell and Brockhausen, 1995] Bell, S. and P. Brockhausen: 1995, 'Discovery of Constraints and Data Dependencies in Databases (Extended Abstract)'. In: N. Lavrac and S. Wrobel (eds.): *Machine Learning: ECML-95 (Proc. European Conf. on Machine Learning, 1995).* Berlin, Heidelberg, New York, pp. 267 – 270, Springer Verlag.

[Borgida, 1996] Borgida, A.: 1996, 'On the Relative Expressiveness of Description Logic and Predicate Logic'. *Arificial Intelligence* **82**, 353–367.

[Cohen, 1993] Cohen, W. W.: 1993, 'Learnability of Restricted Logic Programs'. In: *ILP'93 Workshop*. Bled.

[Cohen and Hirsh, 1994a] Cohen, W. W. and H. Hirsh: 1994a, 'The Learnability of Description Logics with Equality Constraints'. *Machine Learning* **17**, 169–199.

[Chandra and Vardi, 1985] Chandra, A. K. and M. Y. Vardi: 1985, 'The Implication Problem for Functional and Inclusion Dependencies is Undecidable'. *SIAM Journal on Computing* **4**(3), 671–677.

[Cohen and Hirsh, 1994b] Cohen, W. W. and H. Hirsh: 1994b, 'Learning the CLASSIC Description Logic: Theoretical and Experimental Results'. In: *Proc. of the Int. Conf. on Knowledge Representation (KR94).*

[Dehaspe et al., 1999] Dehaspe, L., Toivonen, H., Discovery of frequent Datalog patterns, Data Mining and Knowledge Discovery 3(1), 1999

[Dietterich et al., 1997] Dietterich, T.G., Lathrop, R.H., Lozano-Prez, T., Solving the multiple-instance problem with axis-parallel rectangles, Artificial Intelligence, 89(1-2):31-71, 1997

[Deroski et al., 1999] Deroski, S., Blockeel, H., Kompare, B., Kramer, S., Pfahringer, B., Van Laer, W., Experiments in Predicting Biodegradability, In Proceedings of ILP '99, 1999

[Frazier and Pitt, 1994] Frazier, M. and L. Pitt: 1994, 'Classic Learning'. In: *Proc. of the 7th Annual ACM Conference on Computational Learning Theory*. pp. 23–34.

[Kietz, 1996] Kietz, J.-U.: 1996, 'Induktive Analyse Relationaler Daten'. Ph.D. thesis, Technical University Berlin. (in german).

[Kietz and Lübbe, 1994] Kietz, J.-U. and M. Lübbe: 1994, 'An Efficient Subsumption Algorithm for Inductive Logic Programming'. In: W. Cohen and H. Hirsh (eds.): *Proceedings of the 11th International Conference on Machine Learning IML–94*. San Francisco, CA, Morgan Kaufmann.

[Kietz and Morik, 1994] Kietz, J.-U. and K. Morik: 1994, 'A polynomial approach to the constructive Induction of Structural Knowledge'. *Machine Learning* **14**(2), 193–217.

[Knobbe et al., 1999] Knobbe, A.J., Blockeel, H., Siebes, A., Van der Wallen, D.M.G. Multi-Relational Data Mining, In Proceedings of Benelearn '99, 1999

[Knobbe et al., 2000] Knobbe, A.J., Siebes, A., Blockeel, H., Van der Wallen, D.M.G., Multi-Relational Data Mining, using UML for ILP, In Proceedings of PKDD 2000, 2000

[Kramer, 1999] Kramer, S., Relational Learning vs. Propositionalization, Ph.D thesis, 1999

[Kramer et al., 2000] Kramer, S., Frank, E., Bottom-Up Propositionalization, Work-in-Progress Reports, ILP 2000, 2000

[Kramer et al., 1998] Kramer, S., Pfahringer, B., Helma, C., Stochastic Propositionalization of non-determinate background knowledge, In Proceedings of ILP '98, 1998

[Levy and Rouset, 1998] Levy, A. Y. and M.-C. Rouset: 1998, 'Combining horn rules and description logic in CARIN'. *Artificial Intelligence* **104**, 165–209.

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

[Morik and Brockhausen, 1997] Morik, Katharina and Brockhausen, Peter, *A Multistrategy Approach to Relational Knowledge Discovery in Databases*, Machine Learning Journal volume 27, June 3 1997, pages 287–312, Kluwer, 1997.

[Muggleton and Feng, 1992] Muggleton, S. H. and C. Feng: 1992, 'Efficient induction of logic programs'. In: S. H. Muggleton (ed.): *Inductive Logic Programming*. Academic Press.

[Rouveirol and Ventos, 2000] Rouveirol, C. and V. Ventos: 2000, 'Towards learning in CARIN-$\mathcal{ALN}$'. In: J. Cussens and A. M. Frisch (eds.): *Proc. Tenth International Conference on Inductive Logic Programming, ILP'00*. Berlin, New York, Springer Verlag.

[Srinivasan et al., 1996] Srinivasan, A., King, R.D., Feature construction with ILP: a study of quantitative predictions of biological activity by structural attributes, In Proceedings of ILP '96, 1996

[Srinivasan et al., 1999] Srinivasan, A., King, R.D., Bristol, D.W., An Assessment of ILP-Assisted Models for Toxicology and the PTE-3 Experiment, In Proceedings of ILP '99, 1999

[Todorovski et al., 1999] Todorovski, L., Deroski, S., Experiments in Meta-level Learning with ILP, In Proceedings of PKDD '99, 1999

[Workshop PKDD, 1999] Workshop notes on Discovery Challenge PKDD '99, 1999

Appendix I PKDD paper

The paper version of chapter 3 has been submitted to the PKDD2001 and has been accepted.

Appendix II ILP paper

A 12-page version of the paper "A Data-Pre-processing Method Enabling ILP-Systems to Learn CARIN-$\mathcal{ALN}$ Rules" by Jörg-Uwe Kietz is submitted and currently under review at the 11th International Conference on Inductive Logic Programming (ILP'2001).

Appendix III Tolkien

The user manual of Tolkien is in pdf format and will be packaged with this deliverable.

Appendix IV Rule Tester

The software *rule tester* has to be called on the command line with two arguments:

`ruletester rule_file table_description_file`

table_description_file

In the `table_description_file`, the following commands may appear:

oracle_param

:- oracle_param(DatabaseIdentifier, User, Password, ListOfAdditionalUsers, ListofTableSpaces).

DatabaseIdentifier Here you have to enter the service name for the database. This must be a prolog atom!

User Here you have to enter the user name. This must be a prolog atom!

Password Here you have to enter the password. This must be a prolog atom!

ListOfAdditionalUsers Here you have to enter [].

ListofTableSpaces Here you have to enter [].

Example

:- oracle_param('esl69.dev.ch.swisslife',peter,peters-secret,[],[]).

rt_param

:- rt_param(LogFile,LevelOfDebugInformation).

LogFile prolog atom, log file will be written into $LOGDIR (if set) or current dir (if possible) or /tmp

LevelOfDebugInformation integer from 0 to 2. If set to 0, no debug infos, 1 = protocol into file, 2 = SQL statements (full infos).

Example

:- rt_param('dlruletester.log',2).

oracle_lexicon

:- oracle_lexicon(ListOfPredicates).

ListOfPredicates A Prolog list, containing the following predicates:

- oracle_db_table(TableId, TableName, Owner, TablespaceName, ListOfAttrIds)
- oracle_db_pred(AttrId, TableId, TableName, AttrName, DataType, Nullable, Owner)
- oracle_db_prim_keys(TableId, ListOfAttrIds)
- oracle_db_indexes(TableId, Uniqueness, ListOfAtrIds)
- oracle_data_types(DataType)

TableId Unique identifier for tables, prolog number.

TableName Prolog atom, name of physical oracle object.

Owner Prolog atom, name of physical oracle object.

TablespaceName Prolog atom, name of physical oracle object.

AttrId Unique identifier for attributes, prolog number.

AttrName Prolog atom, name of physical oracle object.

DataType Prolog atom, name of physical oracle object.

Nullable 'y' or 'n', prolog atom.

Uniqueness 'unique' or 'nonunique', prolog atom.

ListOf... Prolog list of the corresponding entries.

Example

:- oracle_lexicon([
oracle_db_table(1,hh_leader,dawami,dawami_d01,[2,1]),
oracle_db_table(2,hh_member,dawami,dawami_d01,[4,3]),
...
oracle_db_table(145,customer,dawami,dawami_d01,[1893, 1892, 1891, 1890, 1889, 1888, 1887, 1886, 1885, 1884]), ...
oracle_db_pred(1,1,hh_leader,hhid,number,n,dawami),
oracle_db_pred(2,1,hh_leader,bwvid,number,n,dawami),

```
oracle_db_pred(3,2,hh_member,hhid,number,n,dawami),
oracle_db_pred(4,2,hh_member,bwvid,number,n,dawami),
...
oracle_db_pred(1884,145,customer,epzivstd,number,y,dawami),
oracle_db_pred(1885,145,customer,epsexcd,number,y,dawami),
oracle_db_pred(1886,145,customer,eplnamecd,number,y,dawami),
oracle_db_pred(1887,145,customer,epsta,number,y,dawami),
oracle_db_pred(1888,145,customer,epgebudat,number,y,dawami),
oracle_db_pred(1889,145,customer,eperweart,number,y,dawami),
oracle_db_pred(1890,145,customer,epberuf,number,y,dawami),
oracle_db_pred(1891,145,customer,ptid,number,n,dawami),
oracle_db_pred(1892,145,customer,bwvwahr,number,y,dawami),
oracle_db_pred(1893,145,customer,bwvid,number,y,dawami),
...
oracle_db_prim_keys(145,[1891]),
...
oracle_db_indexes(145,unique,[1891]),
oracle_db_indexes(145,nonunique,[1893]),
...
oracle_data_types(number),
oracle_data_types(varchar2)]).
```

rt_lexicon

If the concept names are used for predicates, the following command must be used:

:- rt_lexicon(nil).

Otherwise,

:- rt_lexicon(ListOfPredicates).

ListOfPredicates A Prolog list, containing the following predicates:

- pred(Predname, Arity, [TableId, Prednamemapping], [[AttrId,Mappingid], ..., [AttrId,Mappingid]])

Predname Name of the predicate, prolog atom.

Arity Arity of the predicate, prolog number.

TableId see 5

Prednamemapping For future enhancements, any prolog atom.

AttrId see 5

Mappingid For future enhancements, any prolog atom.

Example

:- rt_lexicon([pred('Q',2,[8,rel],[[31,id],[30,id]]),
pred('R',2,[7,rel],[[29,id],[28,id]]),
pred('S',2,[5,rel],[[25,id],[24,id]]),
pred('P',2,[9,rel],[[33,id],[32,id]])]).

rule_file

In the `rule_file`, the following commands may appear, one commend per rule:

rule

:- rule(HLRule).
or
:- rule(DLRule).

HLRule Horn logic rule. Syntax:
body_predicate_1, ..., body_predicate_n −− > head_predicate.

DLRule Horn logic rule with description logic[1]. Syntax:
head_predicate – [body_term_1, ..., body_term_n]

head_predicate Ordinary prolog predicate.

body_predicate Ordinary prolog predicate or built in predicate: le(X,Y), lt(X,Y), ge(X,Y), gt(X,Y), eq(X,Y), ne(X,Y) with the meanings less equal, less than, greater equal, greater than, equal, not equal, respectivly.

body_term body_predicate or **dl_term**.

dl_term PrologVar:<ListOfConceptTermsOrRoleTerms

ListOfConceptTermsOrRoleTerms List of concept_term and/or role_term.

PrologVar Prolog variable, which appears in a body predicate, where this dl_term is about.

concept_term two–place body_predicate (no built–in predicate).

role_term three–place predicate role_name(Atleast, Atmost, Valuerestriction)

Atleast Prolog number.

[1] As this is preliminary, the syntax will be unified within WP13, when the additional requierements for clustering are clear.

Atmost Prolog number.

Valuerestriction ListOfConceptTermsOrRoleTerms or 'NIL'.

Example

```
rule('Q'(A, Y) , 'R'(B, X) , 'S'(B, Y) , le(X, Y) --> 'P'(A, X)).

rule(positive(A)
        - [
        hh_leader(A,B),
        hh_member(A,C),
        A:<[hh_leader(1,1,[belongs_to_hh(1,1,[]),
                                 customer(1,1,[])]),
                   hh_member(1,1,[belongs_to_hh(1,1,[]),
                                 customer(1,1,[])])],
        customer(B,D),
        customer(C,E),
        B:<[belongs_to_hh(1,1,[hh_leader(1,1,[]),
                                     hh_member(1,1,[])]),
                   customer(1,1,[])],
        C:<[belongs_to_hh(1,1,[hh_leader(1,1,[]),
                                     hh_member(1,1,[])]),
                   customer(1,1,[])],
        vn(F,D),
        vn(G,D),
        vn(H,D),
        vp(F,D),
        vp(G,D),
        vp(H,D),
        pz(F,D),
        pz(G,D),
        pz(H,D)]).
```