Multi-Relational Data Mining: An Introduction

Sašo Džeroski
Jožef Stefan Institute
Jamova 39, SI-1000 Ljubljana, Slovenia
saso.dzeroski@ijs.si

ABSTRACT

Data mining algorithms look for patterns in data. While most existing data mining approaches look for patterns in a single data table, multi-relational data mining (MRDM) approaches look for patterns that involve multiple tables (relations) from a relational database. In recent years, the most common types of patterns and approaches considered in data mining have been extended to the multi-relational case and MRDM now encompasses multi-relational (MR) association rule discovery, MR decision trees and MR distance-based methods, among others. MRDM approaches have been successfully applied to a number of problems in a variety of areas, most notably in the area of bioinformatics. This article provides a brief introduction to MRDM, while the remainder of this special issue treats in detail advanced research topics at the frontiers of MRDM.

Keywords

relational data mining, multi-relational data mining, inductive logic programming, relational association rules, relational decision trees, relational distance-based methods

1. IN A NUTSHELL

Data mining algorithms look for patterns in data. Most existing data mining approaches are propositional and look for patterns in a single data table. Relational data mining (RDM) approaches [16], many of which are based on inductive logic programming (ILP, [35]), look for patterns that involve multiple tables (relations) from a relational database. To emphasize this fact, RDM is often referred to as multi-relational data mining (MRDM, [21]). In this article, we will use the terms RDM and MRDM interchangeably. In this introductory section, we take a look at data, patterns, and algorithms in RDM, and mention some application areas.

1.1 Relational data

A relational database typically consists of several tables (relations) and not just one table. The example database in Table 1 has two relations: `Customer` and `MarriedTo`. Note that relations can be defined extensionally (by tables, as in our example) or intensionally through database views (as explicit logical rules). The latter typically represent relationships that can be inferred from other relationships. For example, having extensional representations of the relations mother and father, we can intensionally define the relations grandparent, grandmother, sibling, and ancestor, among others.

Intensional definitions of relations typically represent general knowledge about the domain of discourse. For example, if we have extensional relations listing the atoms that make a compound molecule and the bonds between them, functional groups of atoms can be defined intensionally. Such general knowledge is called domain knowledge or background knowledge.

Table 1: A relational database with two tables and two classification rules: a propositional and a relational.

Customer table

ID	Gender	Age	Income	TotalSpent	BigS
c1	Male	30	214000	18800	Yes
c2	Female	19	139000	15100	Yes
c3	Male	55	50000	12400	No
c4	Female	48	26000	8600	No
c5	Male	63	191000	28100	Yes
c6	Male	63	114000	20400	Yes
c7	Male	58	38000	11800	No
c8	Male	22	39000	5700	No
...	...	...	...	...	...

MarriedTo table

Spouse1	Spouse2
c1	c2
c2	c1
c3	c4
c4	c3
c5	c12
c6	c14
...	...

Propositional rule

IF Income > 108000 THEN BigSpender = Yes

Relational rule

`big_spender`(C1,Age1,Income1,TotalSpent1) ←
`married_to`(C1,C2) ∧
`customer`(C2,Age2,Income2,TotalSpent2,BS2) ∧
Income2 ≥ 108000.

1.2 Relational patterns

Relational patterns involve multiple relations from a relational database. They are typically stated in a more expressive language than patterns defined on a single data table. The major types of relational patterns extend the types of propositional patterns considered in single table data mining. We can thus have relational classification rules, relational regression trees, and relational association rules, among others.

An example relational classification rule is given in Table 1, which involves the relations `Customer` and `MarriedTo`. It predicts a person to be a big spender if the person is married to somebody with high income (compare this to the rule that states a person is a big spender if he has high income, listed above the relational rule). Note that the two persons C1 and C2 are connected through the relation `MarriedTo`.

Relational patterns are typically expressed in subsets of first-order logic (also called predicate or relational logic). Essentials of predicate logic include predicates (`MarriedTo`) and variables (`C1, C2`), which are not present in propositional logic. Relational patterns are thus more expressive than propositional ones.

Most commonly, the logic programming subset of first-order logic, which is strongly related to deductive databases, is used as the formalism for expressing relational patterns. E.g., the relational rule in Table 1 is a logic program clause. Note that a relation in a relational database corresponds to a predicate in first-order logic (and logic programming).

1.3 Relational to propositional

RDM tools can be applied directly to multi-relational data to find relational patterns that involve multiple relations. Most other data mining approaches assume that the data resides in a single table and require preprocessing to integrate data from multiple tables (e.g., through joins or aggregation) into a single table before they can be applied. Integrating data from multiple tables through joins or aggregation, however, can cause loss of meaning or information.

Suppose we are given the relations $customer(CustID, Name, Age, SpendsALot)$ and $purchase(CustID, ProductID, Date, Value, PaymentMode)$, where each customer can make multiple purchases, and we are interested in characterizing customers that spend a lot. Integrating the two relations via a natural join will give rise to a relation $purchase1$ where each row corresponds to a purchase and not to a customer. One possible aggregation would give rise to the relation $customer1(CustID, Age, NofPurchases, TotalValue, SpendsALot)$. In this case, however, some information has been clearly lost during aggregation.

The following pattern can be discovered if the relations *customer* and *purchase* are considered together.

$$customer(CID, Name, Age, yes) \leftarrow$$
$$Age > 30 \wedge$$
$$purchase(CID, PID, D, Value, PM) \wedge$$
$$PM = credit\_card \wedge Value > 100.$$

This pattern says: "a customer spends a lot if she is older than 30, has purchased a product of value more than 100 and paid for it by credit card." It would not be possible to induce such a pattern from either of the relations $purchase1$ and $customer1$ considered on their own.

Besides the ability to deal with data stored in multiple tables directly, RDM systems are usually able to take into account generally valid background (domain) knowledge given as a logic program. The ability to take into account background knowledge and the expressive power of the language of discovered patterns are distinctive for RDM.

Note that data mining approaches that find patterns in a given single table are referred to as attribute-value or propositional learning approaches, as the patterns they find can be expressed in propositional logic. RDM approaches are also referred to as first-order learning approaches, or relational learning approaches, as the patterns they find are expressed in the relational formalism of first-order logic. A more detailed discussion of the single table assumption, the problems resulting from it and how a relational representation alleviates these problems is given by Wrobel [50] (Chapter 4 of [16]).

1.4 Algorithms for relational data mining

A RDM algorithm searches a language of relational patterns to find patterns valid in a given database. The search algorithms used here are very similar to those used in single table data mining: one can search exhaustively or heuristically (greedy search, best-first search, etc.). Just as for the single table case, the space of patterns considered is typically lattice-structured and exploiting this structure is essential for achieving efficiency. The lattice structure is traversed by using refinement operators [46], which are more complicated in the relational case. In the propositional case, a refinement operator may add a condition to a rule antecedent or an item to an item set. In the relational case, a new relation can be introduced as well.

Just as many data mining algorithms come from the field of machine learning, many RDM algorithms come form the field of inductive logic programming (ILP, [35; 30]). Situated at the intersection of machine learning and logic programming, ILP has been concerned with finding patterns expressed as logic programs. Initially, ILP focussed on automated program synthesis from examples, formulated as a binary classification task. In recent years, however, the scope of ILP has broadened to cover the whole spectrum of data mining tasks (classification, regression, clustering, association analysis). The most common types of patterns have been extended to their relational versions (relational classification rules, relational regression trees, relational association rules) and so have the major data mining algorithms (decision tree induction, distance-based clustering and prediction, etc.).

Van Laer and De Raedt [49] (Chapter 10 of [16]) present a generic approach of upgrading single table data mining algorithms (propositional learners) to relational ones (first-order learners). Note that it is not trivial to extend a single table data mining algorithm to a relational one. Extending the key notions to, e.g., defining distance measures for multi-relational data requires considerable insight and creativity. Efficiency concerns are also very important, as it is often the case that even testing a given relational pattern for validity is computationally expensive, let alone searching a space of such patterns for valid ones. An alternative approach to RDM (called propositionalization) is to create a single table from a multi-relational database in a systematic fashion [28] (Chapter 11 of [16]): this approach shares some efficiency concerns and in addition can have limited expressiveness.

A pattern language typically contains a very large number of possible patterns even in the single table case: this number is in practice limited by setting some parameters (e.g., the largest size of frequent itemsets for association rule discovery). For relational pattern languages, the number of possible patterns is even larger and it becomes necessary to limit the space of possible patterns by providing more explicit constraints. These typically specify what relations should be involved in the patterns, how the relations can be interconnected, and what other syntactic constraints the patterns have to obey. The explicit specification of the pattern language (or constraints imposed upon it) is known under the name of declarative bias [38].

1.5 Applications of relational data mining

The use of RDM has enabled applications in areas rich with structured data and domain knowledge, which would be difficult to address with single table approaches. RDM has been used in different areas, ranging from analysis of business data, through environmental and traffic engineering to web mining, but has been especially successful in bioinformatics (including drug design and functional genomics). Bioinformatics applications of RDM are discussed in the article by Page and Craven in this issue. For a comprehensive survey of RDM applications we refer the reader to Džeroski [20] (Chapter 14 of [16]).

1.6 What's in this article

The remainder of this article first gives a brief introduction to inductive logic programming, which (from the viewpoint of MRDM) is mainly concerned with the induction of relational classification rules for two-class problems. It then proceeds to introduce the basic MRDM techniques of discovery of relational association rules, induction of relational decision trees and relational distance-based methods (that include both classification and clustering). The article concludes with an overview of the MRDM literature and Internet resources.

2. INDUCTIVE LOGIC PROGRAMMING

From a KDD perspective, we can say that inductive logic programming (ILP) is concerned with the development of techniques and tools for relational data mining. Patterns discovered by ILP systems are typically expressed as logic programs, an important subset of first-order (predicate) logic, also called relational logic. In this section, we first briefly discuss the language of logic programs, then proceed with a discussion of the major task of ILP and some approaches to solving it.

2.1 Logic programs and databases

Logic programs consist of clauses. We can think of clauses as first-order rules, where the conclusion part is termed the head and the condition part the body of the clause. The head and body of a clause consist of atoms, an atom being a predicate applied to some arguments, which are called terms. In Datalog, terms are variables and constants, while in general they may consist of function symbols applied to other terms. Ground clauses have no variables.

Consider the clause $father(X,Y) \vee mother(X,Y) \leftarrow parent(X,Y)$. It reads: "if X is a parent of Y then X is the father of Y or X is the mother of Y" ($\vee$ stands for logical or). $parent(X,Y)$ is the body of the clause and $father(X,Y) \vee mother(X,Y)$ is the head. *parent*, *father* and *mother* are predicates, X and Y are variables, and $parent(X,Y)$, $father(X,Y)$, $mother(X,Y)$ are atoms. We adopt the Prolog [4] syntax and start variable names with capital letters. Variables in clauses are implicitly universally quantified. The above clause thus stands for the logical formula $\forall X \forall Y : father(X,Y) \vee mother(X,Y) \vee \neg parent(X,Y)$. Clauses are also viewed as sets of literals, where a literal is an atom or its negation. The above clause is then the set $\{father(X,Y), mother(X,Y), \neg parent(X,Y)\}$.

Table 2: Database and logic programming terms.

DB terminology	LP terminology
relation name p	predicate symbol p
attribute of relation p	argument of predicate p
tuple $\langle a_1, \ldots, a_n \rangle$	ground fact $p(a_1, \ldots, a_n)$
relation p - a set of tuples	predicate p - defined extensionally by a set of ground facts
relation q defined as a view	predicate q defined intensionally by a set of rules (clauses)

As opposed to full clauses, definite clauses contain exactly one atom in the head. As compared to definite clauses, program clauses can also contain negated atoms in the body. While the clause in the paragraph above is a full clause, the clause $ancestor(X,Y) \leftarrow parent(Z,Y) \wedge ancestor(X,Z)$ is a definite clause ($\wedge$ stands for logical and). It is also a recursive clause, since it defines the relation *ancestor* in terms of itself and the relation *parent*. The clause $mother(X,Y) \leftarrow parent(X,Y) \wedge not\ male(X)$ is a program clause.

A set of clauses is called a clausal theory. Logic programs are sets of program clauses. A set of program clauses with the same predicate in the head is called a predicate definition. Most ILP approaches learn predicate definitions.

A predicate in logic programming corresponds to a relation in a relational database. A n-ary relation p is formally defined as a set of tuples [48], i.e., a subset of the Cartesian product of n domains $D_1 \times D_2 \times \ldots \times D_n$, where a domain (or a type) is a set of values. It is assumed that a relation is finite unless stated otherwise. A relational database (RDB) is a set of relations.

Thus, a predicate corresponds to a relation, and the arguments of a predicate correspond to the attributes of a relation. The major difference is that the attributes of a relation are typed (i.e., a domain is associated with each attribute). For example, in the relation $lives\_in(X,Y)$, we may want to specify that X is of type *person* and Y is of type *city*. Database clauses are typed program clauses.

A deductive database (DDB) is a set of database clauses. In deductive databases, relations can be defined extensionally as sets of tuples (as in RDBs) or intensionally as sets of database clauses. Database clauses use variables and function symbols in predicate arguments and the language of DDBs is substantially more expressive than the language of RDBs [31; 48]. A deductive Datalog database consists of definite database clauses with no function symbols.

Table 2 relates basic database and logic programming terms. For a full treatment of logic programming, RDBs, and deductive databases, we refer the reader to [31] and [48].

Table 3: A simple ILP problem: learning the *daughter* relation. Positive examples are denoted by $\oplus$ and negative by $\ominus$.

Training examples		Background knowledge	
$daughter(mary, ann).$	$\oplus$	$parent(ann, mary).$	$female(ann).$
$daughter(eve, tom).$	$\oplus$	$parent(ann, tom).$	$female(mary).$
$daughter(tom, ann).$	$\ominus$	$parent(tom, eve).$	$female(eve).$
$daughter(eve, ann).$	$\ominus$	$parent(tom, ian).$	

2.2 The ILP task of relational rule induction

Logic programming as a subset of first-order logic is mostly concerned with deductive inference. Inductive logic programming, on the other hand, is concerned with inductive inference. It generalizes from individual instances/observations in the presence of background knowledge, finding regularities/hypotheses about yet unseen instances.

The most commonly addressed task in ILP is the task of learning logical definitions of relations [42], where tuples that belong or do not belong to the target relation are given as examples. From training examples ILP then induces a logic program (predicate definition) corresponding to a view that defines the target relation in terms of other relations that are given as background knowledge. This classical ILP task is addressed, for instance, by the seminal MIS system [46] (rightfully considered as one of the most influential ancestors of ILP) and one of the best known ILP systems FOIL [42].

Given is a set of examples, i.e., tuples that belong to the target relation p (positive examples) and tuples that do not belong to p (negative examples). Given are also background relations (or background predicates) q_i that constitute the background knowledge and can be used in the learned definition of p. Finally, a hypothesis language, specifying syntactic restrictions on the definition of p is also given (either explicitly or implicitly). The task is to find a definition of the target relation p that is consistent and complete, i.e., explains all the positive and none of the negative tuples.

Formally, given is a set of examples $E = P \cup N$, where P contains positive and N negative examples, and background knowledge B. The task is to find a hypothesis H such that $\forall e \in P : B \wedge H \models e$ (H is complete) and $\forall e \in N : B \wedge H \not\models e$ (H is consistent), where $\models$ stands for logical implication or entailment. This setting, introduced by Muggleton [34], is thus also called learning from entailment. In an alternative setting proposed by De Raedt and Džeroski [15], the requirement that $B \wedge H \models e$ is replaced by the requirement that H be true in the minimal Herbrand model of $B \wedge e$: this setting is called learning from interpretations.

In the most general formulation, each e, as well as B and H can be a clausal theory. In practice, each e is most often a ground example (tuple), B is a relational database (which may or may not contain views) and H is a definite logic program. The semantic entailment ($\models$) is in practice replaced with syntactic entailment ($\vdash$) or provability, where the resolution inference rule (as implemented in Prolog) is most often used to prove examples from a hypothesis and the background knowledge. In learning from entailment, a positive fact is explained if it can be found among the answer substitutions for h produced by a query $? - b$ on database B, where $h \leftarrow b$ is a clause in H. In learning from interpretations, a clause $h \leftarrow b$ from H is true in the minimal Herbrand model of B if the query $b \wedge \neg h$ fails on B.

As an illustration, consider the task of defining relation $daughter(X, Y)$, which states that person X is a daughter of person Y, in terms of the background knowledge relations $female$ and $parent$. These relations are given in Table 3. There are two positive and two negative examples of the target relation $daughter$. In the hypothesis language of definite program clauses it is possible to formulate the following definition of the target relation,

$$daughter(X, Y) \leftarrow female(X), parent(Y, X).$$

which is consistent and complete with respect to the background knowledge and the training examples.

In general, depending on the background knowledge, the hypothesis language and the complexity of the target concept, the target predicate definition may consist of a set of clauses, such as

$$daughter(X, Y) \leftarrow female(X), mother(Y, X).$$
$$daughter(X, Y) \leftarrow female(X), father(Y, X).$$

if the relations $mother$ and $father$ were given in the background knowledge instead of the $parent$ relation.

The hypothesis language is typically a subset of the language of program clauses. As the complexity of learning grows with the expressiveness of the hypothesis language, restrictions have to be imposed on hypothesized clauses. Typical restrictions are the exclusion of recursion and restrictions on variables that appear in the body of the clause but not in its head (so-called new variables).

From a data mining perspective, the task described above is a binary classification task, where one of two classes is assigned to the examples (tuples): $\oplus$ (positive) or $\ominus$ (negative). Classification is one of the most commonly addressed tasks within the data mining community and includes approaches for rule induction. Rules can be generated from decision trees [43] or induced directly [33; 7].

ILP systems dealing with the classification task typically adopt the covering approach of rule induction systems. In a main loop, a covering algorithm constructs a set of clauses. Starting from an empty set of clauses, it constructs a clause explaining some of the positive examples, adds this clause to the hypothesis, and removes the positive examples explained. These steps are repeated until all positive examples have been explained (the hypothesis is complete).

In the inner loop of the covering algorithm, individual clauses are constructed by (heuristically) searching the space of possible clauses, structured by a specialization or generalization operator. Typically, search starts with a very general rule (clause with no conditions in the body), then proceeds to add literals (conditions) to this clause until it only covers (explains) positive examples (the clause is consistent).

When dealing with incomplete or noisy data, which is most often the case, the criteria of consistency and completeness are relaxed. Statistical criteria are typically used

instead. These are based on the number of positive and negative examples explained by the definition and the individual constituent clauses.

2.3 Structuring the space of clauses

Having described how to learn sets of clauses by using the covering algorithm for clause/rule set induction, let us now look at some of the mechanisms underlying single clause/rule induction. In order to search the space of relational rules (program clauses) systematically, it is useful to impose some structure upon it, e.g., an ordering. One such ordering is based on θ-subsumption, defined below.

A substitution $\theta = \{V_1/t_1, ..., V_n/t_n\}$ is an assignment of terms t_i to variables V_i. Applying a substitution θ to a term, atom, or clause F yields the instantiated term, atom, or clause $F\theta$ where all occurrences of the variables V_i are simultaneously replaced by the term t_i. Let c and c' be two program clauses. Clause c θ-subsumes c' if there exists a substitution θ, such that $c\theta \subseteq c'$ [41].

To illustrate the above notions, consider the clause $c = daughter(X,Y) \leftarrow parent(Y,X)$. Applying the substitution $\theta = \{X/mary, Y/ann\}$ to clause c yields

$$c\theta = daughter(mary, ann) \leftarrow parent(ann, mary).$$

Clauses can be viewed as sets of literals: the clausal notation $daughter(X,Y) \leftarrow parent(Y,X)$ thus stands for $\{daughter(X,Y), \overline{parent(Y,X)}\}$ where all variables are assumed to be universally quantified, overline denotes logical negation, and the commas denote disjunction. According to the definition, clause c θ-subsumes c' if there is a substitution θ that can be applied to c such that every literal in the resulting clause occurs in c'. Clause c θ-subsumes the clause

$$c' = daughter(X,Y) \leftarrow female(X), parent(Y,X)$$

under the empty substitution $\theta = \emptyset$, since $\{daughter(X,Y), \overline{parent(Y,X)}\}$ is a proper subset of $\{daughter(X,Y), \overline{female(X)}, \overline{parent(Y,X)}\}$. Furthermore, under the substitution $\theta = \{X/mary, Y/ann\}$, clause c θ-subsumes the clause $c' = daughter(mary, ann) \leftarrow$
$female(mary), parent(ann, mary), parent(ann, tom)$.
θ-subsumption introduces a syntactic notion of generality. Clause c is at least as general as clause c' ($c \leq c'$) if c θ-subsumes c'. Clause c is more general than c' ($c < c'$) if $c \leq c'$ holds and $c' \leq c$ does not. In this case, we say that c' is a specialization of c and c is a generalization of c'. If the clause c' is a specialization of c then c' is also called a refinement of c. The only clause refinements usually considered by ILP systems are the minimal (most general) specializations of the clause.

There are two important properties of θ-subsumption:

- If c θ-subsumes c' then c logically entails c', $c \models c'$. The reverse is not always true. As an example, [23] gives the following two clauses $c = list(cons(V,W)) \leftarrow list(W)$ and $c' = list(cons(X, cons(Y,Z))) \leftarrow list(Z)$. Given the empty list, c constructs lists of any given length, while c' constructs lists of even length only, and thus $c \models c'$. However, no substitution θ exists such that $c\theta = c'$, since θ should map W both to $cons(Y,Z)$ and to Z, which is impossible. Therefore, c does not θ-subsume c'.

- The relation $\leq$ introduces a lattice on the set of reduced clauses [41]. This means that any two reduced clauses have a least upper bound (*lub*) and a greatest lower bound (*glb*). Both the *lub* and the *glb* are unique up to equivalence (renaming of variables) under θ-subsumption. Reduced clauses are the minimal representatives of the equivalence classes of clauses defined by θ-subsumption. For example, the clauses $daughter(X,Y) \leftarrow parent(Y,X)$, $parent(W,V)$ and $daughter(X,Y) \leftarrow parent(Y,X)$ θ-subsume one another and are thus equivalent. The latter is reduced, while the former is not.

The second property of θ-subsumption leads to the following definition: The least general generalization (*lgg*) of two clauses c and c', denoted by $lgg(c,c')$, is the least upper bound of c and c' in the θ-subsumption lattice [41]. The rules for computing the *lgg* of two clauses are outlined later in this chapter.

Note that θ-subsumption and least general generalization are purely syntactic notions since they do not take into account any background knowledge. Their computation is therefore simple and easy to be implemented in an ILP system. The same holds for the notion of generality based on θ-subsumption. On the other hand, taking background knowledge into account would lead to the notion of semantic generality [39; 6], defined as follows: Clause c is at least as general as clause c' with respect to background theory B if $B \cup \{c\} \models c'$.

The syntactic, θ-subsumption based, generality is computationally more feasible. Namely, semantic generality is in general undecidable and does not introduce a lattice on a set of clauses. Because of these problems, syntactic generality is more frequently used in ILP systems.

θ-subsumption is important for inductive logic programming for the following reasons:

- As shown above, it provides a generality ordering for hypotheses, thus structuring the hypothesis space. It can be used to prune large parts of the search space.

- θ-subsumption provides the basis for the following important ILP techniques:
  - clause construction by top-down searching of refinement graphs,
  - bounding the search of refinement graphs from below by the bottom clause constructed as
    - the least general generalization of two (or more) training examples, relative to the given background knowledge [37], or
    - the most specific inverse resolvent of an example with respect to the given background knowledge [34].

2.4 Searching the space of clauses

Most ILP approaches search the hypothesis space of program clauses in a top-down manner, from general to specific hypotheses, using a θ-subsumption-based specialization operator. A specialization operator is usually called a refinement operator [46]. Given a hypothesis language $\mathcal{L}$, a refinement operator ρ maps a clause c to a set of clauses $\rho(c)$ which are specializations (refinements) of c: $\rho(c) = \{c' \mid c' \in \mathcal{L},\ c < c'\}$.

A refinement operator typically computes only the set of minimal (most general) specializations of a clause under θ-subsumption. It employs two basic syntactic operations:

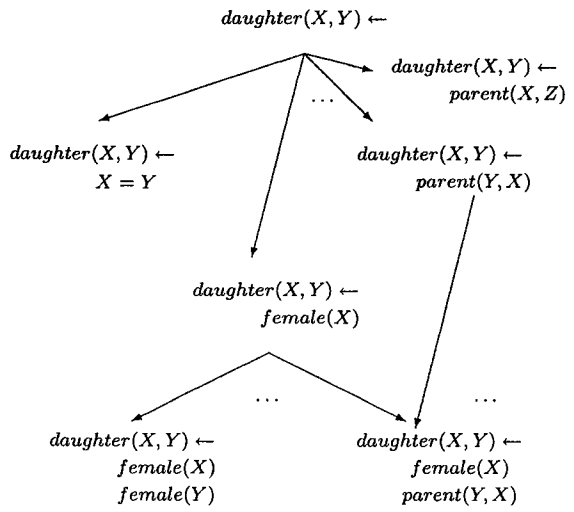

Figure 1: Part of the refinement graph for the family relations problem.

- apply a substitution to the clause, and
- add a literal to the body of the clause.

The hypothesis space of program clauses is a lattice, structured by the θ-subsumption generality ordering. In this lattice, a refinement graph can be defined as a directed, acyclic graph in which nodes are program clauses and arcs correspond to the basic refinement operations: substituting a variable with a term, and adding a literal to the body of a clause.

Figure 1 depicts a part of the refinement graph for the family relations problem defined in Table 3, where the task is to learn a definition of the *daughter* relation in terms of the relations *female* and *parent*.

At the top of the refinement graph (lattice) is the clause

$$c = daughter(X, Y) \leftarrow$$

where an empty body is written instead of the body *true*. The refinement operator ρ generates the refinements of c, which are of the form

$$\rho(c) = \{daughter(X, Y) \leftarrow L\}$$

where L is one of following literals:

- literals having as arguments the variables from the head of the clause: $X = Y$ (this corresponds to applying a substitution X/Y), $female(X)$, $female(Y)$, $parent(X, X)$, $parent(X, Y)$, $parent(Y, X)$, and $parent(Y, Y)$, and
- literals that introduce a new distinct variable Z ($Z \neq X$ and $Z \neq Y$) in the clause body: $parent(X, Z)$, $parent(Z, X)$, $parent(Y, Z)$, and $parent(Z, Y)$.

This assumes that the language is restricted to definite clauses, hence literals of the form *not* L are not considered, and nonrecursive clauses, hence literals with the predicate symbol *daughter* are not considered.

The search for a clause starts at the top of the lattice, with the clause $d(X, Y) \leftarrow$ that covers all example (positive and negative). Its refinements are then considered, then their refinements in turn, an this is repeated until a clause is found which covers only positive examples. In the example above, the clause $daughter(X, Y) \leftarrow female(X), parent(Y, X)$ is such a clause. Note that this clause can be reached in several ways from the top of the lattice, e.g., by first adding $female(X)$, then $parent(Y, X)$ or vice versa.

The refinement graph is typically searched heuristically level-wise, using heuristics based on the number of positive and negative examples covered by a clause. As the branching factor is very large, greedy search methods are typically applied which only consider a limited number of alternatives at each level. Hill-climbing considers only one best alternative at each level, while beam search considers n best alternatives, where n is the beam width. Occasionally, complete search is used, e.g., A^* best-first search or breadth-first search. This search can be bound from below by using so-called bottom clauses, which can be constructed by least general generalization [37] or inverse resolution/entailment [36].

2.5 Transforming ILP problems to propositional form

One of the early approaches to ILP, implemented in the ILP system LINUS [29], is based on the idea that the use of background knowledge can introduce new attributes for learning. The learning problem is transformed from relational to attribute-value form and solved by an attribute-value learner. An advantage of this approach is that data mining algorithms that work on a single table (and this is the majority of existing data mining algorithms) become applicable after the transformation.

This approach, however, is feasible only for a restricted class of ILP problems. Thus, the hypothesis language of LINUS is restricted to function-free program clauses which are typed (each variable is associated with a predetermined set of values), constrained (all variables in the body of a clause also appear in the head) and nonrecursive (the predicate symbol the head does not appear in any of the literals in the body).

The LINUS algorithm which solves ILP problems by transforming them into propositional form consists of the following three steps:

- The learning problem is transformed from relational to attribute-value form.
- The transformed learning problem is solved by an attribute-value learner.
- The induced hypothesis is transformed back into relational form.

The above algorithm allows for a variety of approaches developed for propositional problems, including noise-handling techniques in attribute-value algorithms, such as CN2 [8], to be used for learning relations. It is illustrated on the simple ILP problem of learning family relations. The task is to define the target relation $daughter(X, Y)$, which states that person X is a daughter of person Y, in terms of the background knowledge relations *female*, *male* and *parent*.

Table 4: Non-ground background knowledge for learning the *daughter* relation.

Training examples			Background knowledge	
$daughter(mary, ann)$.	$\oplus$	$parent(X,Y) \leftarrow$	$mother(ann, mary)$.	$female(ann)$.
$daughter(eve, tom)$.	$\oplus$	$mother(X,Y)$.	$mother(ann, tom)$.	$female(mary)$.
$daughter(tom, ann)$.	$\ominus$	$parent(X,Y) \leftarrow$	$father(tom, eve)$.	$female(eve)$.
$daughter(eve, ann)$.	$\ominus$	$father(X,Y)$.	$father(tom, ian)$.	

Table 5: Propositional form of the *daughter* relation problem.

	Variables		Propositional features							
C	X	Y	$f(X)$	$f(Y)$	$m(X)$	$m(Y)$	$p(X,X)$	$p(X,Y)$	$p(Y,X)$	$p(Y,Y)$
$\oplus$	$mary$	ann	$true$	$true$	$false$	$false$	$false$	$false$	$true$	$false$
$\oplus$	eve	tom	$true$	$false$	$false$	$true$	$false$	$false$	$true$	$false$
$\ominus$	tom	ann	$false$	$true$	$true$	$false$	$false$	$false$	$true$	$false$
$\ominus$	eve	ann	$true$	$true$	$false$	$false$	$false$	$false$	$false$	$false$

All the variables are of the type *person*, defined as *person* = {*ann*, *eve*, *ian*, *mary*, *tom*}. There are two positive and two negative examples of the target relation. The training examples and the relations from the background knowledge are given in Table 3. However, since the LINUS approach can use non-ground background knowledge, let us assume that the background knowledge from Table 4 is given.

The first step of the algorithm, i.e., the transformation of the ILP problem into attribute-value form, is performed as follows. The possible applications of the background predicates on the arguments of the target relation are determined, taking into account argument types. Each such application introduces a new attribute. In our example, all variables are of the same type *person*. The corresponding attribute-value learning problem is given in Table 5, where f stands for $female$, m for $male$ and p for $parent$. The attribute-value tuples are generalizations (relative to the given background knowledge) of the individual facts about the target relation.

In Table 5, *variables* stand for the arguments of the target relation, and *propositional features* denote the newly constructed attributes of the propositional learning task. When learning function-free clauses, only the new attributes (propositional features) are considered for learning.

In the second step, an attribute-value learning program induces the following if-then rule from the tuples in Table 5:
$Class = \oplus$ **if** $[female(X) = true] \wedge [parent(Y,X) = true]$

In the last step, the induced if-then rules are transformed into clauses. In our example, we get the following clause:
$daughter(X,Y) \leftarrow female(X), parent(Y,X)$.

The LINUS approach has been extended to handle determinate clauses [17; 30], which allow the introduction of determinate new variables (which have a unique value for each training example). There also exist a number of other approaches to propositionalization, some of them very recent: an overview is given by Kramer et al. [28].

Let us emphasize again, however, that it is in general not possible to transform an ILP problem into a propositional (attribute-value) form efficiently. De Raedt [13] treats the relation between attribute-value learning and ILP in detail, showing that propositionalization of some more complex ILP problems is possible, but results in attribute-value problems that are exponentially large. This has also been the main reason for the development of a variety of new RDM/ILP techniques by upgrading propositional approaches.

2.6 Upgrading propositional approaches

ILP/RDM algorithms have many things in common with propositional learning algorithms. In particular, they share the learning as search paradigm, i.e., they search for patterns valid in the given data. The key differences lie in the representation of data and patterns, refinement operators/generality relationships, and testing coverage (i.e., whether a rule explains an example).

Van Laer and De Raedt [49] explicitly formulate a recipe for upgrading propositional algorithms to deal with relational data and patterns. The key idea is to keep as much of the propositional algorithm as possible and upgrade only the key notions. For rule induction, the key notions are the refinement operator and coverage relationship. For distance-based approaches, the notion of distance is the key one. By carefully upgrading the key notions of a propositional algorithm, a RDM/ILP algorithm can be developed that has the original propositional algorithm as a special case.

The recipe has been followed (more or less exactly) to develop ILP systems for rule induction, well before it was formulated explicitly. The well known FOIL [42] system can be seen as an upgrade of the propositional rule induction program CN2 [8]. Another well known ILP system, PROGOL [36] can be viewed as upgrading the AQ approach [33] to rule induction.

More recently, the upgrading approach has been used to develop a number of RDM approaches that address data mining tasks other than binary classification. These include the discovery of frequent Datalog patterns and relational association rules [11] (Chapter 8 of [16]) [10], the induction of relational decision trees (structural classification and regression trees [27] and first-order logical decision trees [3]), and relational distance-based approaches to classification and clustering ([25], Chapter 9 of [16]; [22]). The algorithms developed have as special cases well known propositional algorithms, such as the APRIORI algorithm for finding frequent patterns; the CART and C4.5 algorithms for learning decision trees; k- nearest neighbor classification, hierarchical and k-medoids clustering. In the following three sections, we briefly review how each of the propositional approaches has been lifted to a relational framework, highlighting the key differences between the relational algorithms and their propositional counterparts.

3. RELATIONAL ASSOCIATION RULES

The discovery of frequent patterns and association rules is one of the most commonly studied tasks in data mining. Here we first describe frequent relational patterns (frequent Datalog patterns) and relational association rules (query extensions). We then look into how a well-known algorithm for finding frequent itemsets has been upgraded do discover frequent relational patterns.

3.1 Frequent Datalog queries and query extensions

Dehaspe and Toivonen [10], [11] (Chapter 8 of [16]) consider patterns in the form of Datalog queries, which reduce to SQL queries. A Datalog query has the form $?-A_1, A_2, \ldots A_n$, where the A_i's are logical atoms.

An example Datalog query is

$$?-person(X), parent(X,Y), hasPet(Y,Z)$$

This query on a Prolog database containing predicates *person*, *parent*, and *hasPet* is equivalent to the SQL query

```
SELECT PERSON.ID, PARENT.KID, HASPET.AID
FROM PERSON, PARENT, HASPET
WHERE PERSON.ID = PARENT.PID
AND PARENT.KID = HASPET.PID
```

on a database containing relations PERSON with argument ID, PARENT with arguments PID and KID, and HASPET with arguments PID and AID. This query finds triples (x, y, z), where child y of person x has pet z.

Datalog queries can be viewed as a relational version of itemsets (which are sets of items occurring together). Consider the itemset $\{person, parent, child, pet\}$. The market-basket interpretation of this pattern is that a person, a parent, a child, and a pet occur together. This is also partly the meaning of the above query. However, the variables X, Y, and Z add extra information: the person and the parent are the same, the parent and the child belong to the same family, and the pet belongs to the child. This illustrates the fact that queries are a more expressive variant of itemsets.

To discover frequent patterns, we need to have a notion of frequency. Given that we consider queries as patterns and that queries can have variables, it is not immediately obvious what the frequency of a given query is. This is resolved by specifying an additional parameter of the pattern discovery task, called the key. The key is an atom which has to be present in all queries considered during the discovery process. It determines what is actually counted. In the above query, if $person(X)$ is the key, we count persons, if $parent(X,Y)$ is the key, we count (parent,child) pairs, and if $hasPet(Y,Z)$ is the key, we count (owner,pet) pairs. This is described more precisely below.

Submitting a query $Q = ?-A_1, A_2, \ldots A_n$ with variables $\{X_1, \ldots X_m\}$ to a Datalog database $\mathbf{r}$ corresponds to asking whether a grounding substitution exists (which replaces each of the variables in Q with a constant), such that the conjunction $A_1, A_2, \ldots A_n$ holds in $\mathbf{r}$. The answer to the query produces answering substitutions $\theta = \{X_1/a_1, \ldots X_m/a_m\}$ such that $Q\theta$ succeeds. The set of all answering substitutions obtained by submitting a query Q to a Datalog database $\mathbf{r}$ is denoted $answerset(Q, \mathbf{r})$.

The absolute frequency of a query Q is the number of answer substitutions θ for the variables in the key atom for which the query $Q\theta$ succeeds in the given database, i.e., $a(Q, \mathbf{r}, key) = |\{\theta \in answerset(key, \mathbf{r}) | Q\theta \text{ succeds w.r.t. } \mathbf{r}\}|$. The relative frequency (support) can be calculated as $f(Q, \mathbf{r}, key) = a(Q, \mathbf{r}, key)/|\{\theta \in answerset(key, \mathbf{r})\}|$. Assuming the key is $person(X)$, the absolute frequency for our query involving parents, children and pets can be calculated by the following SQL statement:

```
SELECT count(distinct *)
FROM SELECT PERSON.ID
    FROM PERSON, PARENT, HASPET
    WHERE PERSON.ID = PARENT.PID
    AND PARENT.KID = HASPET.PID
```

Association rules have the form $A \rightarrow C$ and the intuitive market-basket interpretation "customers that buy A typically also buy C". Given A and C have supports f_A and f_C, respectively, the confidence of the association rule is defined to be $c_{A \rightarrow C} = f_C / f_A$. The task of association rule discovery is to find all association rules $A \rightarrow C$, where f_C and $c_{A \rightarrow C}$ exceed prespecified thresholds (minsup and minconf).

Association rules are typically obtained from frequent itemsets. Suppose we have two frequent itemsets A and C, such that $A \subset C$, where $C = A \cup B$. If the support of A is f_A and the support of C is f_C, we can derive an association rule $A \rightarrow B$, which has confidence f_C / f_A. Treating the arrow as implication, note that we can derive $A \rightarrow C$ from $A \rightarrow B$ ($A \rightarrow A$ and $A \rightarrow B$ implies $A \rightarrow A \cup B$, i.e., $A \rightarrow C$).

Relational association rules can be derived in a similar manner from frequent Datalog queries. From two frequent queries $Q_1 = ?-l_1, \ldots l_m$ and $Q_2 = ?-l_1, \ldots l_m, l_{m+1}, \ldots l_n$, where Q_2 θ-subsumes Q_1, we can derive a relational association rule $Q_1 \rightarrow Q_2$. Since Q_2 extends Q_1, such a relational association rules is named a query extension.

A query extension is thus an existentially quantified implication of the form $?-l_1, \ldots l_m \rightarrow ?-l_1, \ldots l_m, l_{m+1}, \ldots l_n$ (since variables in queries are existentially quantified). A shorthand notation for the above query extension is $?-l_1, \ldots l_m \rightsquigarrow l_{m+1}, \ldots l_n$. We call the query $?-l_1, \ldots l_m$ the body and the sub-query $l_{m+1}, \ldots l_n$ the head of the query extension. Note, however, that the head of the query extension does not correspond to its conclusion (which is $?-l_1, \ldots l_m, l_{m+1}, \ldots l_n$).

Assume the queries $Q_1 = ?-person(X), parent(X,Y)$ and $Q_2 = ?-person(X), parent(X,Y), hasPet(Y,Z)$ are frequent, with absolute frequencies of 40 and 30, respectively. The query extension $E = ?-person(X), parent(X,Y) \rightsquigarrow hasPet(Y,Z)$ can be considered a relational association rule with a support of 30 and confidence of $30/40 = 75\%$. Note the difference in meaning between the query extension E and two obvious, but incorrect, attempts at defining relational association rules. The clause $person(X), parent(X,Y) \rightarrow hasPet(Y,Z)$ (which stands for the logical formula $\forall XYZ : person(X) \wedge parent(X,Y) \rightarrow hasPet(Y,Z)$) would be interpreted as follows: "if a person has a child, then this child has a pet". The implication $?-person(X), parent(X,Y) \rightarrow ?-hasPet(Y,Z)$, which stands for $(\exists XY : person(X) \wedge parent(X,Y)) \rightarrow (\exists YZ : hasPet(Y,Z))$ is trivially true if at least one person in the database has a pet. The correct interpretation of the query extension E is: "if a person has a child, then this person also has a child that has a pet."

3.2 Discovering frequent queries: WARMR

The task of discovering frequent queries is addressed by the RDM system WARMR [10]. WARMR takes as input a database **r**, a frequency threshold *minfreq*, and declarative language bias $\mathcal{L}$. The latter specifies a *key* atom and input-output modes for predicates/relations, discussed below.

WARMR upgrades the well-known APRIORI algorithm for discovering frequent patterns, which performs levelwise search [2] through the lattice of itemsets. APRIORI starts with the empty set of items and at each level l considers sets of items of cardinality l. The key to the efficiency of APRIORI lies in the fact that a large frequent itemset can only be generated by adding an item to a frequent itemset. Candidates at level $l+1$ are thus generated by adding items to frequent itemsets obtained at level l. Further efficiency is achieved using the fact that all subsets of a frequent itemset have to be frequent: only candidates that pass this tests get their frequency to be determined by scanning the database.

In analogy to APRIORI, WARMR searches the lattice of Datalog queries for queries that are frequent in the given database **r**. In analogy to itemsets, a more complex (specific) frequent query Q_2 can only be generated from a simpler (more general) frequent query Q_1 (where Q_1 is more general than Q_2 if Q_1 θ-subsumes Q_2; see Section 2.3 for a definition of θ-subsumption). WARMR thus starts with the query $? - key$ at level 1 and generates candidates for frequent queries at level $l+1$ by refining (adding literals to) frequent queries obtained at level l.

Table 6: An example specification of declarative language bias settings for WARMR.

```
warmode_key(person(-)).
warmode(parent(+, -)).
warmode(hasPet(+, cat)).
warmode(hasPet(+, dog)).
warmode(hasPet(+, lizard)).
```

Suppose we are given a Prolog database containing the predicates *person*, *parent*, and *hasPet*, and the declarative bias in Table 6. The latter contains the key atom $parent(X)$ and input-output modes for the relations *parent* and *hasPet*. Input-output modes specify whether a variable argument of an atom in a query has to appear earlier in the query (+), must not (-) or may, but need not to (±). Input-output modes thus place constraints on how queries can be refined, i.e., what atoms may be added to a given query.

Given the above, WARMR starts the search of the refinement graph of queries at level 1 with the query $? - person(X)$. At level 2, the literals $parent(X,Y)$, $hasPet(X,cat)$, $hasPet(X,dog)$ and $hasPet(X,lizard)$ can be added to this query, yielding the candidate queries $?-person(X),parent(X,Y)$, $?-person(X),hasPet(X,cat)$, $? - person(X),hasPet(X,dog)$, and $? - person(X)$, $hasPet(X,lizard)$. Taking the first of the level 2 queries, we the following literals are added to obtain level 3 queries: $parent(Y,Z)$ (note that $parent(Y,X)$ cannot be added, because X already appears in the query being refined), $hasPet(Y,cat)$, $hasPet(Y,dog)$ and $hasPet(Y,lizard)$.

While all subsets of a frequent itemset must be frequent in APRIORI, not all sub-queries of a frequent query need be frequent queries in WARMR. Consider the query $? - person(X),parent(X,Y),hasPet(Y,cat)$ and assume it is frequent. The sub-query $? - person(X),hasPet(Y,cat)$ is not allowed, as it violates the declarative bias constraint that the first argument of *hasPet* has to appear earlier in the query. This causes some complications in pruning the generated candidates for frequent queries: WARMR keeps a list of infrequent queries and checks whether the generated candidates are subsumed by a query in this list. The WARMR algorithm is given in Table 7.

Table 7: The WARMR algorithm for discovering frequent Datalog queries.

Algorithm WARMR(**r**, $\mathcal{L}$, *key*, *minfreq*; Q)
Input: Database **r**; Declarative language bias $\mathcal{L}$ and *key* ; threshold *minfreq*;
Output: All queries $Q \in \mathcal{L}$ with frequency $\geq$ *minfreq*

1. Initialize level $d := 1$
2. Initialize the set of candidate queries $\mathcal{Q}_1 := \{?\text{- } key\}$
3. Initialize the set of (in)frequent queries $\mathcal{F} := \emptyset$; $\mathcal{I} := \emptyset$
4. While $\mathcal{Q}_d$ not empty
5. Find frequency of all queries $Q \in \mathcal{Q}_d$
6. Move those with frequency below *minfreq* to $\mathcal{I}$
7. Update $\mathcal{F} := \mathcal{F} \cup \mathcal{Q}_d$
8. Compute new candidates:
   $\mathcal{Q}_{d+1}$ = WARMRgen($\mathcal{L}$; $\mathcal{I}$; $\mathcal{F}$; $\mathcal{Q}_d$))
9. Increment d
10. Return $\mathcal{F}$

Function WARMRgen($\mathcal{L}$; $\mathcal{I}$; $\mathcal{F}$; $\mathcal{Q}_d$);

1. Initialize $\mathcal{Q}_{d+1} := \emptyset$
2. For each $Q_j \in \mathcal{Q}_d$, and for each refinement $Q'_j \in \mathcal{L}$ of Q_j:
   Add Q'_j to $\mathcal{Q}_{d+1}$, unless:
   (i) Q'_j is more specific than some query $\in \mathcal{I}$, or
   (ii) Q'_j is equivalent to some query $\in \mathcal{Q}_{d+1} \cup \mathcal{F}$
3. Return $\mathcal{Q}_{d+1}$

WARMR upgrades APRIORI to a multi-relational setting following the upgrading recipe (see Section 2.6). The major differences are in finding the frequency of queries (where we have to count answer substitutions for the key atom) and the candidate query generation (by using a refinement operator and declarative bias). WARMR has APRIORI as a special case: if we only have predicates of zero arity (with no arguments), which correspond to items, WARMR can be used to discover frequent itemsets.

More importantly, WARMR has as special cases a number of approaches that extend the discovery of frequent itemsets with, e.g., hierarchies on items [47], as well as approaches to discovering sequential patterns [1], including general episodes [32]. The individual approaches mentioned make use of the specific properties of the patterns considered (very limited use of variables) and are more efficient than WARMR for the particular tasks they address. The high expressive power of the language of patterns considered has its computational costs, but it also has the important advantage that a variety of different pattern types can be explored without any changes in the implementation.

WARMR can be (and has been) used to perform propositionalization, i.e., to transform MRDM problems to propositional (single table) form. WARMR is first used to discover frequent queries. In the propositional form, examples correspond to answer substitutions for the key atom and the binary attributes are the frequent queries discovered. An attribute is true for an example if the corresponding query succeeds for the corresponding answer substitution. This approach has been applied with considerable success to the tasks of predictive toxicology [9] and genome-wide prediction of protein functional class [24].

4. RELATIONAL DECISION TREES

Decision tree induction is one of the major approaches to data mining. Upgrading this approach to a relational setting has thus been of great importance. In this section, we first look into what relational decision trees are, i.e., how they are defined, then discuss how such trees can be induced from multi-relational data.

4.1 Relational Classification, Regression, and Model Trees

Without loss of generality, we can say the task of relational prediction is defined by a two-place target predicate $target(ExampleID, ClassVar)$, which has as arguments an example ID and the class variable, and a set of background knowledge predicates/relations. Depending on whether the class variable is discrete or continuous, we talk about relational classification or regression. Relational decision trees are one approach to solving this task.

An example relational decision tree is given in Figure 3. It predicts the maintenance action A that should be taken on machine M ($maintenance(M, A)$), based on parts the machine contains ($haspart(M, X)$), their condition ($worn(X)$) and ease of replacement ($irreplaceable(X)$). The target predicate here is $maintenance(M, A)$, the class variable is A, and background knowledge predicates are $haspart(M, X)$, $worn(X)$ and $irreplaceable(X)$.

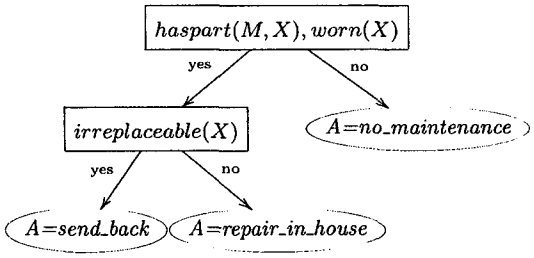

Figure 3: A relational decision tree, predicting the class variable A in the target predicate $maintenance(M, A)$.

Relational decision trees have much the same structure as propositional decision trees. Internal nodes contain tests, while leaves contain predictions for the class value. If the class variable is discrete/continuous, we talk about relational classification/regression trees. For regression, linear equations may be allowed in the leaves instead of constant class-value predictions: in this case we talk about relational model trees.

The tree in Figure 3 is a relational classification tree, while the tree in Figure 2 is a relational regression tree.

Table 8: A decision list representation of the relational decision tree in Figure 3.

$maintenance(M, A) \leftarrow haspart(M, X), worn(X),$
$\quad irreplaceable(X)$!, $A = send\_back$
$maintenance(M, A) \leftarrow haspart(M, X), worn(X)$, !,
$\quad A = repair\_in\_house$
$maintenance(M, A) \leftarrow A = no\_maintenance$

The latter predicts the degradation time (the logarithm of the mean half-life time in water [19]) of a chemical compound from its chemical structure, where the latter is represented by the atoms in the compound and the bonds between them. The target predicate is $degrades(C, LogHLT)$, the class variable $LogHLT$, and the background knowledge predicates are $atom(C, AtomID, Element)$ and $bond(C, A_1, A_2, BondType)$. The test at the root of the tree $atom(C, A1, cl)$ asks if the compound C has a chlorine atom $A1$ and the test along the left branch checks whether the chlorine atom $A1$ is connected to a nitrogen atom $A2$.

As can be seen from the above examples, the major difference between propositional and relational decision trees is in the tests that can appear in internal nodes. In the relational case, tests are queries, i.e., conjunctions of literals with existentially quantified variables, e.g., $atom(C, A1, cl)$ and $haspart(M, X), worn(X)$. Relational trees are binary: each internal node has a left (yes) and a right (no) branch. If the query succeeds, i.e., if there exists an answer substitution that makes it true, the yes branch is taken.

It is important to note that variables can be shared among nodes, i.e., a variable introduced in a node can be referred to in the left (yes) subtree of that node. For example, the X in $irreplaceable(X)$ refers to the machine part X introduced in the root node test $haspart(M, X), worn(X)$. Similarly, the $A1$ in $bond(C, A1, A2, BT)$ refers to the chlorine atom introduced in the root node $atom(C, A1, cl)$. One cannot refer to variables introduced in a node in the right (no) subtree of that node. For example, referring to the chlorine atom $A1$ in the right subtree of the tree in Figure 2 makes no sense, as going along the right (no) branch means that the compound contains no chlorine atoms.

The actual test that has to be executed in a node is the conjunction of the literals in the node itself and the literals on the path from the root of the tree to the node in question. For example, the test in the node $irreplaceable(X)$ in Figure 3 is actually $haspart(M, X), worn(X), irreplaceable(X)$. In other words, we need to send the machine back to the manufacturer for maintenance only if it has a part which is both worn and irreplaceable. Similarly, the test in the node $bond(C, A1, A2, BT), atom(C, A2, n)$ in Figure 2 is in fact $atom(C, A1, cl), bond(C, A1, A2, BT), atom(C, A2, n)$. As a consequence, one cannot transform relational decision trees to logic programs in the fashion "one clause per leaf" (unlike propositional decision trees, where a transformation "one rule per leaf" is possible).

Relational decision trees can be easily transformed into first-order decision lists, which are ordered sets of clauses (clauses in logic programs are unordered). When applying a decision list to an example, we always take the first clause that applies and return the answer produced. When applying a logic program, all applicable clauses are used and a set of answers can be produced. First-order decision lists can be represented by Prolog programs with cuts (!) [4]: cuts ensure that only the first applicable clause is used.

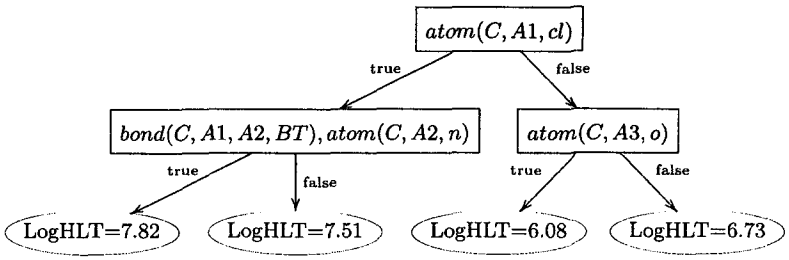

Figure 2: A relational regression tree for predicting the degradation time $LogHLT$ of a chemical compound C (target predicate $degrades(C, LogHLT)$).

Table 9: A decision list representation of the relational regression tree for predicting the biodegradability of a compound, given in Figure 2.

$degrades(C, LogHLT) \leftarrow atom(C, A1, cl),$
$\quad bond(C, A1, A2, BT), atom(C, A2, n), LogHLT = 7.82, !$
$degrades(C, LogHLT) \leftarrow atom(C, A1, cl),$
$\quad LogHLT = 7.51, !$
$degrades(C, LogHLT) \leftarrow atom(C, A3, o),$
$\quad LogHLT = 6.08, !$
$degrades(C, LogHLT) \leftarrow LogHLT = 6.73.$

Table 10: A logic program representation of the relational decision tree in Figure 3.

$a(M) \leftarrow haspart(M, X), worn(X), irreplaceable(X)$
$b(M) \leftarrow haspart(M, X), worn(X)$
$maintenance(M, A) \leftarrow not\ a(M), A = no\_maintenance$
$maintenance(M, A) \leftarrow b(M), A = repair\_in\_house$
$maintenance(M, A) \leftarrow a(M), not\ b(M), A = send\_back$

A decision list is produced by traversing the relational regression tree in a depth-first fashion, going down left branches first. At each leaf, a clause is output that contains the prediction of the leaf and all the conditions along the left (yes) branches leading to that leaf. A decision list obtained from the tree in Figure 3 is given in Table 8. For the first clause (*send_back*), the conditions in both internal nodes are output, as the left branches out of both nodes have been followed to reach the corresponding leaf. For the second clause, only the condition in the root is output: to reach the *repair_in_house* leaf, the left (yes) branch out of the root has been followed, but the right (no) branch out of the $irreplaceable(X)$ node has been followed. A decision list produced from the relational regression tree in Figure 2 is given in Table 9.

Generating a logic program from a relational decision tree is more complicated. It requires the introduction of new predicates. We will not describe the transformation process in detail, but rather give an example. A logic program, corresponding to the tree in Figure 3 is given in Table 10.

4.2 Induction of Relational Decision Trees

The two major algorithms for inducing relational decision trees are upgrades of the two most famous algorithms for inducting propositional decision trees. SCART [26; 27] is an upgrade of CART [5], while TILDE [3; 14] is an upgrade of C4.5 [43]. According to the upgrading recipe, both SCART and TILDE have their propositional counterparts as special cases. The actual algorithms thus closely follow CART and C4.5. Here we illustrate the differences between SCART and CART by looking at the TDIDT (top-down induction of decision trees) algorithm of SCART (Table 11).

Given a set of examples, the TDID algorithm first checks if a termination condition is satisfied, e.g., if all examples belong to the same class c. If yes, a leaf is constructed with an appropriate prediction, e.g., assigning the value c to the class variable. Otherwise a test is selected among the possible tests for the node at hand, examples are split into subsets according to the outcome of the test, and tree construction proceeds recursively on each of the subsets.

Table 11: The TDIDT part of the SCART algorithm for inducing relational decision trees.

```
procedure DivideAndConquer(TestsOnYesBranchesSofar, DeclarativeBias, Examples)

if TerminationCondition(Examples)
then
   NewLeaf = CreateNewLeaf(Examples)
   return NewLeaf
else
   PossibleTestsNow = GenerateTests(TestsOnYesBranchesSofar, DeclarativeBias)
   BestTest = FindBestTest(PossibleTestsNow, Examples)
   (Split1, Split2) = SplitExamples(Examples, TestsOnYesBranchesSofar, BestTest)
   LeftSubtree = DivideAndConquer(TestsOnYesBranchesSofar ∧ BestTest, Split1)
   RightSubtree = DivideAndConquer(TestsOnYesBranchesSofar, Split2)
   return [BestTest, LeftSubtree, RightSubtree]
```

A tree is thus constructed with the selected test at the root and the subtrees resulting from the recursive calls attached to the respective branches.

The major difference in comparison to the propositional case is in the possible tests that can be used in a node. While in CART these remain (more or less) the same regardless of where the node is in the tree (e.g., $A = v$ or $A < v$ for each attribute and attribute value), in SCART the set of possible tests crucially depend on the position of the node in the tree. In particular, it depends on the tests along the path from the root to the current node, more precisely the variables appearing in those tests and the declarative bias. To emphasize this, we can think of a GENERATETESTS procedure being separately employed before evaluating the tests. The inputs to this procedure are the tests on positive branches from the root to the current node and the declarative bias. These are also inputs to the top level TDIDT procedure.

The declarative bias in SCART contains statements of the form *schema(CofL, TandM)*, where *CofL* is a conjunction of literals and *TandM* is a list of type and mode declarations for the variables in those literals. Two such schema statements, used in the induction of the regression tree in Figure 2 are as follows: *schema((bond(V, W, X, Y), atom(V, X, Z)), [V:chemical:'+', W:atomid:'+', X:atomid:'-', Y:bondtype:'-', Z:element: '=']*) and *schema(bond(V, W, X, Y), [V:chemical:'+', W:atomid:'+', X:atomid:'-', Y:bondtype: '=']*). In the lists, each variable in the conjunction is followed by its type and mode declaration: '+' denotes that the variable must be bound (i.e., appear in *TestsOnYesBranchesSofar*), - that it must not be bound, and = that it must be replaced by a constant value.

Assuming we have taken the left branch out of the root in Figure 2, $TestsOnYesBranchesSofar = atom(C, A1, cl)$. Taking the declarative bias with the two schema statements above, the only choice for replacing the variables V and W in the schemata are the variables c and $A1$, respectively. The possible tests at this stage are thus of the form $bond(C, A1, A2, BT), atom(C, A2, E)$, where E is replaced with an element (such as *cl* - chlorine, *s* - sulphur, or *n* - nitrogen), or of the form $bond(C, A1, A2, BT)$, where BT is replaced with a bond type (such as *single*, *double*, or *aromatic*). Among the possible tests, the test $bond(C, A1, A2, BT), atom(C, A2, n)$ is chosen.

The approaches to relational decision tree induction are among the fastest MRDM approaches. They have been successfully applied to a number of practical problems. These include learning to predict the biodegradability of chemical compounds [19] and learning to predict the structure of diterpene compounds from their NMR spectra [18].

5. RELATIONAL DISTANCE-BASED APPROACHES

To upgrade distance-based approaches to learning, including prediction and clustering, it is necessary to upgrade the key notion of a distance measure from the propositional to the relational case. Such a measure could then be used within standard statistical approaches, such as nearest-neighbor prediction or hierarchical agglomerative clustering. In their system RIBL, Emde and Wettschereck [22] propose a relational distance measure. Below we first briefly discuss this measure, then outline how it has been used for relational classification and clustering [25].

5.1 The RIBL distance measure

Propositional distance measures are defined between examples that have the form of vectors of attribute values. They essentially sum up the differences between the examples' values along each of the dimensions of the vectors. Given two examples $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$, their distance might be calculated as

$$\text{distance}(x, y) = \sum_{i=1}^{n} \text{difference}(x_i, y_i)/n$$

where the difference between attribute values is defined as

$$\text{difference}(x_i, y_i) = \begin{cases} |x_i - y_i| & \text{if continuous} \\ 0 & \text{if discrete and } x_i = y_i \\ 1 & \text{otherwise} \end{cases}$$

In a relational representation, an example (also called instance or case) can be described by a set of facts about multiple relations. A fact of the target predicate of the form $target(ExampleID, A_1, ..., A_n)$ specifies an instance through its ID and properties, and additional information can be specified through background knowledge predicates. In Table 12, the target predicate *member(PersonID,A,G,I,MT)* specifies information on members of a particular club, which includes age, gender, income and membership type. The background predicates $car(OwnerID, CT, TS, M)$ and $house(OwnerID, DistrictID, Y, S)$ provide information on property owned by club members: for cars this includes car type, top speed and manufacturer, for houses the district, construction year and size. Additional information is available on districts through the predicate $district(DistrictID, P, S, C)$, i.e., the popularity, size, and country of the district.

Table 12: Two examples on which to study a relational distance measure.

member(*person1*, *45*, *male*, *20*, *gold*)
member(*person2*, *30*, *female*, *10*, *platinum*)
car(*person1*, *wagon*, *200*, *volkswagen*)
car(*person1*, *sedan*, *220*, *mercedesbenz*)
car(*person2*, *roadster*, *240*, *audi*)
car(*person2*, *coupe*, *260*, *bmw*)
house(*person1*, *murgle*, *1987*, *560*)
house(*person1*, *montecarlo*, *1990*, *210*)
house(*person2*, *murgle*, *1999*, *430*)
district(*montecarlo*, *famous*, *large*, *monaco*)
district(*murgle*, *famous*, *small*, *slovenia*)

The basic idea behind the RIBL [22] distance measure is as follows. To calculate the distance between two objects/examples, their properties are taken into account first (at depth 0). Next (at depth 1), objects immediately related to the two original objects are taken into account, or more precisely, the distances between the corresponding related objects. At depth 2, objects related to those at depth 1 are taken into account, and so on, until a user-specified depth limit is reached.

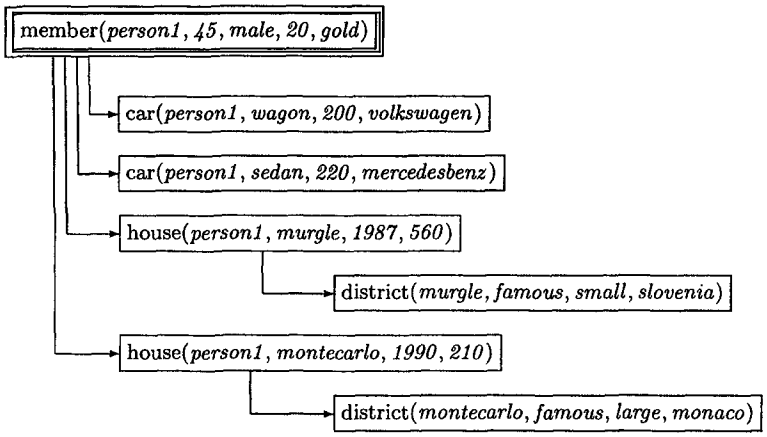

Figure 4: The case of (all facts related to) member(*person1*, *45*, *male*, *2000000*, *gold*) constructed with respect to the background knowledge in Table 12 and a depth limit of 2.

In our example, when calculating the distance between $e_1 = member(person1, 45, male, 20, gold)$ and $e_2 = member(person2, 30, female, 10, platinum)$, the properties of the persons (age, gender, income, membership type) are first compared and differences between them calculated and summed (as in the propositional case). At depth 1, cars and houses owned by the two persons are compared, i.e., distances between them are calculated. At depth 2, the districts where the houses reside are taken into account when calculating the distances between houses. Before beginning to calculate distances, RIBL collects all facts related to a person into a so-called case. The case for *person1* generated with a depth limit of 2 is given in Figure 4.

Let us calculate the distance between the two club members according to the distance measure. $d(e_1, e_2) = 1/5 \cdot (d(person1, person2) + d(45, 30) + d(male, female) + d(20, 10) + d(gold, platinum))$. With a depth limit of 0, the identifiers *person1* and *person2* are treated as discrete values, $d(person1, person2) = 1$ and we have $d(e_1, e_2) = (1 + (45 - 30)/100 + 1 + (20 - 10)/50 + 1)/5 = 0.67$; the denominators 100 and 50 denote the highest possible differences in age and income.

To calculate $d(person1, person2)$ at level 1, we collect the facts directly related to the two persons and partition them according to the predicates: we thus have $F_1, car =$ { car(*person1*, *wagon*, *200*, *volkswagen*), car(*person1*, *sedan*, *220*, *mercedesbenz*)} ; $F_2, car =$ { car(*person2*, *roadster*, *240*, *audi*), car(*person2*, *coupe*, *260*, *bmw*)}; $F_1, house =$ { house (*person1*, *murgle*, *1987*, *560*), house(*person1*, *montecarlo*, *1990*, *210*)}; and $F_2, house =$ { house(*person2*, *murgle*, *1999*, *430*)}. Then $d(person1, person2) = (d(F_1, car, F_2, car) + d(F_1, house, F_2, house))/2$. Distances between sets of facts are calculated as follows. We take the smaller set of facts (or the first, if they are of the same size): for $d(F_1, house, F_2, house)$, we take $F_2, house$. For each fact in this set, we calculate its distance to the nearest element of the other set, e.g., $F_1, house$, summing up these distances (the house of *person2* is closer to the house of *person1* in *murgle* then to the one in *montecarlo*). We add a penalty for the possible mismatch in cardinality and normalize with the cardinality of the larger set:

$d(F_1, house, F_2, house) =$
$[1 + min(d(house(person2, murgle, 1999, 430), house(person1, murgle, 1987, 560)), d(house(person2, murgle, 1999, 430), house(person1, montecarlo, 1990, 210))]/2 =$
$0.5 \cdot [1 + min((0 + (1999 - 1987)/100 + |430 - 560|/1000)/3, (1 + (1999 - 1990)/100 + (430 - 210)/1000)/3)]$
$= 0.5 + 0.5 \cdot min(0.25/3, 1.31/3) = 13/24.$

For calculating $d(F_1, car, F_2, car)$, we take F_1, car and note that both cars of *person1* are closer to the *audi* of *person2* than to the *bmw*. We thus have $d(F_1, car, F_2, car) = 0.5 \cdot [min_{c \in F_2, car} d(\text{car}(person1, wagon, 200, volkswagen), c) + min_{c \in F_2, car} d(\text{car}(person1, sedan, 220, mercedesbenz), c)]$
$= 0.5 \cdot [(1 + |200 - 240|/100 + 1)/3, (1 + |220 - 240|/100 + 1)/3] = 11/15$. Thus, at level 1, $d(person1, person2) = 0.5 \cdot (13/24 + 11/15) = 0.6375$ and $d(e_1, e_2) = (0.6375 + (45 - 30)/100 + 1 + (20 - 10)/50 + 1)/5 = 0.5975$.

Finally, at level 2, the distance between the two districts is taken into account when calculating $d(F_1, house, F_2, house)$. We have $d(murgle, montecarlo) = (0 + 1 + 1)/3 = 2/3$. However, since the house of *person2* is closer to the house of *person1* in *murgle* then to the one in *montecarlo*, the value of $d(F_1, house, F_2, house)$ does not change as it equals
$0.5 \cdot [1 + min((0 + (1999 - 1987)/100 + |430 - 560|/1000)/3, (2/3 + (1999 - 1990)/100 + (430 - 210)/1000)/3)]$
$= 0.5 + 0.5 \cdot min(0.25/3, (2/3 + 0.31)/3) = 13/24$. $d(e_1, e_2)$ is thus the same at level 1 and level 2 and is equal to 0.5975.

We should note here that the RIBL distance measure is not a metric [44]. However, some relational distance measures that are metrics have been proposed recently [45]. Designing distance measures for relational data is still a largely open and lively research area. Since distances and kernels are strongly related, this area is also related to designing kernels for structured data (Gaertner; this issue).

5.2 Relational distance-based learning

Once we have a relational distance measure, we can easily adapt classical statistical approaches to prediction and clustering, such as the nearest-neighbor method and hierarchical agglomerative clustering, to work on relational data. This is precisely what has been done with the RIBL distance measure.

The original RIBL [22] addresses the problem of prediction, more precisely classification. It uses the k-nearest neighbor (kNN) method in conjunction with the RIBL distance measure to solve the problem addressed. RIBL was successfully applied to the practical problem of diterpene structure elucidation [18], where it outperformed propositional approaches as well as a number of other relational approaches.

RIBL2 [25] upgrades the RIBL distance measure by considering lists and terms as elementary types, much like discrete and numeric values. Edit distances are used for these, while the RIBL distance measure is followed otherwise. RIBL2 has been used to predict mRNA signal structure and to automatically discover previously uncharacterized mRNA signal structure classes [25].

Two clustering approaches have been developed that use the RIBL distance measure [25]. RDBC uses hierarchical agglomerative clustering, while FORC adapts the k-means approach. The latter relies on finding cluster centers, which is easy for numeric vectors but far from trivial in the relational case. FORC thus uses the k-medoids method, which defines a cluster center as the existing case/example that has the smallest sum of squared distances to all other cases in the cluster and only uses distance information.

6. MRDM LITERATURE AND INTERNET RESOURCES

The book *Relational Data Mining*, edited by Džeroski and Lavrač [16] provides a cross-section of the state-of-the-art in this area at the turn of the millennium. This introductory article is largely based on material from that book.

The RDM book originated from the *International Summer School on Inductive Logic Programming and Knowledge Discovery in Databases* (ILP&KDD-97), held 15–17 September 1997 in Prague, Czech Republic, organized in conjunction with the *Seventh International Workshop on Inductive Logic Programming* (ILP-97). The teaching materials from this event are available on-line at `http://www-ai.ijs.si/SasoDzeroski/ILP2/ilpkdd/`. A summer school on relational data mining, covering both basic and advanced topics in this area, was organized in Helsinki in August 2002, preceding ECML/PKDD 2002 (The 13th European Conference on Machine Learning and The 6th European Conference on Principles and Practice of Knowledge Discovery in Databases). The slides are available online at `http://www-ai.ijs.si/SasoDzeroski/RDMSchool/`. A report on this event by Džeroski and Ženko is included in this special issue.

Recent developments in MRDM are presented at the annual workshop on Multi-Relational Data Mining. The first European event on this topic was held in 2001 (`http://mrdm.dantec.nl/`). Two international workshops were organized in conjunction with KDD-2002 and KDD-2003 (`http://www-ai.ijs.si/SasoDzeroski/MRDM2002/` and `MRDM2003/`). Of interest are the workshops on *Learning Statistical Models from Relational Data* held at AAAI-2000 (`http://robotics.stanford.edu/srl/workshop.html`) and IJCAI-2003 (`http://kdl.cs.umass.edu/events/srl2003/`) and the workshop on *Mining Graphs, Trees and Sequences*, held at ECML/PKDD-2003 (`http://www.ar.sanken.osaka-u.ac.jp/MGTS-2003CFP.html`).

The present issue is the first special issue devoted to the topic of multi-relational data mining. Two journal special issues address the related topic of using ILP for KDD: *Applied Artificial Intelligence* (vol. 12(5), 1998), and *Data Mining and Knowledge Discovery* (vol. 3(1), 1999).

Many papers related to MRDM appear in the ILP literature. For an overview of the ILP literature, see Chapter 3 of the RDM book [16]. ILP-related bibliographic information can be found at ILPnet2's on-line library (`http://www.cs.bris.ac.uk/~ILPnet2/Library/`).

The major publication venue for ILP-related papers is the annual ILP workshop. The first *International Workshop on Inductive Logic Programming (ILP-91)* was organized in 1991. Since 1996, the proceedings of the ILP workshops are published by Springer within the Lecture Notes in Artificial Intelligence/Lecture Notes in Computer Science series.

Papers on ILP appear regularly at major data mining, machine learning and artificial intelligence conferences. The same goes for a number of journals, including *Journal of Logic Programming*, *Machine Learning*, and *New Generation Computing*. Each of these has published several special issues on ILP. Special issues on ILP containing extended versions of selected papers from ILP workshops appear regularly in the *Machine Learning* journal.

Selected papers from the ILP-91 workshop appeared as a book *Inductive Logic Programming*, edited by Muggleton [35], while selected papers from ILP-95 appeared as a book *Advances in Inductive Logic Programming*, edited by De Raedt [12]. Authored books on ILP include *Inductive Logic Programming: Techniques and Applications* by Lavrač and Džeroski [30] and *Foundations of Inductive Logic Programming* by Nienhuys-Cheng and de Wolf [40]. The first provides a practically oriented introduction to ILP, but is dated now, given the fast development of ILP in the recent years. The other deals with ILP from a theoretical perspective.

Besides the Web sites mentioned so far, the ILPnet2 site @ IJS (`http://www-ai.ijs.si/ ilpnet2/`) is of special interest. It contains an overview of ILP related resources in several categories. These include a list of and pointers to ILP-related educational materials, ILP applications and datasets, as well as ILP systems. It also contains a list of ILP-related events and an electronic newsletter. For a detailed overview of ILP-related Web resources we refer the reader to Chapter 16 of the RDM book [16].

7. SUMMARY

As one of the position statements in this special issue states (Domingos; this issue), (multi-)relational data mining (MRDM) is a field whose time has come. The present article provides an entry point into this lively and exciting research area. Since many of the MRDM techniques around have their origin in logic-based approaches to learning and inductive logic programming (ILP), an introduction to ILP was given first. Three major approaches to MRDM were covered next: relational association rules, relational decision trees and relational distance-based approaches. Finally, a brief overview of the literature and Internet resources in this area were provided for those looking for more detailed information. While a variety of successful applications of MRDM exist, these are not covered in this article: we refer the reader to [20], as well as Page and Craven; Domingos; and Getoor (this issue).

The bulk of this special issue is devoted to hot topics and recent advances in MRDM. To some extent, hot topics in MRDM mirror hot topics in data mining and machine learning. These include ensemble methods (not covered here, see, e.g., Chapter 12 of [16]), kernel methods (Gaertner; this issue), probabilistic methods (De Raedt and Kersting; this issue), and scalability issues (Blockeel and Sebag; this issue). The same goes for application areas, with computational biology and bioinformatics being the most popular (Page and Craven; this issue). Web mining and link mining (link analysis/link discovery) follow suit (Domingos; this issue; Getoor; this issue).

Mining data that is richer in structure than a single table is rightfully attracting an ever increasing amount of research effort. It is important to realize that the formulation of multi-relational data mining is very general and has as special cases mining of data in the form of sequences, trees, and graphs. A survey article (Washio and Motoda; this issue) and a position statement (Holder and Cook; this issue) on mining data in the form of graphs are included in this special issue. Exploring representations that are richer than propositional and poorer than full first-order logic may be well worthwhile, because of the possibility to design more efficient algorithms.

In summary, the issue provide an introduction to the important and exciting research field of multi-relational data mining. It also outlines recent advances and interesting open questions. (The latter include, for example, the seamless integration of MRDM approaches within actual database management systems (DBMSs) and using the query optimization techniques of the DBMSs to improve the efficiency of MRDM approaches.) In this way, we hope to attract the attention of data mining researchers and stimulate new research and solutions to open problems in this area, which can have practical applications and far reaching implications for the entire field of data mining.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 3–14. IEEE Computer Society Press, Los Alamitos, CA, 1995.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, CA, 1996.

[3] H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101: 285–297, 1998.

[4] I. Bratko. *Prolog Programming for Artificial Intelligence*, 3rd edition. Addison-Wesley, Harlow, England, 2001.

[5] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth, Belmont, 1984.

[6] W. Buntine. Generalized subsumption and its applications to induction and redundancy. *Artificial Intelligence*, 36(2): 149–176, 1988.

[7] P. Clark and R. Boswell. Rule induction with CN2: Some recent improvements. In *Proceedings of the Fifth European Working Session on Learning*, pages 151–163. Springer, Berlin, 1991.

[8] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3(4): 261–283, 1989.

[9] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining*, pages 30–36. AAAI Press, Menlo Park, CA, 1998.

[10] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. *Data Mining and Knowledge Discovery*, 3(1): 7–36, 1999.

[11] L. Dehaspe and H. Toivonen. Discovery of Relational Association Rules. In [16], pages 189–212, 2001.

[12] L. De Raedt, editor. *Advances in Inductive Logic Programming.* IOS Press, Amsterdam, 1996.

[13] L. De Raedt. Attribute-value learning versus inductive logic programming: the missing links (extended abstract). In *Proceedings of the Eighth International Conference on Inductive Logic Programming*, pages 1–8. Springer, Berlin, 1998.

[14] L. De Raedt, H. Blockeel, L. Dehaspe, and W. Van Laer. Three Companions for Data Mining in First Order Logic. In [16], pages 105–139, 2001.

[15] L. De Raedt and S. Džeroski. First order jk-clausal theories are PAC-learnable. *Artificial Intelligence*, 70: 375–392, 1994.

[16] S. Džeroski and N. Lavrač, editors. *Relational Data Mining.* Springer, Berlin, 2001.

[17] S. Džeroski, S. Muggleton, and S. Russell. PAC-learnability of determinate logic programs. In *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*, pages 128–135. ACM Press, New York, 1992.

[18] S. Džeroski, S. Schulze-Kremer, K. Heidtke, K. Siems, D. Wettschereck, and H. Blockeel. Diterpene structure elucidation from ^{13}C NMR spectra with Inductive Logic Programming. *Applied Artificial Intelligence*, 12: 363–383, 1998.

[19] S. Džeroski, H. Blockeel, B. Kompare, S. Kramer, B. Pfahringer, and W. Van Laer. Experiments in Predicting Biodegradability. In *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, pages 80–91. Springer, Berlin, 1999.

[20] S. Džeroski. Relational Data Mining Applications: An Overview. In [16], pages 339–364, 2001.

[21] S. Džeroski, L. De Raedt, and S. Wrobel, editors. *Proceedings of the First International Workshop on Multi-Relational Data Mining.* KDD-2002: Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Edmonton, Canada, 2002.

[22] W. Emde and D. Wettschereck. Relational instance-based learning. In *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 122–130. Morgan Kaufmann, San Mateo, CA, 1996.

[23] P. Flach. Logical approaches to machine learning - an overview. *THINK*, 1(2): 25–36, 1992.

[24] R.D. King, A. Karwath, A. Clare, and L. Dehaspe. Genome scale prediction of protein functional class from sequence using data mining. In *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining*, pages 384–389. ACM Press, New York, 2000.

[25] M. Kirsten, S. Wrobel, and T. Horváth. Distance Based Approaches to Relational Learning and Clustering. In [16], pages 213–232, 2001.

[26] S. Kramer. Structural regression trees. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 812–819. MIT Press, Cambridge, MA, 1996.

[27] S. Kramer and G. Widmer. Inducing Classification and Regression Trees in First Order Logic. In [16], pages 140–159, 2001.

[28] S. Kramer, N. Lavrač, and P. Flach. Propositionalization Approaches to Relational Data Mining. In [16], pages 262–291, 2001.

[29] N. Lavrač, S. Džeroski, and M. Grobelnik. Learning nonrecursive definitions of relations with LINUS. In *Proceedings of the Fifth European Working Session on Learning*, pages 265–281. Springer, Berlin, 1991.

[30] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, Chichester, 1994. Freely available at `http://www-ai.ijs.si/SasoDzeroski/ILPBook/`.

[31] J. Lloyd. *Foundations of Logic Programming*, 2nd edition. Springer, Berlin, 1987.

[32] H. Mannila and H. Toivonen. Discovering generalized episodes using minimal occurrences. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 146–151. AAAI Press, Menlo Park, CA, 1996.

[33] R. Michalski, I. Mozetič, J. Hong, and N. Lavrač. The multi-purpose incremental learning system AQ15 and its testing application on three medical domains. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 1041–1045. Morgan Kaufmann, San Mateo, CA, 1986.

[34] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4): 295–318, 1991.

[35] S. Muggleton, editor. *Inductive Logic Programming.* Academic Press, London, 1992.

[36] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13: 245–286, 1995.

[37] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, pages 368–381. Ohmsha, Tokyo, 1990.

[38] C. Nedellec, C. Rouveirol, H. Ade, F. Bergadano, and B. Tausend. Declarative bias in inductive logic programming. In L. De Raedt, editor, *Advances in Inductive Logic Programming*, pages 82–103. IOS Press, Amsterdam, 1996.

[39] T. Niblett. A study of generalisation in logic programs. In *Proceedings of the Third European Working Session on Learning*, pages 131–138. Pitman, London, 1988.

[40] S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming.* Springer, Berlin, 1997.

[41] G. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163. Edinburgh University Press, Edinburgh, 1969.

[42] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3): 239–266, 1990.

[43] J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.

[44] J. Ramon. *Clustering and instance based learning in first order logic.* PhD Thesis. Katholieke Universiteit Leuven, Belgium, 2002. `http://www.cs.kuleuven.ac.be/~janr/phd/`.

[45] J. Ramon and M. Bruynooghe. A polynomial time computable metric between point sets. *Acta Informatica*, 37(10): 765–780.

[46] E. Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, MA, 1983.

[47] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the Twenty-first International Conference on Very Large Data Bases*, pages 407–419. Morgan Kaufmann, San Mateo, CA, 1995.

[48] J. Ullman. *Principles of Database and Knowledge Base Systems*, volume 1. Computer Science Press, Rockville, MA, 1988.

[49] V. Van Laer and L. De Raedt. How to Upgrade Propositional Learners to First Order Logic: A Case Study. In [16], pages 235–261, 2001.

[50] S. Wrobel. Inductive Logic Programming for Knowledge Discovery in Databases. In [16], pages 74–101, 2001.