3 Inductive Logic Programming in a Nutshell

Sašo Džeroski

Inductive logic programming (ILP) is concerned with the development of techniques and tools for relational data mining (RDM). Besides the ability to deal with data stored in multiple tables, ILP systems are usually able to take into account generally valid background (domain) knowledge in the form of a logic program. They also use the powerful language of logic programs for describing discovered patterns. This chapter introduces the basics of ILP and RDM. First it introduces the basics of logic programming and relates logic programming terminology to database terminology. It then discusses the major settings for, tasks of, and approaches to ILP and RDM. The tasks of learning relational classification rules, decision trees, and association rules and approaches to solving them are discussed next, followed by relational distance-based approaches. The chapter also briefly discusses recent trends in ILP and RDM research.

3.1 Introduction

From a knowledge discovery in database (KDD) perspective, we can say that inductive logic programming (ILP) is concerned with the development of techniques and tools for relational data mining (RDM). While typical data mining approaches find patterns in a given single table, relational data mining approaches find patterns in a given relational database. In a typical relational database, data resides in multiple tables. ILP tools can be applied directly to such multi-relational data to find patterns that involve multiple relations. This is a distinguishing feature of ILP approaches: most other data mining approaches can only deal with data that 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 can cause loss of meaning or information. Suppose we are given the relation $customer(CustID, Name, Age, SpendsALot)$ and the relation $purchase(CustID, ProductID, Date, Value, PaymentMode)$, where each customer can make multiple purchases, and we are in-

terested 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 the aggregation process.

The following pattern can be discovered by an ILP system 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, ILP systems are usually able to take into account generally valid background (domain) knowledge in the form of a logic program. The ability to take into account background knowledge and the expressive power of the language of discovered patterns are also distinctive for ILP.

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. ILP 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 can be found in [49] and in (chapter 4 of [15]).

The remainder of this chapter first introduces the basics of logic programming and relates logic programming terminology to database terminology. It then discusses the major settings for, tasks of, and approaches to ILP and RDM. The tasks of learning relational classification rules, decision trees, and association rules and approaches to solving them are discussed in the following three sections. Relational distance-based approaches are covered next. The chapter concludes with a brief discussion of recent trends in ILP and RDM research.

3.2 Logic Programming

We first briefly describe the basic logic programming terminology and relate it to database terminology, then proceed with a more complete introduction to logic

programming. The latter discusses both the syntax and semantics of logic programs. While syntax defines the language of logic programs, semantics is concerned with assigning meaning (truth-values) to such statements. Proof theory focuses on (deductive) reasoning with such statements.

For a thorough treatment of logic programming we refer to the standard textbook of Lloyd [31]. The overview below is mostly based on the comprehensive and easily readable text by Hogger [22].

3.2.1 The Basics of Logic Programming

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 $parent(X,Y)$, $father(X,Y)$, and $mother(X,Y)$ 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 $\{father(X,Y), mother(X,Y), \neg parent(X,Y)\}$.

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. 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. An n-ary relation p is formally defined as a set of tuples [47], 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

Table 3.1 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 -	predicate p -
a set of tuples	defined extensionally
	by a set of ground facts
relation q	predicate q
defined as a view	defined intensionally
	by a set of rules (clauses)

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 DDBs, 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, 47]. A deductive Datalog database consists of definite database clauses with no function symbols.

Table 3.1 relates basic database and logic programming terms. For a full treatment of logic programming, RDBs, and DDBs, we refer the reader to [31] and [47].

3.2.2 The Syntax and Semantics of Logic Programs

The basic concepts of logic programming include the language (syntax) of logic programs, as well as notions from model and proof theory (semantics). The syntax defines what are legal sentences/statements in the language of logic programs. Model theory (semantics) is concerned with assigning meaning (truth-values) to such statements. Proof theory focuses on (deductive) reasoning with such statements.

3.2.2.1 Syntax: The Language

A first-order *alphabet* consists of variables, predicate symbols, and function symbols (which include constants). A *variable* is a term, and a *function symbol* immediately followed by a bracketed n-tuple of terms is a term. Thus $f(g(X), h)$ is a term when f, g, and h are function symbols and X is a variable – strings starting with lowercase letters denote predicate and function symbols, while strings starting with uppercase letters denote variables. A *constant* is a function symbol of arity 0 (i.e., followed by a bracketed 0-tuple of terms, which is often left implicit). A *predicate symbol*

immediately followed by a bracketed n-tuple of terms is called an *atomic formula* or *atom*. For example, $mother(maja, filip)$ and $father(X, Y)$ are atoms.

A *well-formed formula* (also called a sentence or statement) is either an atomic formula or takes one of the following forms: F, (F), $\overline{F}$, $F \vee G$, $F \wedge G$, $F \leftarrow G$, $F \leftrightarrow G$, $\forall X : F$ and $\exists X : F$, where F and G are well-formed formulae and X is a variable. $\overline{F}$ denotes the negation of F, $\vee$ denotes logical disjunction (or), and $\wedge$ logical conjunction (and). $F \leftarrow G$ stands for implication (F if G, $F \vee \overline{G}$) and $F \leftrightarrow G$ stands for equivalence (F if and only if G). $\forall$ and $\exists$ are the universal (for all X F holds) and the existential quantifier (there exists an X such that F holds). In the formulae $\forall X : F$ and $\exists X : F$, all occurrences of X are said to be *bound*. A *sentence* or a *closed formula* is a well-formed formula in which every occurrence of every variable symbol is bound. For example, $\forall Y \exists X father(X, Y)$ is a sentence, while $father(X, andy)$ is not.

The clausal form is a normal form for first-order sentences. A *clause* is a disjunction of *literals* – a *positive literal* is an atom, a *negative literal* the negation of an atom – preceded by a prefix of universal quantifiers, one for each variable appearing in the disjunction. In other words, a clause is a formula of the form $\forall X_1 \forall X_2 ... \forall X_s (L_1 \vee L_2 \vee ... L_m)$, where each L_i is a literal and $X_1, X_2, ..., X_s$ are all the variables occurring in $L_1 \vee L_2 \vee ... L_m$.

A clause can also be represented as a finite set (possibly empty) of literals. The set $\{A_1, A_2, ..., A_h, \overline{B_1}, \overline{B_2}, ..., \overline{B_b}\}$, where A_i and B_i are atoms, stands for the clause $(A_1 \vee ... \vee A_h \vee \overline{B_1} \vee ... \vee \overline{B_b})$, which is equivalently represented as $A_1 \vee ... \vee A_h \leftarrow B_1 \wedge ... \wedge B_b$. Most commonly, this same clause is written as $A_1, ..., A_h \leftarrow B_1, ..., B_b$, where $A_1, ..., A_h$ is called the *head* and $B_1, ..., B_b$ the *body* of the clause. Commas in the head of the clause denote logical disjunction, while commas in the body of the clause denote logical conjunction. A set of clauses is called a *clausal theory* and represents the conjunction of its clauses.

A clause is a *Horn clause* if it contains at most one positive literal; it is a *definite clause* if it contains exactly one positive literal. A set of definite clauses is called a *definite logic program*. A fact is a definite clause with an empty body, e.g., $parent(mother(X), X) \leftarrow$, also written simply as $parent(mother(X), X)$. A *goal* (also called a *query*) is a Horn clause with no positive literals.

A *program clause* is a clause of the form $A \leftarrow L_1, ..., L_m$ where A is an atom, and each of $L_1, ..., L_m$ is a positive or negative literal. A negative literal in the body of a program clause is written in the form *not* B, where B is an atom. A *normal program* (or *logic program*) is a set of program clauses. A *predicate definition* is a set of program clauses with the same predicate symbol (and arity) in their heads.

Let us now illustrate the above definitions with some examples. The clause

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

is a definite program clause, while the clause

$$daughter(X, Y) \leftarrow not\ male(X), father(Y, X)$$

is a normal program clause. Together, the two clauses constitute a predicate definition of the predicate *daughter*/2. This predicate definition is also a normal logic program. The first clause is an abbreviated representation of the formula

$$\forall X \forall Y : daughter(X,Y) \vee \overline{female(X)} \vee \overline{mother(Y,X)}$$

and can also be written in set notation as

$$\{daughter(X,Y), \overline{female(X)}, \overline{mother(Y,X)}\}.$$

The set of *variables* in a term, atom, or clause F is denoted by $vars(F)$. 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. A term, atom, or clause F is called *ground* when there is no variable occurring in F, i.e., $vars(F) = \emptyset$. The fact $daughter(mary, ann)$ is thus ground.

A clause or clausal theory is called *function-free* if it contains only variables as terms, i.e., contains no function symbols (this also means no constants). The clause $daughter(X,Y) \leftarrow female(X), mother(Y,X)$ is function-free and the clause $even(s(s(X)) \leftarrow even(X)$ is not. A *Datalog clause* (program) is a definite clause (program) that contains no function symbols of nonzero arity. This means that only variables and constants can be used as predicate arguments. The size of a term, atom, clause, or a clausal theory T is the number of symbols that appear in T, i.e., the number of all occurrences in T of predicate symbols, function symbols, and variables.

3.2.2.2 Semantics: Model Theory

Model theory is concerned with attributing meaning (truth-value) to sentences (well-formed formulae) in a first-order language. Informally, the sentence is mapped to some statement about a chosen domain through a process known as interpretation. An *interpretation* is determined by the set of ground facts (ground atomic formulae) to which it assigns the value true. Sentences involving variables and quantifiers are interpreted by using the truth-values of the ground atomic formulae and a fixed set of rules for interpreting logical operations and quantifiers, such as "$\overline{F}$ is true if and only if F is false."

An interpretation which gives the value true to a sentence is said to satisfy the sentence; such an interpretation is called a *model* for the sentence. An interpretation which does not satisfy a sentence is called a *counter-model* for that sentence. By extension, we also have the notion of a model (counter-model) for a set of sentences (e.g., for a clausal theory): an interpretation is a model for the set if and only if it is a model for each of the set's members. A sentence (set of sentences) is *satisfiable* if it has at least one model; otherwise it is *unsatisfiable*.

A sentence F *logically implies* a sentence G if and only if every model for F is also a model for G. We denote this by $F \models G$. Alternatively, we say that G is a *logical* (or *semantic*) *consequence* of F. By extension, we have the notion of *logical implication* between sets of sentences.

A *Herbrand interpretation* over a first-order alphabet is a set of ground facts constructed with the predicate symbols in the alphabet and the ground terms from the corresponding Herbrand domain of function symbols; this is the set of ground atoms considered to be true by the interpretation. A Herbrand interpretation I is a model for a clause c if and only if for all substitutions θ such that $c\theta$ is ground $body(c)\theta \subset I$ implies $head(c)\theta \cap I \neq \emptyset$. In that case, we say c is true in I. A Herbrand interpretation I is a model for a clausal theory T if and only if it is a model for all clauses in T. We say that I is a *Herbrand model* of c, respectively T.

Roughly speaking, the truth of a clause c in a (finite) interpretation I can be determined by running the goal (query) $body(c), not\ head(c)$ on a database containing I, using a *theorem prover* such as Prolog [4]. If the query succeeds, the clause is false in I; if it fails, the clause is true. Analogously, one can determine the truth of a clause c in the *minimal (least) Herbrand model* of a theory T by running the goal $body(c), not\ head(c)$ on a database containing T.

To illustrate the above notions, consider the Herbrand interpretation $i = \{parent(saso, filip), parent(maja, filip), son(filip, saso), son(filip, maja)\}$.
The clause $c = parent(X, Y) \leftarrow son(Y, X)$ is true in i, i.e., i is a model of c. On the other hand, i is not a model of the clause $parent(X, X) \leftarrow$ (which means that everybody is their own parent).

3.2.3 Semantics: Proof Theory

Proof theory focuses on (deductive) reasoning with logic programs. Whereas model theory considers the assignment of meaning to sentences, proof theory considers the generation of sentences (conclusions) from other sentences (premises). More specifically, proof theory considers the *derivability* of sentences in the context of some set of inference rules, i.e., rules for sentence derivation. Formally, an inference system consists of an initial set S of sentences (axioms) and a set R of inference rules.

Using the inference rules, we can derive new sentences from S and/or other derived sentences. The fact that sentence s can be derived from S is denoted $S \vdash s$. A proof is a sequence $s_1, s_2, ..., s_n$, such that each s_i is either in S or derivable using R from S and $s_1, ..., s_{i-1}$. Such a proof is also called a *derivation* or *deduction*. Note that the above notions are of entirely syntactic nature. They are directly relevant to the computational aspects of automated deductive inference.

The set of inference rules R defines the derivability relation $\vdash$. A set of inference rules is *sound* if the corresponding derivability relation is a subset of the logical implication relation, i.e., for all S and s, if $S \vdash s$, then $S \models s$. It is *complete* if the other direction of the implication holds, i.e., for all S and s, if $S \models s$, then $S \vdash s$. The properties of *soundness* and *completeness* establish a relation between

the notions of *syntactic* ($\vdash$) and *semantic* ($\models$) *entailment* in logic programming and first-order logic. When the set of inference rules is both sound and complete, the two notions coincide.

Resolution comprises a single inference rule applicable to clausal-form logic. From any two clauses having an appropriate form, resolution derives a new clause as their consequence. For example, the clauses $daughter(X, Y) \leftarrow female(X), parent(Y, X)$ and $female(sonja) \leftarrow$ resolve into $daughter(sonja, Y) \leftarrow parent(Y, sonja)$. Resolution is sound: every resolvent is implied by its parents. It is also refutation complete: the empty clause is derivable by resolution from any set S of Horn clauses if S is unsatisfiable.

3.3 Inductive Logic Programming: Settings and Approaches

Logic programming as a subset of first-order logic is mostly concerned with deductive inference. ILP, 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.

In this section, we discuss the different ILP settings as well as the different relational learning tasks, starting with the induction of logic programs (sets of relational rules). We also discuss the two major approaches to solving relational learning tasks, namely transforming relational problems to propositional form and upgrading propositional algorithms to a relational setting.

3.3.1 Logical Settings for Concept Learning

One of the most basic and most often considered tasks in machine learning is the task of inductive concept learning (table 3.3.1). Given $\mathcal{U}$, a universal set of objects (observations), a *concept* $\mathcal{C}$ is a subset of objects in $\mathcal{U}$, $\mathcal{C} \subseteq \mathcal{U}$. For example, if $\mathcal{U}$ is the set of all patients in a given hospital, $\mathcal{C}$ could be the set of all patients diagnosed with hepatitis A. The task of *inductive concept learning* is defined as follows: Given instances and non-instances of concept $\mathcal{C}$, find a hypothesis (classifier) H able to tell whether $x \in \mathcal{C}$, for each $x \in \mathcal{U}$.

To define the task of inductive concept learning more precisely, we need to specify $\mathcal{U}$ the space of instances (examples), as well as the space of hypotheses considered. This is done through specifying the languages of examples (L_E) and concept descriptions (L_H). In addition, a coverage relation $covers(H, e)$ has to be specified, which tells us when an example e is considered to belong to the concept represented by hypothesis H. Examples that belong to the target concept are termed positive; those that do not are termed negative. Given positive and negative examples, we want hypotheses that are complete (cover all positive examples) and consistent (do not cover negative examples).

Looking at concept learning in a logical framework, De Raedt [11] considers three settings for concept learning. The key aspect that varies in these settings is

Table 3.2 The task of inductive concept learning

Given:
■ a language of examples L_E
■ a language of concept descriptions L_H
■ a **covers** relation between L_H and L_E, defining when an example e is *covered* by a hypothesis$H$: $covers(H, e)$
■ sets of positive P and negative N examples described in L_E
Find hypothesis H from L_H, such that
■ **completeness:** H covers all positive examples $p \in P$
■ **consistency:** H does not cover any negative example $n \in N$

the notion of coverage, but the languages L_E and L_H vary as well. We characterize these for each of the three settings below.

- In *learning from entailment*, the coverage relation is defined as $covers(H, e)$ iff $H \models e$. The hypothesis logically entails the example. Here H is a clausal theory and e is a clause.
- In *learning from interpretations*, we have $covers(H, e)$ iff e is model of H. The example has to be a model of the hypothesis. H is a clausal theory and e is a Herbrand interpretation.
- In *learning from satisfiability*, $covers(H, e)$ iff $H \wedge e \not\models \bot$. The example and the hypothesis taken together have to be satisfiable. Here both H and e are clausal theories.

The setting of learning from entailment, introduced by Muggleton [34], is the one that has received the most attention in the field of ILP. The alternative ILP setting of learning from interpretations was proposed by De Raedt and Džeroski [14]: this setting is a natural generalization of propositional learning. Many learning algorithms for propositional learning have been upgraded to the learning from interpretations ILP setting. Finally, the setting of learning from satisfiability was introduced by Wrobel and Džeroski [50], but has rarely been used in practice due to computational complexity problems.

De Raedt [11] also discusses the relationships among the three settings for concept learning. Learning from finite interpretations reduces to learning from entailment. Learning from entailment reduces to learning from satisfiability. Learning from interpretations is thus the easiest and learning from satisfiability the hardest of the three settings.

As introduced above, the logical settings for concept learning do not take into account background knowledge, one of the essential ingredients of ILP. However, the definitions of the settings are easily extended to take it into account. Given background knowledge B, which in its most general form can be a clausal theory,

the definition of coverage should be modified by replacing H with $B \wedge H$ for all three settings.

3.3.2 The ILP Task of Relational Rule Induction

The most commonly addressed task in ILP is the task of learning logical definitions of relations [40], 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 [44] (rightfully considered as one of the most influential ancestors of ILP) and one of the best known ILP systems FOIL [40].

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] (and discussed in the previous section), is thus also called learning from entailment.

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 3.3. 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.

Table 3.3 A simple ILP problem: learning the *daughter* relation. Positive examples are denoted by $\oplus$ and negative by $\ominus$

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

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).

Declarative bias [38] explicitly specifies the language of hypotheses (clauses) considered by the ILP system at hand. This is input to the learning system (and not hard-wired in the learning algorithm). Various types of declarative bias have been used by different ILP systems, such as argument types and input/output modes, parameterized language bias (e.g., maximum number of variables, literals, depth of variables, arity, etc.), clause templates, and grammars. For example, a suitable clause template for learning family relationships would be $P(X, Y) \leftarrow Q(X, Z), R(Z, Y)$. Here P, Q, and R are second order variables that can be replaced by predicates, e.g., *grandmother*, *mother*, and *parent*. The same template can be used to learn the notions of grandmother and a grandfather.

3.3.3 Other Tasks of Relational Learning

Initial efforts in ILP focused on relational rule induction, more precisely on concept learning in first-order logic and synthesis of logic programs; cf. [34]. An overview of early work is given in the textbook on ILP by Lavrač and Džeroski [30]. Representative early ILP systems addressing this task are CIGOL [36], FOIL [40], GOLEM [37], and LINUS [29]. More recent representative ILP systems are PROGOL [35] and ALEPH [46].

State-of-the-art ILP approaches now span most of the spectrum of data mining tasks and use a variety of techniques to address these. The distinguishing features of using multiple relations directly and discovering patterns expressed in first-order logic are present throughout: the ILP approaches can thus be viewed as upgrades of traditional approaches. Van Laer and De Raedt [48] (chapter 10 of [15]) present a case study of upgrading a propositional approach to classification rule induction to first-order logic. Note, however, that upgrading to first-order logic is non-trivial: the expressive power of first-order logic implies computational costs and much work is needed in balancing the expressive power of the pattern languages used and the computational complexity of the data mining algorithm looking for such patterns. This search for a balance between the two has occupied much of the ILP research in the last ten years.

Present ILP approaches to multi-class classification involve the induction of relational classification rules (ICL [48]), as well as first order logical decision trees in TILDE [3] and S-CART [26]. ICL upgrades the propositional rule inducer CN2 [6]. TILDE and S-CART upgrade decision tree induction as implemented in C4.5 [41] and CART [5]. A nearest-neighbor approach to relational classification is implemented in RIBL [21] and its successor RIBL2.

Relational regression approaches upgrade propositional regression tree and rules approaches. TILDE and S-CART, as well as RIBL2, can handle continuous classes. FORS [23] learns decision lists (ordered sets of rules) for relational regression.

The main nonpredictive or descriptive data mining tasks are clustering and discovery of association rules. These have been also addressed in a first-order logic setting. The RIBL distance measure has been used to perform hierarchical agglomerative clustering in RDBC , as well as k-means clustering (see section 3.7). Section 3.6 describes a relational approach to the discovery of frequent queries and query extensions, a first-order version of association rules.

With such a wide arsenal of RDM techniques, there is also a variety of practical applications. ILP has been successfully applied to discover knowledge from relational data and background knowledge in the areas of molecular biology (including drug design, protein structure prediction, and functional genomics), environmental sciences, traffic control, and natural language processing. An overview of such applications is given by Džeroski [19] and (chapter 14 in [15]).

3.3.4 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 in 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 [7], 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 3.4 Nonground background knowledge for learning the *daughter* relation

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

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 3.3. However, since the LINUS approach can use nonground background knowledge, let us assume that the background knowledge from table 3.4 is given.

Table 3.5 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*

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 3.5, 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 3.5, *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 3.5:

$Class = \oplus$ **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 [16, 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] (chapter 11 of [15]).

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 [12] 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 and ILP techniques by upgrading propositional approaches.

3.3.5 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 [48] 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, an 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 [40] system can be seen as an upgrade of the propositional rule induction program CN2 [7]. Another well-known ILP system, PROGOL [35], 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 [9] (chapter 8 of [15]), [8], 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 [15], [21]). 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 two sections, we briefly review how the propositional approaches for association rule discovery and decision tree inducion have been lifted to a relational framework, highlighting the key differences between the relational algorithms and their propositional counterparts.

3.4 Relational Classification Rules

The first and still most commonly addressed problem in ILP is the one of learning logic programs (sets of relational rules for binary classification). This section first describes the covering algorithm for inducing sets of rules, then the induction of individual rules. In particular, we discuss how the space of rules/clauses is structured and searched.

3.4.1 The Covering Approach to Relational Rule Induction

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

Table 3.6 Top-down (general-to-specific) search of refinement graphs

hypothesis $H := \emptyset$
repeat {covering}
    clause $c := p(X_1, ...X_n) \leftarrow$
    repeat {specialization}
        build the set S of all refinements of c
        $c :=$ the best element of S (according to a heuristic)
    until stopping criterion is satisfied ($B \cup H \cup \{c\}$ is consistent)
    add c to H
    delete all examples from P entailed by $B \cup H \cup \{c\}$
until stopping criterion is satisfied
        ($B \cup H \cup \{c\}$ is complete)

$\ominus$ (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 [41] or induced directly [33, 6].

ILP systems dealing with the classification task typically adopt the covering approach of rule induction systems (table 3.6). 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.

3.4.2 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 $c\theta \subseteq c'$ [39].

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 $\{daughter(X, Y), \neg parent(Y, X)\}$ where all variables are assumed to be universally quantified, $\neg$ 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 $c' = daughter(X, Y) \leftarrow female(X), parent(Y, X)$ under the empty substitution $\theta = \emptyset$, since $\{daughter(X, Y), \neg parent(Y, X)\}$ is a proper subset of $\{daughter(X, Y), \neg female(X), \neg parent(Y, X)\}$. 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.

Under a semantic notion of generality, c is more general than c' if c logically entails c' ($c \models c'$). If c θ-subsumes c', then $c \models c'$. The reverse is not always true. The syntactic, θ-subsumption-based, generality is computationally more feasible. Namely, semantic generality is in general undecidable. Thus, syntactic generality is frequently used in ILP systems.

The relation $\leq$ defined by θ-subsumption introduces a lattice on the set of reduced clauses [39]: this enables ILP systems to prune large parts of the search space. θ-subsumption also provides the basis for clause construction by top-down searching of refinement graphs and bounding the search of refinement graphs from below by using a bottom clause (which can be constructed as least general generalizations, i.e., least upper bounds of example clauses in the θ-subsumption lattice).

3.4.3 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 [44]. 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: $\rho(c) = \{c' \mid c' \in \mathcal{L},\ c < c'\}$.

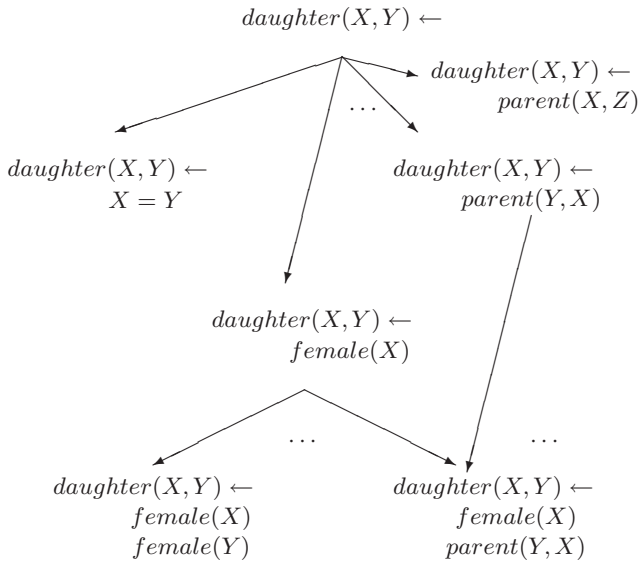


Figure 3.1 Part of the refinement graph for the family relations problem.

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:

- 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 3.1 depicts a part of the refinement graph for the family relations problem defined in table 3.3, 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 with an empty body $c = daughter(X, Y) \leftarrow$. 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: $X = Y$ (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 examples (positive and negative). Its refinements are then considered, then their refinements in turn, and 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 levelwise, 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 [35].

3.5 Relational Decision Trees

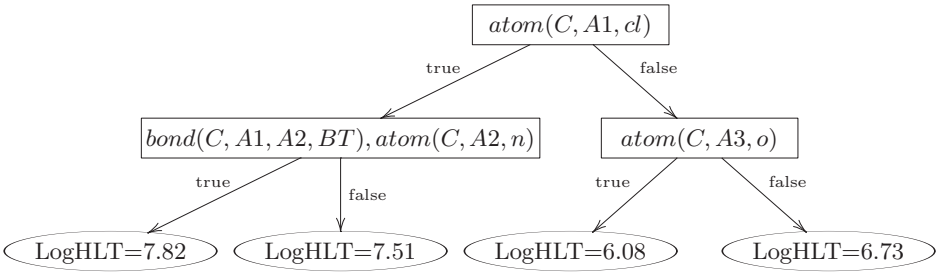
Decision tree induction is one of the major approaches to data mining. Upgrading this approach to a relational setting has thus been of great importance. In this section, we first look into what relational decision trees are, i.e., how they are defined, then discuss how such trees can be induced from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

3.5.1 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 of a relational decision tree is given in figure 3.3. It predicts the maintenance action A to 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 $maintenance(M, A)$,

Figure 3.2 A relational regression tree for predicting the degradation time $LogHLT$ of a chemical compound C (target predicate $degrades(C, LogHLT)$).



the class variable is A, and background knowledge predicates are $haspart(M, X)$, $worn(X)$, and $irreplaceable(X)$.

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 3.3 is a relational classification tree, while the tree in figure 3.2 is a relational regression tree. The latter predicts the degradation time (the logarithm of the mean half-life time in water [18]) 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

Table 3.7 A decision list representation of the relational decision tree in figure 3.3

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

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 3.2 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 3.3 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 3.2 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).

Table 3.8 A decision list representation of the relational regression tree for predicting the biodegradability of a compound, given in figure 3.2

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

Table 3.9 A logic program representation of the relational decision tree in figure 3.3

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

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.

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 3.3 is given in table 3.7. 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 3.2 is given in table 3.8.

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 3.3, is given in table 3.9.

3.5.2 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, 13] is an upgrade of C4.5 [41]. According to the upgrading recipe, both SCART and TILDE have their propositional counterparts as special cases. The actual algorithms thus closely follow

Table 3.10 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]
```

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 3.10).

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. 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 depends 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 on 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 statements, used in the induction of the regression tree in figure 3.2 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 *TestsOnYesBranches-Sofar*), − 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 3.2, $TestsOnYesBranchesSofar = atom(C, A1, cl)$. 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 multi-relational data mining approaches. They have been successfully applied to a

number of practical problems. These include learning to predict the biodegradability of chemical compounds [18] and learning to predict the structure of diterpene compounds from their nuclear magnetic resonance spectra [17].

3.6 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 to discover frequent relational patterns.

3.6.1 Frequent Datalog Queries and Query Extensions

Dehaspe and colleagues [8], [9] (chapter 8 of [15]) 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., $a(Q, \mathbf{r}, key) = |\{\theta \in answerset(key, \mathbf{r}) | Q\theta \text{ succeeds w.r.t. } \mathbf{r}\}|$. 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." If itemsets 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 rule 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 subquery $l_{m+1},\ldots l_n$ 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, where E is defined as $E = ?-person(X),parent(X,Y) \rightsquigarrow hasPet(Y,Z)$, 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."

3.6.2 Discovering Frequent Queries: WARMR

The task of discovering frequent queries is addressed by the RDM system WARMR [8]. WARMR takes as input a database $\mathbf{r}$, a frequency threshold $minfreq$, and declarative language bias $\mathcal{L}$. $\mathcal{L}$ 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 [1] 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 test 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 $\mathbf{r}$. 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 3.4.2 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 3.11 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 3.11. 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 ($\pm$). 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 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, 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 subqueries 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 subquery $? - person(X), hasPet(Y, cat)$ 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 3.12.

WARMR upgrades APRIORI to a multi-relational setting following the upgrading recipe (see section 3.3.5). 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 [45], as well as approaches to discovering sequential patterns [2], including general epi-

Table 3.12 The WARMR algorithm for discovering frequent Datalog queries.

Algorithm WARMR($\mathbf{r}$, $\mathcal{L}$, *key*, *minfreq*; Q)
Input: Database $\mathbf{r}$; 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 $\mathcal{Q}_1 := \{$*?- 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 $\mathcal{Q}_{d+1}$

sodes [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 [10] and genome-wide prediction of protein functional class [24].

3.7 Relational Distance-Based Methods

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

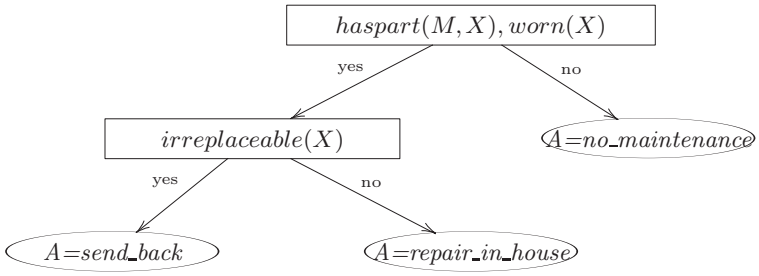


Figure 3.3 A relational decision tree, predicting the class variable A in the target predicate $maintenance(M, A)$.

standard statistical approaches, such as nearest-neighbor prediction or hierarchical agglomerative clustering. In their system RIBL, Emde and Wettschereck [21] 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].

3.7.1 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 3.13, 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 3.13 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 [21] 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.

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 3.4.

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. Thus we have

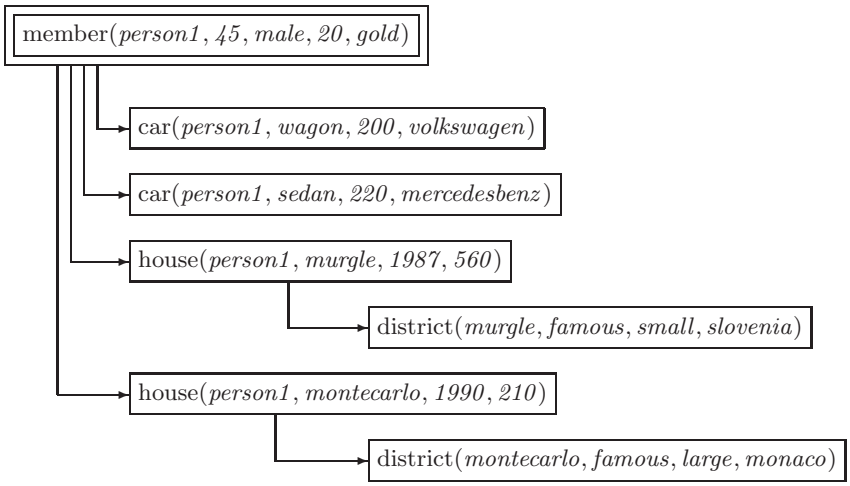


Figure 3.4 All facts related to member(*person1*, *45*, *male*, *2000000*, *gold*) constructed with respect to the background knowledge in table 3.13 and a depth limit of 2.

$$
\begin{aligned}
F_1, car = \{&\text{car}(person1, wagon, 200, volkswagen),\\
&\text{car}(person1, sedan, 220, mercedesbenz)\}\\
F_2, car = \{&\text{car}(person2, roadster, 240, audi),\\
&\text{car}(person2, coupe, 260, bmw)\}\\
F_1, house = \{&\text{house}(person1, murgle, 1987, 560),\\
&\text{house}(person1, montecarlo, 1990, 210)\}\\
F_2, house = \{&\text{house}(person2, murgle, 1999, 430)\}.
\end{aligned}
$$

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:

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

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 $d(F_1, car, F_2, car) = 0.5 \cdot [min_{c \in F_2, car} d(\text{car}(person1, wagon, 200, volkswagen), c) + 0.5 \cdot min_{c \in F_2, car} d(\text{car}(person1, sedan, 220, mercedesbenz), c)] = 0.5 \cdot [(1 + |200 - 240|/100 + 1)/3, (1 + |220 - 240|/100 + 1)/3] = 11/15$. 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 $0.5 \cdot [1 + min((0 + (1999 - 1987)/100 + |430 - 560|/1000)/3, 0.5 \cdot [1 + min((2/3 + (1999 - 1990)/100 + (430 - 210)/1000)/3)] = 0.5 + 0.5 \cdot min(0.25/3, (2/3 + 0.31)/3) = 13/24$. $d(e_1, e_2)$ 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 [42]. However, some relational distance measures that are metrics have been proposed recently [43]. 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.

3.7.2 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 [21] addresses the problem of prediction, more precisely classification. It uses the k-nearest neighbor 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 [17], 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.

3.8 Recent Trends in ILP and RDM

Hot topics and recent advances in ILP and RDM mirror the hot topics in data mining and machine learning. These include scalability issues, ensemble methods, and kernel methods.

Scalability issues do indeed deserve a lot of attention when learning in a relational setting, as the complexity of learning increases with the expressive power of the hypothesis language. Scalability methods for ILP include classical ones, such as sampling or turning the loop of hypothesis evaluation inside out (going through each example once) in decision tree induction. Methods more specific to ILP, such as query packs, have also been considered.

Boosting was the first ensemble method to be used on top of a relational learning system. This was followed by bagging. More recently, methods for learning random forests have been adapted to the relational setting.

Kernel methods have become the mainstream of research in machine learning and data mining in recent years. The development of kernel methods for learning in a relational setting has thus emerged as a natural research direction. Significant effort has been devoted to the development of kernels for structured/relational data, such as graphs and sequences.

The latest developments in ILP and RDM are discussed in a special issue of *SIGKDD Explorations* [20]. Besides the topics mentioned above, the hottest research topic in ILP and RDM is the study of probabilistic representations and learning methods. A variety of these have been recently considered. A comprehensive survey of such methods is presented in this book.

References

[1] 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.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering*, 1995.

[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, UK, 2001.

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

[6] P. Clark and R. Boswell. Rule induction with CN2: Some recent improvements. In *Proceedings of the Fifth European Working Session on Learning*, 1991.

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

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

[9] L. Dehaspe and H. Toivonen. Discovery of relational association rules. In [15], pages 189–212, 2001.

[10] 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*, 1998.

[11] L. De Raedt. Logical settings for concept learning. *Artificial Intelligence*, 95: 187–201, 1997.

[12] L. De Raedt. Attribute-value learning versus inductive logic programming: the missing links. In *Proceedings of the Eighth International Conference on Inductive Logic Programming*, 1998.

[13] L. De Raedt, H. Blockeel, L. Dehaspe, and W. Van Laer. Three companions for data mining in first order logic. In [15], pages 105–139, 2001.

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

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

[16] S. Džeroski, S. Muggleton, and S. Russell. PAC-learnability of determinate logic programs. In *Proceedings of the Fifth ACM Workshop on Computational Learning Theory*, 1992.

[17] 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.

[18] S. Džeroski, H. Blockeel, B. Kompare, S. Kramer, B. Pfahringer, and W. Van Laer. Experiments in predicting biodegradability. In *Proceedings of the*

International Workshop on Inductive Logic Programming, 1999.

[19] S. Džeroski. Relational data mining applications: An overview. In [15], pages 339–364, 2001.

[20] S. Džeroski and L. De Raedt, editors. *SIGKDD Explorations,* Special Issue on Multi-Relational Data Mining, 5(1), 2003.

[21] W. Emde and D. Wettschereck. Relational instance-based learning. In *Proceedings of the Thirteenth International Conference on Machine Learning*, 1996.

[22] C. Hogger. *Essentials of Logic Pogramming.* Clarendon Press, Oxford, UK, 1990.

[23] A. Karalič and I. Bratko. First order regression. *Machine Learning* 26: 147-176, 1997.

[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*, 2000.

[25] M. Kirsten, S. Wrobel, and T. Horváth. Distance based approaches to relational learning and clustering. In [15], pages 213–232, 2001.

[26] S. Kramer. Structural regression trees. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996.

[27] S. Kramer and G. Widmer. Inducing classification and regression trees in first order logic. In [15], pages 140–159, 2001.

[28] S. Kramer, N. Lavrač, and P. Flach. Propositionalization approaches to relational data mining. In [15], 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*, 1991.

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

[31] J. Lloyd. *Foundations of Logic Programming*, 2nd edition. Springer-Verlag, 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*, 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*, 1986.

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

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

[36] S. Muggleton and W. Buntine. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, 1988.

[37] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, 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] 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.

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

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

[42] J. Ramon. *Clustering and Instance Based Learning in First Order Logic*. PhD Thesis. Katholieke Universiteit Leuven, Belgium, 2002.

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

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

[45] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the Twenty-first International Conference on Very Large Data Bases*, 1995.

[46] A. Srinivasan. The Aleph manual. Technical Report, Computing Laboratory, Oxford University, Oxford, UK, 2000.

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

[48] V. Van Laer and L. De Raedt. How to upgrade propositional learners to first order logic: A case study. In [15], pages 235–261, 2001.

[49] S. Wrobel. Inductive logic programming for knowledge discovery in databases. In [15], pages 74–101, 2001.

[50] S. Wrobel and S. Džeroski. The ILP description learning problem: Towards a general model-level definition of data mining in ILP. In *Proceedings Fachgruppentreffen Maschinelles Lernen*. University of Dortmund, Germany, 1995.