Discovery of Relational Association Rules

Luc Dehaspe * and Hannu T.T. Toivonen †

*Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium
email: Luc.Dehaspe@PharmaDM.com

† Nokia Research Center
P.O. Box 407, FIN-00045 Nokia Group, Finland
email: Hannu.TT.Toivonen@nokia.com

Abstract. Within KDD, the discovery of frequent patterns has been studied in a variety of settings. In its simplest form, known from association rule mining, the task is to discover all frequent item sets, i.e., all combinations of items that are found in a sufficient number of examples. We present algorithms for relational association rule discovery that are well-suited for exploratory data mining. They offer the flexibility required to experiment with examples more complex than feature vectors and patterns more complex than item sets.

1 Introduction

Discovery of recurrent patterns in large data collections has become one of the central topics in data mining. In tasks where the goal is to uncover structure in the data and where there is no preset target concept, the discovery of relatively simple but frequently occurring patterns has shown good promise.

Association rules [2] are a basic example of this kind of setting. A prototypical application example is in market basket analysis: first find out which items tend to be sold together (the frequent item set discovery phase), and next postprocess these frequent patterns into rules about the conditional probability that one set of items will be in the basket given another set already there (the association rule discovery phase). The motivation for such an application is the potentially high business value of the discovered patterns. At the heart of the task is the problem of determining all combinations of items that occur frequently together, where “frequent” is defined as “exceeding a user-specified frequency threshold”. The use of a frequency threshold for filtering out non-interesting patterns is natural for a large number of data mining problems. Patterns that are rare, e.g., that concern only a couple of customers, are probably not reliable nor useful for the user.

Problem settings that are close to the problem of discovering association rules include the use of item type hierarchies [14,15,31], the discovery of episodes in event sequences [21,23], and the search of sequential patterns from series of transactions [4,32]. For all these cases the pattern language is more complex
than in the market basket application, and specialized algorithms exist for the tasks.

We present a powerful inductive logic programming algorithm, \textsc{Warmr} \cite{11}, for a large subfamily of this type of tasks. \textsc{Warmr} discovers frequent PROLOG queries that succeed with respect to a “sufficient” number of examples. In other words, the pattern language consists of PROLOG queries, and \textsc{Warmr} outputs those that are “frequent” in a given PROLOG (or relational) database. The PROLOG formulation is very general, as it allows the use of variables and multiple relations in patterns, and it thus significantly extends the expressive power of patterns that can be found.

The flexibility of \textsc{Warmr} is a strong advantage over previous algorithms for the discovery of frequent patterns. Each discovery task is specified to \textsc{Warmr} in terms of a declarative language bias definition. The language bias declarations determine which PROLOG queries are admissible. With different languages (and databases) \textsc{Warmr} can be adapted to diverse tasks, including the settings mentioned above and also more complex novel problems, without requiring changes to the implementation. \textsc{Warmr} thus supports truly explorative data mining; pattern types can be modified and experimented with very flexibly with a single tool.

This chapter is organized as follows. We start in Section 2 by introducing the relational patterns and explain how they relate to the original item sets and association rules. In Section 3, we discuss two evaluation measures that are useful to filter and rank output rules. In Section 4, we introduce a formalism for specifying the class of patterns to be discovered. In Section 5, we describe \textsc{Warmr}, an algorithm that discovers frequent PROLOG queries and relational association rules. A sample run with \textsc{Warmr} is shown in Section 6. Finally, in Section 7, we conclude with a brief discussion.

2 From Association Rules to Query Extensions

Essentially, the type of database we consider is a relational database (see, e.g., \cite{13}), and the type of patterns we consider reduce to SQL queries. We say that a query-pattern matches the database if the set of tuples returned by the query is not empty.

Consider the relational database \textit{RD} in Table 1 that consists of three relations.

As an example of a pattern, let us consider SQL query \textit{SQ1}:

\begin{verbatim}
SELECT Customer.Id, Parent.IdJr
FROM Customer, Parent, Buys
WHERE Customer.Id = Parent.IdSr
AND Parent.IdJr = Buys.Id
AND Buys.Item = 'cola'
\end{verbatim}

i.e., “find couples (customer,child) where the child buys cola”.
<table>
<thead>
<tr>
<th>Customer</th>
<th>Parent</th>
<th>Buys</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>idstr</td>
<td>id</td>
</tr>
<tr>
<td>allen</td>
<td>allen</td>
<td>allen wine</td>
</tr>
<tr>
<td>bill</td>
<td>bill</td>
<td>cola</td>
</tr>
<tr>
<td>carol</td>
<td>carol</td>
<td>pizza</td>
</tr>
<tr>
<td>diana</td>
<td>diana</td>
<td>pizza</td>
</tr>
</tbody>
</table>

Table 1. Relational database $RD$ with customer information

We say SQL query-pattern $SQ_1$ matches relational database $RD$ because there exists a tuple, that is $<\text{allen, bill}>$, in the resulting relation.

In practice we will use PROLOG [8, 34] for the representation of data and patterns. More specifically, databases correspond to PROLOG programs and patterns reduce to queries evaluated w.r.t. those programs. We say the pattern matches the database if the query succeeds with the program loaded.

Assume PROLOG database $D$ shown in Table 2,

<table>
<thead>
<tr>
<th>customer(\text{allen}).</th>
<th>parent(\text{allen, bill}).</th>
<th>buys(\text{allen, wine}).</th>
</tr>
</thead>
<tbody>
<tr>
<td>customer(\text{bill}).</td>
<td>parent(\text{allen, carol}).</td>
<td>buys(\text{bill, cola}).</td>
</tr>
<tr>
<td>customer(\text{carol}).</td>
<td>parent(\text{bill, zoe}).</td>
<td>buys(\text{bill, pizza}).</td>
</tr>
<tr>
<td>customer(\text{diana}).</td>
<td>parent(\text{carol, diana}).</td>
<td>buys(\text{diana, pizza}).</td>
</tr>
</tbody>
</table>

Table 2. Prolog database $D$ with customer information

and query $Q_1$:

\[
\text{?- customer}(X), \text{parent}(X,Y), \text{buys}(Y,\text{cola}).
\]

We say query-pattern $Q_1$ matches database $D$ because there exists an answer $\{X = \text{allen}, Y = \text{bill}\}$ for which the pattern succeeds. Observe that PROLOG database $D$ and PROLOG query $Q_1$ are equivalent to respectively relational database $RD$ and SQL query $RQ_1$.

Queries can be viewed as “relational item sets”. Compare example query $Q_1$ to itemset:

$$\{\text{customer, parent, child, cola, buyer}\}$$

The “market-basket interpretation” of this pattern would be that in the example a customer, a parent, a child, and a cola buyer occur. This is in fact also partly the meaning of query pattern $Q_1$. The variables $X$ and $Y$ in $Q_1$ however add extra information: customer and parent are the same (variable $X$), child and cola buyer are the same (variable $Y$), and parent and child belong to the same family. We could drop those variables and construct a query equivalent to the itemset above. This illustrates the fact that queries are indeed a more expressive variant of item sets.
Now let us consider the relational version of association rules. We introduce new terminology to describe these rules. This can be motivated by two observations. First, at least some people in the data mining community are not happy with the name association rules [20]. Second, especially in first-order logic, association rules in their traditional notation can easily be confused with PROLOG clauses. Therefore we call relational association rules query extensions, and use a new notation.

Consider again query $Q_1$, and assume we have a method to compute that this query holds for 25% of our customers (we explain this method in the next section):

```
?- customer(X), parent(X,Y), buys(Y,cola).
```

i.e., “25% of our customers are parents of children who buy cola” (Allen).

We may also have statistics on another query $Q_2$:

```
?- customer(X), parent(X,Y).
```

i.e., “75% of our customers are parents” (Allen, Bill, and Carol).

In the original itemset framework we would construct from the combination of two itemsets

\[
\{\text{customer, parent, child, cola\_buyer}\}
\]

\[
\text{FREQ: 0.25}
\]

\[
\{\text{customer, parent, child}\}
\]

\[
\text{FREQ: 0.75}
\]

an association rule

\[
\{\text{customer, parent, child}\} \rightarrow \{\text{cola\_buyer}\}
\]

\[
\text{FREQ: 0.25}
\]

\[
\text{CONF: } \frac{0.25}{0.75} = 0.33
\]

where \text{CONF} is the confidence of the rule, or the conditional probability that item cola\_buyer is present, given that items customer, parent, and child are. In a similar fashion, we would like to obtain from the combination of PROLOG queries $Q_1$ and $Q_2$ a relational association rule with 33\% confidence.

Let us first consider two solutions that are obvious but not correct. The first obvious choice would be to construct a PROLOG clause:

```
\text{buys}(Y,cola) :- customer(X), parent(X,Y).
```

However, as a property of customers, this clause should be read as “if a customer has a child, then this child will buy cola”. Notice Allen does not meet this rule: his child Carol does not buy cola. In fact, Diana is the only customer who does meet this rule, and then only trivially; she is not a parent. Depending on whether we want to count these trivial cases, the confidence of the clause would be 0%
or 25%, but never 33% as required. Clearly, PROLOG clauses do not correspond to the relational association rules we are looking for.

In a second attempt we use a format close to the original one:

\[
?- \text{customer}(X), \text{parent}(X, Y) \rightarrow ?- \text{buys}(Y, \text{cola}).
\]

The question then is how to interpret this formula. The right hand side could be interpreted as “someone buys cola”, which holds for all customers, much like “the moon circles the earth”.

The following is a first correct, albeit lengthy, version:

\[
?- \text{customer}(X), \text{parent}(X, Y) \rightarrow ?- \text{customer}(X), \text{parent}(X, Y), \text{buys}(Y, \text{cola}).
\]

i.e., “if a customer has a child, then that customer also has a child that buys cola”. Notice that in this case the rule does hold for Allen: he has a child, for instance Carol, and he also has a child that buys cola, that is Bill. If we ignore the customers without children, for whom the rule holds trivially, then we find indeed that its confidence equals 33%.

Finally, we introduce a new notation for the above rule that avoids repetition of conditions, and thus preserves the brevity of the original association rules. Like the above rule, query extension $QE_1$

\[
?- \text{customer}(X), \text{parent}(X, Y) \rightarrow \text{buys}(Y, \text{cola}).
\]

FREQ: 0.25
CONF: 0.33

should be read as “if a customer has a child, then that customer also has a child that buys cola”.

Formally, a query extension is an existentially quantified implication

\[
?- l_1, \ldots, l_m \rightarrow ?- l_1, \ldots, l_{m+1}, \ldots, l_n.
\]

with $1 \leq m < n$. By their definition, query extensions are closely connected to queries, actually they are two queries, where one is longer than \text{-extends}-the other. To avoid confusion with clauses (which are also implications) and to shorten notation we write this query extension as

\[
?- l_1, \ldots, l_m \rightarrow l_{m+1}, \ldots, l_n.
\]

We call query

\[
?- l_1, \ldots, l_m.
\]

the \textit{body} and the subquery

\[l_{m+1}, \ldots, l_n.
\]

the \textit{head} of the query extension.

It should be stressed that in the case of query extensions, the head does not correspond to the conclusion (as with clauses). Following standard terminology, we rather look at the unshortened notation, and call query

\[
?- l_1, \ldots, l_m, l_{m+1}, \ldots, l_n.
\]
the conclusion of the query extension.

To simplify the discussion, and circumvent the pitfalls of negation, we will restrict ourselves to so-called range-restricted queries, and query extensions. A range-restricted query is a query in which all variables that occur in negative literals also occur in at least one positive literal. A range-restricted query extension is a query extension such that both the body and the conclusion are range-restricted queries. The patterns below marked with * are not range-restricted (the responsible Y variable is shown in bold), the unmarked patterns are range-restricted.

Queries:

?- buys(X,pizza), ¬friend(X,Y), buys(Y,cola).
*?- buys(X,pizza), ¬friend(X,Y).

Query extensions:

?- buys(X,pizza), buys(Y,cola) ⇒ ¬friend(X,Y).
*?- buys(X,pizza) ⇒ ¬friend(X,Y).
*?- buys(X,pizza), ¬friend(X,Y) ⇒ buys(X,cola).

3 Evaluation measures

3.1 Frequency and confidence

The frequency of a query extension QE is defined as the frequency of the query that makes up the conclusion of QE. The confidence of a query extension QE is defined as the frequency of the conclusion of QE divided by the frequency of the body of QE. Both evaluation metrics are copied from the association rule literature. In this section we concentrate on how to compute the frequency of queries, which is at the heart of both statistics.

Let us reconsider the following query:

\[ Q_1 : ?- \text{customer}(X), \text{parent}(X,Y), \text{buys}(Y,cola). \]

Query \( Q_1 \) can be interpreted as a boolean attribute \( Q_1^A \) of customers: each customer either is or is not “parent of a cola buyer”. To find out for a particular customer, we substitute the variable \( X \) in \( Q_1 \) with the customer’s identifier, and evaluate against the database. For instance, query

\[ ?- \text{customer}(allen), \text{parent}(allen,Y), \text{buys}(Y,cola). \]

i.e., “customer allen is parent of a cola buyer Y”, succeeds with \( Y = \text{bill} \). Hence, attribute \( Q_1^A \) has value 1 for customer allen. Observe substitutions of \( X \) with the three remaining customer names bill, carol, diana all result in failing queries, such that for these customers attribute \( Q_1^A \) has value 0.

In general, to cast a PROLOG query \( Q \) as an attribute \( Q^A \) we need two things:

1. a separate relation in the database with example identifiers; in our example relation \text{CUSTOMER} with identifiers allen, bill, carol, diana; and
2. the atom corresponding to this relation (here $\text{customer}(X)$) has to occur in query $Q$.

We then define the frequency of a query $Q$ as the sum of value $Q^4$ for all examples, i.e., the number of example identifiers for which query $Q$ succeeds. Since we usually report the relative frequencies, we divide this number by the total number of examples.

Given the above interpretation of conjunctive queries as boolean attributes, and given four conjunctive queries with the obligatory $\text{customer}(X)$ atom:

\[ Q_1: \text{?- customer}(X), \text{parent}(X, Y), \text{buys}(Y, \text{cola}). \]
\[ Q_2: \text{?- customer}(X), \text{parent}(X, Y). \]
\[ Q_3: \text{?- customer}(X), \text{parent}(X, Y), \text{buys}(Y, \text{wine}). \]
\[ Q_4: \text{?- customer}(X), \text{buys}(X, Y). \]

Table 3 contains the attribute-value description with frequencies, of our PROLOG database $D$.

\[
\begin{array}{|c|c|c|c|c|}
\hline
\text{ID} & Q_1 & Q_2 & Q_3 & Q_4 \\
\hline
\text{allen} & 1 & 1 & 0 & 1 \\
\text{bill} & 0 & 1 & 0 & 1 \\
\text{carol} & 0 & 1 & 0 & 0 \\
\text{diana} & 0 & 0 & 0 & 1 \\
\hline
\text{FREQ} & 0.25 & 0.75 & 0 & 0.75 \\
\hline
\end{array}
\]

Table 3. Attribute-value description, with frequencies, of PROLOG database $D$.

In our framework the obligatory $\text{customer}(X)$ atom is essential, as it determines what is counted. Each binding of the variables in this atom uniquely identifies an entity. Therefore, we call $\text{customer}(X)$ the key parameter of the relational pattern discovery task. This parameter has been absent from previous formulations of the frequent pattern discovery task because it has always followed unambiguously from the application context.

The original absence of the key parameter can be traced back to the limited knowledge representation framework in which frequent pattern discovery has traditionally been casted. Reconsider the canonical market basket problem. There is no confusion possible about the object of counting: market baskets (also called transactions). Compare this to our customers database. We can do something similar and count baskets. But we might as well focus on customers -- as we have done above -- or even children of customers. With the key parameter we can change this focus while leaving the other inputs, in particular the database, untouched.

Before we move on a third evaluation measure, let us once more establish the link with relational database terminology. In SQL syntax the absolute frequency
of $Q$ with respect to a relational database can be obtained with the following query, inspired by [18, 5]:

```sql
SELECT count(distinct *)
FROM   SELECT fields that correspond to the variables in key
       FROM   relations in $Q$
       WHERE  conditions expressed in $Q$
```

For instance, for query $Q_1$, the following SQL query

```sql
SELECT count(distinct *)
FROM   SELECT Customer.Id
       FROM   Customer, Parent, Buys
       WHERE  Customer.Id = Parent.IdSr
       AND    Parent.IdJr = Buys.Id
       AND    Buys.Item='cola'
```

would return absolute frequency 1.

### 3.2 Deviation

We sometimes use a third quality criterion for query extensions which is less common in the association rule literature: the deviation label on a query extension indicates the unusualness of the dependency between the body and the conclusion. Essentially, we measure the statistical significance of the confidence with one-tailed test using the overall frequency of the head as the null hypothesis. In the rest of this section we explain this evaluation measure in more detail.

Consider the following query extension $QE_2$:

```
?- customer($X$), parent($X,Y$) => buys($X,pizza$).
FREQ: 0.25
CONF: 0.75
```

It is not clear from these statistics whether there is anything special about customers that are parents. It might well be that, overall, about three quarters of the customers buy pizza. In that case, the rule $QE_2$ may not be worth noticing\(^1\). If on the other hand on average as much as 99% or as few as 1% of the customers turn out to buy pizza, customer-parents can be considered to deviate from the expected as they buy unusually little or lot of pizza\(^2\).

Various deviation measures have been proposed in a data mining context, e.g., in EXPLORA [16, 17] and Midos [38]. We use the binomial distribution to measure the deviation of the confidence of a rule from the mean.

---

\(^1\) A difference of say 71%-75% may still be “important” if the number of observed orders is very large. Our deviation measure should (and will) account for this.

\(^2\) The inverse of the previous comment holds here. If only say four customers are observed, even a difference 33%-75% may be nonessential.
Assume an experiment that consists of \( N \) independent trials, where in each trial some event either happens or fails to happen. Let \( p \) be the probability that the event will happen in a single trial, and \( q = 1 - p \) be the probability that the event will fail to happen in a single trial. We can then compute the probability that the event will happen in exactly \( X \) trials (and fail to happen in \( N - X \) trials) with the following probability function:

\[
bino(p, N, X) = \binom{N}{X} p^X q^{N-X}
\]

Since \( \sum_{X=0}^{N} bino(p, N, X) = 1 \), function \( bino \) defines a discrete probability distribution, called the binomial distribution (or Bernoulli distribution) for \( X \).

Given \( N \) and \( p \), the probability that an event will happen in \( Y \) or more trials is \( \sum_{X=Y}^{N} bino(p, N, X) \). If \( Y \) is greater than the mean \( pN \), this probability is less than 0.5, and we say result \( Y \) deviates positively from mean \( pN \). The closer probability \( \sum_{X=Y}^{N} bino(p, N, X) \) approaches 0, the higher the positive deviation.

Given \( N \) and \( p \), the probability that an event will happen in \( Y \) or less trials is \( \sum_{X=0}^{Y} bino(p, N, X) \). If \( Y \) is less than the mean \( pN \), this probability is less than 0.5, and we say result \( Y \) deviates negatively from mean \( pN \). The closer probability \( \sum_{X=0}^{Y} bino(p, N, X) \) approaches 0, the higher the negative deviation.

We now translate these definitions to the present context. The experiment consists in repeatedly drawing an object matched by the body of the query extension where \( N \) is the absolute frequency of the body. Following [16, 17], we call this set the subgroup. In \( QE_2 \), the subgroup would be the customers with kids, and since there are 3 of them, \( N = 3 \).

Again following [16, 17], we call the set of objects matched by the head of the query extension the target group. Each time we draw an element from the subgroup, this element can either be in or out the target group. The probability \( p \) that event “in target group” will happen in a single trial “draw from subgroup” is assumed to be identical to the probability that event “in target group” will happen in a single trial “draw from total population”. That is, \( p \) is set to the frequency of the head of the query extension. In \( QE_2 \) this is the fraction of customers that buy pizza, which happens to be 0.5.

We then observe the result of the experiment. The number of successful trials corresponds to the number of elements in the intersection of subgroup and target group, i.e., to the absolute frequency of the query extension. In \( QE_2 \), there is one customer that has kids and buys pizza.

Finally, we can compare the outcome of the experiment with the assumed mean and measure to what extent it deviates (positively or negatively). In \( QE_2 \), we expect \( 0.5 \times 3 = 1.5 \) elements in the intersection. Obviously, our observed result 1 deviates negatively. We quantify this deviation as follows: if we draw 3 elements from the set of all customers, then the probability that this selection will contain 1 or less customers who buy pizza is \( \sum_{X=0}^{1} bino(0.5, 3, X) = 0.5 \). This means that, if we repeat the experiment 100 times, a result of 1 or less is bound to come up in about 50 cases. In other words, rule \( QE_2 \) is not at all remarkable.
It should be stressed at this point that it is necessary to be very careful with any statistical interpretation of the deviation test. In statistical decision theory, the 0.5 deviation would be interpreted as the significance level of the hypothesis that the proportion of target group elements is smaller in the subgroup than in the total population. The problem is that in frequent query discovery we look at many target groups. Due to what statisticians call the multiplicity effect [29], some of them are bound to come up seemingly “significant”. Therefore we use the deviation measure rather as a ranking criterion. Its purpose is to filter away uninteresting patterns, rather than to corroborate interesting ones. The closer the deviation is to 0.5 the less interesting it is.

4 Declarative language bias

The representation of patterns as PROLOG queries requires a formalism to constrain the query language \( \mathcal{L} \) to a set of meaningful and useful patterns. With association rules the definition of \( \mathcal{L} \) is straightforward: \( \mathcal{L} \) is simply \( 2^I \), the collection of all subsets of the set \( I \) of items. Srikan, Vu, and Agrawal [33] describe a technique to impose and exploit user-defined constraints on combinations of items, but otherwise the definition of \( \mathcal{L} \) has received little attention in the frequent pattern discovery literature. In inductive logic programming, on the other hand, this issue has been studied extensively in the subfield of declarative language bias. This is motivated by huge, often infinite, search spaces, that require a tight specification of patterns worth considering. Several formalisms have been proposed for adding language bias information in a declarative manner to the search process (for an overview, see [1, 27]). Our formalism, WARMODE, is an adaptation to WARMR of the RMODE format developed for TILDE [6] which, in turn, is based on the formalism originally developed for PROGOL [25].

We will demonstrate later how the WARMODE notation can be used to constrain \( \mathcal{L} \) to some interesting classes of patterns. The required subset of WARMODE is described below.

4.1 The WARMODE basics

Let us first look at the simple case where \( \mathcal{L} \) contains no variables, i.e., only ground queries are allowed. Under these circumstances, the WARMODE notation extends the straightforward \( \mathcal{L} = 2^I \) bias to PROLOG queries: given a set \( \text{Atoms} \) of ground atoms, the language \( \mathcal{L} \) consists of \( 2^{\text{Atoms}} \), i.e., of all possible combinations of the atoms. For example,

\[
\text{Atoms} = \{ \text{parent}(a,b), \text{buys}(a,\text{cola}) \}
\]

defines \( \mathcal{L} = 2^{\text{Atoms}} = \)

\[
\{ \varnothing, \text{true}, \varnothing, \text{parent}(a,b), \varnothing, \text{buys}(a,\text{cola}), \varnothing, \text{parent}(a,b), \text{buys}(a,\text{cola}) \} \]
When variables are allowed in $\mathcal{L}$, the power set idea can be extended to a set of literals, as done, for instance, in [36] and [10]. However, this solution is inconvenient for two reasons. First, we might want to define infinite languages. For instance, in our running examples, we might want to go up the family tree and allow queries

?- parent(X1, X2), parent(X2, X3), parent(X3, X4), ...

of arbitrary length. Thereto, an atom with predicate symbol in $\text{Atoms}$ (e.g., $\text{parent}$) should be allowed several times in the query and not just once as in $\text{Atoms}$. Second, we do not want to control the exact names of the variables in the query, as we do for constants, but rather the sharing of names between variables. For example, query

?- rectangle(Width, Height), Width<Height.

is equivalent to query

?- rectangle(X, Y), X<Y.

but not to query

?- rectangle(Width, Height), X<Y.

In the \textsc{Warmode} framework, non-ground atoms in $\text{Atoms}$ are allowed to occur multiple times in the query, as long as their variables obey the so-called \textit{mode constraints}. These are declared for each variable argument of each atom by means of three \textit{mode-labels} $+$, $-$, and $\pm$:

$+$ the variable is strictly \textit{input}, i.e. bound before the atom is called

$-$ the variable is strictly \textit{output}, i.e. bound by the atom

$\pm$ the variable can be both input and/or output, i.e. anything goes

In our approach the atoms of the query are evaluated one by one, following an ordering that is consistent with the mode declarations. The intuition is that the evaluation of some atoms, such as $X < Y$ in the example above, presupposes the binding of certain \textit{input} variables to a constant. On the other hand some atoms, e.g. $\text{rectangle}(\text{Width}, \text{Height})$, are allowed or required to introduce new \textit{output} variables bound during evaluation of the atom. A query is then mode conform if an ordering of atoms exists such that every input variable occurs in one of the previous atoms, and no output variable does. Some examples of queries that are consistent and inconsistent with mode declarations are listed in Table 4. Throughout this chapter we use the notational convention that the atoms of a query are evaluated from left to right.

4.2 Typing in \textsc{Warmode}

Additional constraints on the sharing of variable names can be imposed via type declarations. The \textsc{Warmode} convention is to append these to the mode declarations in $\text{Atoms}$. A query is then type conform if and only if arguments that share a variable name either have identical types or at least one of the arguments is untyped. For example, with mode and type declarations
**Table 4.** Examples of Warmode definitions and queries (dis)allowed in the corresponding languages.

Atom \_s = \{\text{parent}(\_p,\_a), \text{buys}(\_p,\_a)\},

query

\?- \text{parent}(X, Y), \text{buys}(Y, \_a).

is in \_L, but not

\?- \text{parent}(X, Y), \text{buys}(X, \_a).

since the first arguments of predicates parent and buys have incompatible types \_p, \_a. Notice the difference between constants (\_a) and types (\_p, \_a) in declarations **Atoms:** types are preceded by a mode label and constants are not.

### 4.3 The Warmode key

As we have seen, the frequent query discovery task requires the specification of a key atom which is obligatory in all queries. Within Warmode notation this is done with key = KeyAtom, where KeyAtom is a mode and type declaration as defined above. Obviously, the key atom declaration should not contain any + mode labels.

For an example of a non-ground language, consider

key = parent(\_p,\_p)

and

**Atoms** = \{\text{buys}(\_p,\_a)\}.

These declarations together define \_L =

\{ \?- \text{parent}(X, Y), \\
?- \text{parent}(X, Y), \text{buys}(X, \_a), \\
?- \text{parent}(X, Y), \text{buys}(Y, \_a), \\
?- \text{parent}(X, Y), \text{buys}(X, \_a), \text{buys}(X, \_a), \\
?- \text{parent}(X, Y), \text{buys}(X, \_a), \text{buys}(Y, \_a), \\
\ldots \}

Notice the fourth query in the above list is logically redundant.
4.4 Logical redundancy and WARMODE

The WARMODE notation is only meant to capture application-specific constraints on $\mathcal{L}$. These are usually supplemented by a number of general constraints hard-wired in the data mining algorithm in which WARMODE is embedded. Logical non-redundancy is such an application-neutral and algorithm-specific constraining principle. For instance, the last WARMODE definition in Table 3 allows

? $p(X, a), q(X), p(Y, a), q(Y), q(Y)$.

which is logically equivalent to the shorter

? $p(X, a), q(X)$.

Within WARM the first pattern would be filtered away in the candidate generation phase, as explained below in Section 5.2.

5 Query (extension) discovery with WARMR

Design of algorithms for frequent pattern discovery has turned out to be a popular topic in data mining (for a sample of algorithms, see [2, 3, 19, 30, 35]). Almost all algorithms are on some level based on the same idea of levelwise search, known in data mining from the A PRIORI algorithm [3]. We first review the generic levelwise search method and its central properties and then introduce the algorithm WARMR [10] for finding frequent queries. To conclude this section, we recall how this method fits in the two-phased discovery of frequent and confident rules.

5.1 The levelwise algorithm

The levelwise algorithm [22] is based on a breadth-first search in the lattice spanned by a specialization relation $\preceq$ between patterns, cf. [24], where $p_1 \preceq p_2$ denotes “pattern $p_1$ is more general than pattern $p_2$”, or “$p_2$ is more specific than pattern $p_1$”.

The method looks at a level of the lattice at a time, starting from the most general patterns. The method iterates between candidate generation and candidate evaluation phases: in candidate generation, the lattice structure is used for pruning non-frequent patterns from the next level; in the candidate evaluation phase, frequencies of candidates are computed with respect to the database. Pruning is based on monotonicity of $\preceq$ with respect to frequency: if a pattern is not frequent then none of its specializations is frequent. So while generating candidates for the next level, all the patterns that are specializations of infrequent patterns can be pruned. For instance, in the A PRIORI algorithm for frequent itemsets, candidates are generated such that all their subsets (i.e., generalizations) are frequent.

The levelwise approach has two crucial useful properties:
Algorithm 1: Warmr

Inputs: Database \( \mathbf{r} \); WARMOde language \( \mathcal{L} \) and key; threshold \( \text{minfreq} \)

Outputs: All queries \( Q \in \mathcal{L} \) with frequency \( \geq \text{minfreq} \)

1. Initialize level \( d := 1 \)
2. Initialize the set of candidate queries \( Q_1 := \{ \text{key} \} \)
3. Initialize the set of infrequent queries \( \mathcal{I} := \emptyset \)
4. Initialize the set of frequent queries \( \mathcal{F} := \emptyset \)
5. While \( Q_d \) not empty
6. Find frequency of all queries \( Q \in Q_d \) using Warmr - Eval
7. Move the queries \( Q \) with frequency below \( \text{minfreq} \) to \( \mathcal{I} \)
8. Update \( \mathcal{F} := \mathcal{F} \cup Q_d \)
9. Compute new candidates \( Q_{d+1} \) from \( Q_d \), \( \mathcal{F} \) and \( \mathcal{I} \) using Warmr - Gen
10. Increment \( d \)
11. Return \( \mathcal{F} \)

- Assuming all candidates of a level are tested in single database pass, the database is scanned at most \( d + 1 \) times, where \( d \) is the maximum level (size) of a frequent pattern. This is an important factor when mining large databases.
- The time complexity is in practice linear in the product of the size of the result times the number of examples, assuming matching patterns against the data is fast.

5.2 The Warmr algorithm

Algorithm 1, steps (5-10), shows Warmr’s main loop as an iteration of candidate evaluation in step (6) and candidate generation in step (9). The manipulation of set \( \mathcal{I} \) of infrequent queries in steps (3) and (7) is necessary for the generation phase. This is discussed below, together with some other features that distinguish Warmr from apriori.

Specialization relation The subset specialization relation used in most frequent pattern discovery settings can in some restricted cases also be used for structuring a space of prolog queries, as done in [36,37] and [10]. In general however, the subset condition is too strong. For instance, we would like to consider query

\[ ?- \text{likes}(Z,Z), \text{buys}(Z, \text{cola}). \]

as a specialization of query

\[ ?- \text{likes}(X, Y), \text{likes}(Y, X). \]
although the first query is not a subset of the second.

The most obvious general-purpose definition of the subsumption relation is based on logical implication: \( \text{Query}_1 \subseteq \text{Query}_2 \) if and only if \( \text{Query}_2 \models \text{Query}_1 \). Logical implication could detect for instance that \( \text{?- like}(X,Y) \), \( \text{like}(Y,X) \) is a generalization of \( \text{?- like}(Z,Z), \text{buys}(Z,\text{cola}) \). However, due to the high computational cost of the logical implication check, inductive logic programming algorithms often rely on a stronger variant coined \( \theta \)-subsumption by Plotkin [28].

\( \text{Query}_1 \theta \)-subsumes \( \text{Query}_2 \) (abbreviated to: \( \text{Query}_1 \theta \text{ts Query}_2 \) if and only if there exists a (possibly empty) substitution of the variables of \( \text{Query}_2 \), such that every atom of the resulting query occurs in \( \text{Query}_1 \), i.e., \( \text{Query}_1 \supseteq \text{Query}_2 \theta \).

For instance,

\[
\text{?- like}(Z,Z) \), \text{buys}(Z,\text{cola}) \), \text{ts} \text{?- like}(X,Y) , \text{like}(Y,X).
\]

with \( \theta = \{X/Z, Y/Z\} \).

For all queries \( Q_1 \) and \( Q_2 \):

\[
Q_1 \supseteq Q_2 \iff Q_2 \text{ ts } Q_1 \\
Q_2 \text{ ts } Q_1 \iff Q_1 \models Q_2
\]

Consider for instance queries:

\[
\text{?- like}(Z,Z) \), \text{buys}(Z,\text{cola}) \), \text{ts} \text{?- like}(Z,Z).
\]

\[
\text{?- like}(Z,Z) \), \text{buys}(Z,\text{cola}) \), \text{ts} \text{?- like}(X,Y) , \text{like}(Y,X).
\]

\[
\text{?- rich}(X), \text{not rich}(f(f(X))) \), \text{ts} \text{?- rich}(X), \text{not rich}(f(X)).
\]

The examples above illustrate \( \theta \)-subsumption is weaker than the subset relation, but stronger than logical implication. To understand the third example it helps to interpret \( f(X) \) as father of \( X \). Then the formula says that if there is rich person whose grandfather is not rich, it follows that there is a rich person (the same as before or his/her father) whose father is not rich.

---

\(^3\) Plotkin’s definition as used everywhere else in the ILP literature, including other chapters in this book, applies to clauses, i.e., disjunctions. We have adapted it to queries, which are conjunctions. More details on this adaptation can be found in [9], pp. 80–83.
Algorithm 2: \textsc{Warmr-Eval}

\begin{algorithm}
\begin{algorithmic}
\STATE \textbf{Inputs:} Database $r$; set of queries $Q$;Warmr key
\STATE \textbf{Outputs:} The frequencies of queries $Q$

1. For each query $Q_j \in Q$, initialize frequency counter $q_j := 0$
2. For each answer $\theta_k$ resulting from the execution of query $?^* \text{- key}$ w.r.t. database $r$:
   \begin{enumerate}
   \item Isolate the relevant fraction of the database $r_k \subseteq r$
   \item For each query $Q_j \in Q$, do the following:
   \begin{enumerate}
   \item If query $Q_j \theta_k$ succeeds w.r.t. $r_k$, increment counter $q_j$
   \end{enumerate}
   \end{enumerate}
\end{algorithmic}
\end{algorithm}

Candidate evaluation Algorithm Warmr-Eval adapts our earlier notion of frequency of a single query $Q$ to the levelwise approach, which matches a set of patterns against one example at a time. The example is here represented by $\theta_k$, the substitution for the key variables obtained in step (2) by running $?^* \text{- key}$ against the database. The algorithm, in step (2.b), applies a fixed substitution $\theta_k$ to the subsequent queries $Q_j$ drawn from $Q$, and increments an associated counter $q_j$ in case $Q_j \theta_k$ succeeds with respect to the database.

If we execute the latter evaluation with respect to $r$, we still need one pass through the database per query, instead of one pass per level. The solution adopted in Warmr-Eval, step (2.a), is based on the assumption that there exists a relatively small subset $r_k$ of $r$, such that the evaluation of any $Q \theta_k$ only involves tuples from $r_k$. Readers familiar with relational database technology might notice a similar assumption underlies the definition of a cluster index. In many cases $\{ r_k \}$ is a partition on $r$. The algorithm then makes a single pass through the data in the sense that the key values $\theta_k$ are retrieved one by one, the subsequent subdatabases $r_k$ are activated once in (2.a), and all queries are evaluated locally with respect to $r_k$ in (2.b). An experimental evaluation of this localisation of information in a related data mining task can be found in [7].

Consider as an example our database $D$ with customer information. Each subset $r_k$ of this database would contain a fact $\text{customer}(cid_k) \leftarrow$ and zero or more facts $\text{parent}(\text{wid}_k, \text{atype}) \leftarrow$. This subset of the database indeed suffices for solving queries $Q \theta_k$ built with predicates customer and parent. But what about the facts of the form $\text{buys}(\text{person}, \text{item}) \leftarrow$? Since person might refer both to parent or child, these facts are relevant for many keys wid$_k$ and involved in solving queries in many examples $\theta_k$. As a consequence, they cannot be assigned to one $r_k$ exclusively. The solution is to either duplicate the information, or put these facts in a separate section of the database which is always activated.

Candidate generation To generate candidates, Warmr-Gen employs at step (2) a classical specialization operator under $\theta$-subsumption [28, 26]. A specialization operator $\rho$ maps queries $\in L$ onto sets of queries $\in 2^L$, such that for any Query1 and $\forall$ Query2 $\in \rho(\text{Query1})$, Query1 $\theta$-subsumes Query2. The op-
erator used in WARMR - GEN essentially adds one atom to the query at a time, as allowed by WARMODE declarations.

Mode and type declarations on variables may cause an atom to be added for the first time only deep down the lattice. For instance, in our running example, atom \( \text{buys}(Y,\text{cola}) \) should only be added once \( \text{parent}(X,Y) \) is in \( Q \). This complicates pruning significantly. We can no longer require that all generalizations of a candidate are frequent as some of the generalizations, such as

\[
?- \text{customer}(X), \text{buys}(Y,\text{cola}).
\]

might simply not be in the language of admissible patterns. Instead, WARMR - GEN at step (2.i) requires candidates not to be \( \theta \)-subsumed by any infrequent query. In step (2.ii), we also require that candidates and frequent queries are mutually inequivalent under \( \theta \)-subsumption. This way a potentially huge set of redundant solutions is eliminated.

**Algorithm 3 :** WARMR - GEN

**Inputs:** WARMODE language \( \mathcal{L} \); infrequent queries \( \mathcal{I} \); frequent queries \( \mathcal{F} \);
frequent queries \( \mathcal{Q}_d \) for level \( d \)

**Outputs:** Candidate queries \( \mathcal{Q}_{d+1} \) for level \( d+1 \)

1. Initialize \( \mathcal{Q}_{d+1} := \emptyset \)
2. For each query \( Q_j \in \mathcal{Q}_d \), and for each immediate specialization \( Q'_j \in \mathcal{L} \) 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 \( \in \mathcal{Q}_{d+1} \cup \mathcal{F} \)

---

5.3 Two-phased discovery of frequent and confident rules

Frequent patterns are commonly not considered useful for presentation to the user as such. Their popularity is mainly based on the fact that they can be efficiently post-processed into rules that exceed given confidence and frequency threshold values. The best known example of this two-phased strategy is the discovery of association rules [2], and closely related patterns include episodes [23] and sequential patterns [4]. For all these patterns, the threshold values offer a natural way of pruning weak and rare rules.

As observed in [2] for association rules, confident and frequent query extensions can be found effectively in two steps. In the first step one determines the set of all frequent queries, and in the second produces query extensions whose confidence exceeds the given threshold. As explained above, query extensions are in fact couples of queries where one extends the other. The construction of
query extensions therefore simply consists of finding all such couples in the list of frequent queries. Formally, we look for couples of queries \((B, C)\) such that conclusion \(C\) \(\theta\)-subsumes body \(B\). As an example, reconsider the combination of two queries \((Q_2, Q_1)\):

\[
Q_2: \text{?- customer}(X), \text{parent}(X, Y).
\]

FREQ: 0.75

\[
Q_1: \text{?- customer}(X), \text{parent}(X, Y), \text{buys}(Y, cola).
\]

FREQ: 0.25

where \(Q_1\) extends/\(\theta\)-subsumes \(Q_2\), to query extension \(QE_1\):

\[
\text{?- customer}(X), \text{parent}(X, Y) \sim \text{buys}(Y, cola).
\]

FREQ: 0.25

CONF: 0.33

If all frequent queries and their frequencies are known as a result of the first step, then this easy second step is guaranteed to output all frequent and confident query extensions. This is illustrated with a sample run of \textsc{Warmr} on the customer database in the next section.

6 A sample run

In this section we show \textsc{Warmr} at work on the \textit{customer} example used throughout this chapter.

This application is defined by two input files: the data and the pattern language. The \textsc{Prolog} database for this application is presented in Table 2, the language bias declarations in Table 5. Frequency and confidence thresholds are set to 10\%, the default values in \textsc{Warmr}.

\begin{table}[h]
\begin{center}
\begin{tabular}{l}
\hline
\texttt{warmode_key(customer(-)).} \\
\texttt{warmode(parent(+,-)).} \\
\texttt{warmode(buys(+,cola)).} \\
\texttt{warmode(buys(+,pizza)).} \\
\texttt{warmode(buys(+,wine)).} \\
\hline
\end{tabular}
\end{center}
\caption{Input file with declarative language bias settings in \textsc{Warmode} format.}
\end{table}

Given these inputs, \textsc{Warmr} outputs 25 frequent queries. These are all listed, with their frequency in Table 6. Notice we have omitted the \textit{customer}(A) prefix to save space.

In a second phase, these 25 queries are used to generate relational association rules, or rather, query extensions. In total, \textsc{Warmr} outputs 93 rules with frequency and confidence above the 10\% threshold. Table 7 lists the 35 query extensions with two literals in the head; the omitted \textit{customer}(A) and \textit{parent}(A,B), \textit{buys}(A,cola), \textit{buys}(A,pizza) or \textit{buys}(A,wine).
<table>
<thead>
<tr>
<th>Frequent query: ?- customer(A),...</th>
<th>freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent(A,B).</td>
<td>0.75</td>
</tr>
<tr>
<td>buys(A,cola).</td>
<td>0.25</td>
</tr>
<tr>
<td>buys(A,pizza).</td>
<td>0.50</td>
</tr>
<tr>
<td>buys(A,wine).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), parent(B,C).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), buys(B,cola).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), buys(A,cola).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), buys(B,pizza).</td>
<td>0.50</td>
</tr>
<tr>
<td>parent(A,B), buys(A,pizza).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), buys(A,wine).</td>
<td>0.25</td>
</tr>
<tr>
<td>buys(A,cola), buys(A,pizza).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), parent(B,C), buys(B,cola).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), parent(B,C), buys(C,pizza).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), parent(B,C), buys(B,pizza).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), parent(B,C), buys(A,wine).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), buys(B,cola), buys(B,pizza).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), buys(A,cola), buys(A,wine).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), buys(A,cola), buys(A,pizza).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), buys(B,pizza), buys(A,wine).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), parent(B,C), buys(B,cola), buys(B,pizza).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), parent(B,C), buys(B,cola), buys(A,wine).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), parent(B,C), buys(B,pizza), buys(A,wine).</td>
<td>0.25</td>
</tr>
<tr>
<td>parent(A,B), buys(B,cola), buys(B,pizza), buys(A,wine).</td>
<td>0.25</td>
</tr>
</tbody>
</table>

Table 6. All queries in the customer domain with frequency above 0.10 (prefix ?- customer(A) omitted).
<table>
<thead>
<tr>
<th>frequent and confident query extension: (?- \text{customer}(A))</th>
<th>freq</th>
<th>conf</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent(A,B) (\sim) parent(B,C).</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>parent(A,B) (\sim) buys(B,cola).</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>parent(A,B) (\sim) buys(A,cola).</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>parent(A,B) (\sim) buys(B,pizza).</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>parent(A,B) (\sim) buys(A,pizza).</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>parent(A,B) (\sim) buys(A,wine).</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>parent(A,B) (\sim) parent(B,C), buys(B,cola).</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>parent(A,B) (\sim) parent(B,C), buys(B,pizza).</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>parent(A,B) (\sim) parent(B,C), buys(A,wine).</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>parent(A,B) (\sim) buys(B,cola), buys(B,pizza).</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>parent(A,B) (\sim) buys(B,cola), buys(A,wine).</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>parent(A,B) (\sim) buys(B,cola), buys(B,pizza), buys(A,wine).</td>
<td>0.25</td>
<td>0.33</td>
</tr>
<tr>
<td>buys(A,cola) (\sim) parent(A,B).</td>
<td>0.25</td>
<td>1.00</td>
</tr>
<tr>
<td>buys(A,cola) (\sim) buys(A,pizza).</td>
<td>0.25</td>
<td>1.00</td>
</tr>
<tr>
<td>buys(A,pizza) (\sim) parent(A,B).</td>
<td>0.25</td>
<td>0.50</td>
</tr>
<tr>
<td>buys(A,pizza) (\sim) buys(A,cola).</td>
<td>0.25</td>
<td>0.50</td>
</tr>
<tr>
<td>buys(A,wine) (\sim) parent(A,B).</td>
<td>0.25</td>
<td>1.00</td>
</tr>
<tr>
<td>buys(A,wine) (\sim) parent(A,B), parent(B,C).</td>
<td>0.25</td>
<td>1.00</td>
</tr>
<tr>
<td>buys(A,wine) (\sim) parent(A,B), buys(B,cola).</td>
<td>0.25</td>
<td>1.00</td>
</tr>
<tr>
<td>buys(A,wine) (\sim) parent(A,B), buys(B,pizza).</td>
<td>0.25</td>
<td>1.00</td>
</tr>
<tr>
<td>buys(A,wine) (\sim) parent(A,B), parent(B,C), buys(B,cola).</td>
<td>0.25</td>
<td>1.00</td>
</tr>
<tr>
<td>buys(A,wine) (\sim) parent(A,B), parent(B,C), buys(C,pizza).</td>
<td>0.25</td>
<td>1.00</td>
</tr>
<tr>
<td>buys(A,wine) (\sim) parent(A,B), parent(B,C), buys(B,pizza).</td>
<td>0.25</td>
<td>1.00</td>
</tr>
<tr>
<td>buys(A,wine) (\sim) parent(A,B), buys(B,cola), buys(B,pizza).</td>
<td>0.25</td>
<td>1.00</td>
</tr>
<tr>
<td>buys(A,wine) (\sim) parent(A,B), parent(B,C), buys(B,cola), buys(B,pizza).</td>
<td>0.25</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Table 7. All query extension in the customer domain with 2-literal heads and with frequency and confidence above 0.10 (prefix \(?- \text{customer}(A)\) omitted).
Table 7 only shows frequency and confidence, not the deviation measure introduced above. Since the rules are discovered on a very small data sample, none of them are interesting according to this measure. This example does illustrate the importance of an additional ranking criterion: even with such a small database and language, about 100 rules are generated. In cases where this list gets much longer, association rules (relational or not) can no longer be considered summaries of the data, unless they are presented to the user in some reasonable order.

7 Discussion

We have presented a general PROLOG formulation of the frequent pattern discovery problem: find out which PROLOG queries in a user-defined language succeed frequently in a given database. These queries can then be combined into relational association rules, or rather query extensions, with frequency and confidence above the user-specified threshold. We outlined WARMODE, a declarative formalism for specifying the language bias, i.e., the search space \( \mathcal{L} \) of admissible or potentially interesting PROLOG queries. We also gave an algorithm, WARMR, for solving such tasks, and presented a sample run of WARMR on a small application.

In this chapter we have used a toy application which has the advantage that all examples can be verified by the reader. The scientific and commercial potential of WARMR and frequent query discovery is demonstrated in [12,11,9] where real world applications are discussed in telecommunications and biochemistry. In both cases the network structure of the data calls for the additional expressive power offered by WARMR.

WARMR, which is available for academic purposes upon request, is a flexible tool that can be used by both users and developers as an explorative data mining tool: pattern types can be modified in a flexible way, and thus a number of frequent pattern discovery settings can be easily experimented with without changes in the implementation. It is not inconceivable that for any setting addressed with WARMR, specialized algorithms can be developed that will outperform WARMR by several orders of magnitude. This is for instance the case with the market basket application and the APRIORI algorithm designed specifically for this task. However, a generic tool such as WARMR is complementary to these specialized algorithms and offers several advantages both to users and developers.

To the users WARMR offers mainly two types of flexibility. First, the user can jump from one frequent pattern discovery setting to another with just minor changes to the language bias and background knowledge. Individual pattern types that turn out to be of particular interest can then be mined in a second stage with specialized algorithms. An additional danger with using a specialized algorithm as a first approach is that any information which cannot be used within this method is bound to be ignored or even cut away in a preprocessing step.
A second type of flexibility comes with the possibility to add background knowledge. Background knowledge has at least two functions in the process of knowledge discovery in databases: it can be used to (1) add information in the form of general rules, but also (2) to change with minor effort the view on the data, without going through the typically laborious preprocessing of the raw data themselves. Again, once the experiments converge on some specific setting, efficiency can be cranked up by reorganization of the data into some very specific format.

On the other hand, for the developers of specialized algorithms WARMR can function as a benchmark, and as a verification/validation method: the special algorithm should run significantly faster, and produce the same output.

8 Acknowledgements

Luc Dehaspe is Fellow of the K.U.Leuven Research Council. Hannu T.T. Toivonen was at Rolf Nevanlinna Institute, University of Helsinki, while this work was done. He has been supported by the Academy of Finland.

The authors are grateful to Luc De Raedt and Heikki Mannila for comments on the paper and for many fundamental ideas and discussions, to Hendrik Blockeel, Bart Demoen, and Wim Van Laer for their share in the implementation of WARMR.

References