Metapattern Generation for Integrated Data Mining

Wei-Min Shen
Information Sciences Institute and
Computer Science Department
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
shen@isi.edu

Bing Leng
Inference Corporation
4th Fl, 8410 W Bryn Mawr Ave
Chicago, IL 60631
leng@inference.com

Abstract

Metapatterns (also known as metaqueries) have been proposed as a new approach to integrated data mining, and applied to several real-world applications successfully. However, designing the right metapatterns for a given application still remains a difficulty task. In this paper, we present a metapattern generator that can automatically generate metapatterns from new databases. By integrating this generator with the existing metapattern-based discovery loop, our system has now become both interactive and automatic. It can suggest new metapatterns for humans to choose and test, or pursue these metapatterns on its own. This ability not only makes the process of data mining more efficient and productive, but also provide a new method for unsupervised learning of relational patterns. We have applied this method to several simple databases and obtained some encouraging results.

Introduction

Metapatterns (also known as metaqueries) (Shen *et al.* 1995; Kero *et al.* 1995; Fu & Han 1995; Shen & Leng 1996) have been proposed as a new data mining approach to integrate induction, deduction, and human guidance. They are second-order expressions, such as

$$P(X,Y) \wedge Q(Y,Z) \Rightarrow R(X,Z),$$

where P, Q, and R are variables for predicates and X, Y, and Z are variables for objects. Metapatterns are used to control the discovery loop shown in Figure 1. For the deductive part of the loop, metapatterns outline data-collecting strategies and serve as the basis for the generation of specific queries. Queries are generated by instantiating the variables in the left-hand side of metapatterns with relevant table names and column names in the database of interest and then run against the database to collect relevant data. For example, one possible query instantiated from the above metapattern is:

$$parent(X,Y) \wedge brother(Y,Z) \Rightarrow uncle(X,Z).$$

The final results of discovered patterns are associated with probability factors to reflect how much support received from the underlying databases and provide handlers to deal with noise. For the inductive part of the loop, metapatterns serve as generic descriptions of classes of pattern to be discovered: A metapattern determines which inductive action to apply, and what format the final results should be in. Furthermore, since metapatterns are declarative expressions, they serve as a search control interface between humans and systems. By examing, selecting, and executing different metapatterns, human users can expand or contract the search space and change the search direction at will. For more detailed description of metapatterns, readers are encouraged to read (Shen *et al.* 1995) and (Shen & Leng 1996).

Although metapatterns are powerful tools for data mining and have been applied successfully to several real-world applications (Shen 1992; Shen *et al.* 1995; Fu & Han 1995), designing the right metapatterns for a given application is not an easy task. If a metapattern is too specific, then it may miss the interesting patterns. If a metapattern is too general, then it may exhaust the computing resources that are available. To generate the right metapatterns, one must not only understand the nature of the underlying data, but also analyze the patterns discovered from the previous metapatterns.

To illustrate how productive metapatterns are generated manually, consider our experience in the chemical research domain as an example. Following a suggestion by a chemist, we initially used a metapattern to find the relationship between a set of compounds that have different percentages of the ingredients 'A322' and 'B721' and their chemical properties. However, the patterns returned based on this metapattern did not show any trends. When we showed the results to the chemist, he discovered that these compounds also contained auxiliary chemicals that may affect the properties in a different way. Given this knowledge, we constrained the metapattern so that the compounds that had such auxiliary ingredients were not considered. Sure enough, the resulting patterns showed many clear trends. We can see from this example that if domain experts can directly interact with the system (i.e., metapatterns can be automatically generated and offered to them to choose and test), then the entire discovery process can be much more efficient and pro-

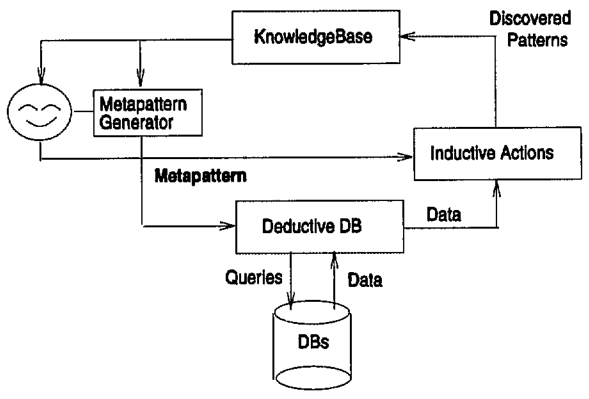

Figure 1: The Metapattern-Based Discovery Loop

ductive. The system should provide suggestions and feedback of metapatterns so the experts can discover new knowledge and try better metapatterns.

For this purpose, we have developed a metapattern generator for the metapattern-based discovery loop. Given a completely new database, the generator first identifies, by examining the types and ranges of the data columns, a set of ***significant connections*** among tables. Using these connections, a connection graph is then built and all the loops in this graph are found. These loops are then used to generate an initial set of ***transitivity metapatterns***. These initial metapatterns are then run against the database; based on the results, new metapatterns are generated dynamically by adding more constraints to the more plausible metapatterns. Since transitivity metapatterns are among the most general types of metapatterns, this top-down approach can generate many interesting metapatterns (and patterns) automatically. In fact, we have applied this approach to some of the well-known examples in the literature of supervised learning of relation patterns, and demonstrated that our system can learn these concepts without "supervision," that is, without requiring humans to pre-label the data as positive or negative examples of some pre-specified target concepts.

The Metapattern Generator

As shown in Figure 1, the metapattern generator will be placed in parallel with the human user and will generate metapatterns based on either the metaknowledge of the database (e.g., schema), or the patterns that are discovered with other metapatterns. Human users can interact with the generator by examining, selecting, and executing the metapatterns, and they can also create metapatterns by themselves as before. The motivation for having this generator is to give human users suggestions for new metapatterns so that the more expert users can get inspirations, and the less expert users can learn how to perform data mining in a particular domain by observation.

The Space of Metapatterns

In order to generate metapatterns, we must first understand the nature of the set of all possible metapatterns. In this paper, we consider Horn clause metapatterns and assume that the underlying databases are relational. Thus, a predicate variable in a metapattern can be bound to relational table names or built-in predicates (such as ***equal*** or ***greaterThan***), and an object variable can be bound to column names or constant values (such as the integer "20" or the language "English").

For a fixed length, metapatterns can be ordered from the most general to the most specific, depending on how many variables, either for predicates or for objects, are present. For example, among all metapatterns that have three binary (second-order) predicates, the most general ones include:

$$P(X,Y) \land Q(Y,Z) \Rightarrow R(X,Z) \qquad \text{(MP-1)}$$
$$P(X,Y) \land Q(X,Z) \Rightarrow R(X,W) \qquad \text{(MP-2)}$$
$$P(X,Y) \land Q(Y,Z) \Rightarrow R(X,W) \qquad \text{(MP-3)}$$

These metapatterns are the most general because they contain only variables. On the other hand, the most specific metapatterns corresponding to the general ones listed above (MP-1, MP-2, and MP-3) include:

authorOf('Orwell','AnimalFarm') ∧
writtenIn('AnimalFarm','English') → canWrite('Orwell','English')

likes('John',*tools*) ∧ hasHobby('John','Carpenter') →
ownsOne('John',*hammer*)

livesIn('Mary','House1') ∧ costs('House1',900893) →
Income('Mary','High')

All the variables in these metapatterns are bound to specific table names (e.g., likes), column names (e.g., *tools*), or constant values (e.g. 'John'). We define a *family* of metapatterns to be the set of all metapatterns of length n. The metapatterns in a family are partially ordered by the number of variables they contain. One can traverse a family of metapatterns from the general to the specific by incrementally instantiating the predicate variables with table names and built-in predicates and the object variables with column names and in turn constant values.

Notice that ***not*** all metapatterns in a family are interesting. In order to have some prediction value, a metapattern must be *connected*. That is, the predicate on the right-hand side must share at least one variable with some predicate on the left-hand side. For example, the metapattern $P(X,Y) \land Q(Y,Z) \Rightarrow R(U,V)$ is not interesting because its right-hand side is not connected to the left-hand side. Furthermore, predicates like $P(X,X)$ or $p(X,X)$ are not considered interesting because they do not link to others.

With the families of metapatterns so defined, one natural question to ask is whether the length of metapatterns can be arbitrarily long. Fortunately, for any given set of databases, the length of the longest metapatterns is bounded because metapatterns must be connected and there is only a fixed number of tables, columns, built-in predicates, and values that are presented in the databases. (Even if a column is typed "real", the number of distinct values in the column is finite because of the implementation.) Therefore, it is meaningful to define the *space* of all possible metapatterns for an application to be the union of all possible families of metapatterns.

Generating Metapatterns Based on Data Schema and Ranges

The space of metapatterns is very large and it is infeasible to enumerate them all. Our approach is to start with the set of most general metapatterns, and incrementally generate interesting ones as the process of discovery continues. Our goal is not to cover all the metapatterns but to guide the discovery process in fruitful directions.

Among all the general metapatterns, the transitivity metapattern (see MP-1 for example) is the most interesting. In essence, it subsumes many other types of metapatterns, such as implication, inheritance, transfers-through, and function dependency (Shen 1992). In this section, we describe how transitivity meatpatterns are generated. How other types of metapatterns are generated based on transitivities will be described in the section after the next.

The set of all possible transitivity metapatterns can be generated based on the data schema and ranges of the databases. The idea is to first identify the sets of columns that are significantly connected, and then use these sets to build metapatterns.

Two columns, from different database tables, are significantly connected, if they have the same type and have ranges that overlap each other above a user specified threshold o, where $0 < o < 1$. The degree of overlapping is computed as follows. Let C_x and C_y be two columns, and V_x and V_y be their value sets, respectively, then the overlapping of C_x and C_y is the maximum number of the shared values relative to either V_x or V_y, as follows:

$$Overlap(C_x, C_y) = \max\left(\frac{|V_x \cap V_y|}{|V_x|}, \frac{|V_x \cap V_y|}{|V_y|}\right)$$

where $|\cdot|$ denotes the cardinality of a set. Here domain knowledge may be used to eliminate unnecessary connections (e.g., height vs. temperature) or suggest and establish syntactically different connections (e.g., color vs. light frequency). Each pair of columns that are connected are then given a reference name, and these connections will be represented in a significant connection table (SCT), where each row is a connection, each column is a table, and each non-empty entry is the name of a connected data field (or column).

To illustrate the idea, consider for example an abstract database shown in Figure 2. In this database, there are four tables, t_1 to t_4, each has some columns c_{ij}. For simplicity, the value ranges of each column are also listed along with the schema. (In reality, value ranges can be obtained by simple SQL queries.)

Schema and Data Ranges

Table	Columns Type[ValueRange]		
t_1	c_{11}, char(2)	c_{12}, int[2–7]	c_{13}, real[0.4–0.8]
t_2	c_{21}, int[12–17]	c_{22}, real[0.1–0.7]	c_{23}, char(3)
t_3	c_{31}, int[13–16]	c_{32}, char(2)	
t_4	c_{41}, char(3)	c_{42}, real[0.0–0.1]	c_{43}, int[4–7]

Table t_1

c_{11}	c_{12}	c_{13}
jj	5	0.5
nn	5	0.8
ll	7	0.5
qq	5	0.5
kk	5	0.6
pp	4	0.6
mm	2	0.5
nn	4	0.6
kk	4	0.4
nn	5	0.4

Table t_2

c_{21}	c_{22}	c_{23}
14	0.5	mmm
14	0.6	iii
14	0.3	jjj
12	0.7	nnn
12	0.1	lll
15	0.6	ppp
15	0.4	mmm
13	0.6	ooo
16	0.6	ooo
17	0.4	mmm
14	0.4	lll
14	0.6	kkk
15	0.3	mmm
12	0.5	mmm
15	0.4	nnn
15	0.6	ooo
16	0.5	ppp
16	0.7	ppp

Table t_3

c_{31}	c_{32}
14	oo
15	kk
16	mm
15	kk
16	ll
15	ll
15	mm
13	oo
16	oo
14	mm
13	mm
14	mm

Table t_4

c_{41}	c_{42}	c_{43}
nnn	0.0	5
mmm	0.0	6
rrr	0.1	4
mmm	0.0	4
ooo	0.1	7
jjj	0.0	4
kkk	0.0	5
mmm	0.0	5
jjj	0.1	4
mmm	0.0	5
jjj	0.0	5
jjj	0.0	5
lll	0.0	7
nnn	0.1	4

Figure 2: An Example Database

Given these information, pairs of columns that are connected can be easily determined according to our definition. For example, suppose the threshold o for overlapping is set to 0.6, then column c_{13} in table t_1 and column c_{22} in table t_2 are connected because they have the same data type and their overlapping is 0.9. A reference name, X_1, is then created for this pair of connected columns. After considering every pair of columns, a significant connection table, shown in the upright part of Figure 3, is constructed. As we can see, every connected pair of columns is represented as a row in this SCT. For instance, columns c_{13} and c_{22} are in the first row, where c_{13} is under t_1 while c_{22} is under t_2.

For reasons that will become clear later, we also represent the information in SCT as a graph G, where each node in G is an non-empty entry in the SCT, and each edge connects two non-empty entries that are on the same row or column in the SCT. For example, the graph built from the SCT in Figure 3 is shown in the lower-left part of Figure 3, where node (t_1, X_1) and node (t_2, X_1) represent two non-empty entries, c_{13} and c_{22}, in the SCT. Since they are in the same row, there is an horizontal edge between them. Similarly, node (t_1, X_1) and node (t_1, X_2) represent two non-empty entries, c_{13} and c_{11}, in the same column of the SCT, so there is a vertical edge between them.

SCT

	t_1	t_2	t_3	t_4
X_1	c_{13}	c_{22}		
X_2	c_{11}		c_{32}	
X_3	c_{12}			c_{43}
X_4		c_{21}	c_{31}	
X_5		c_{23}		c_{41}

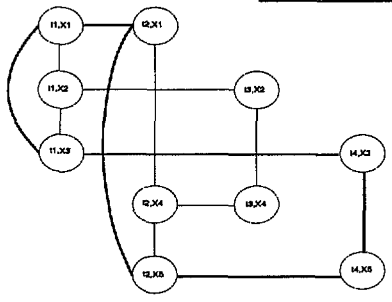

Figure 3: A Significant Connection Table (SCT) and its graph G

The graph G generated above provides a basis for generating all possible transitivity patterns in a given database. The idea is to find all the cycles in the graph with alternated vertical and horizontal edges, and convert each of these cycles into a "cycle" of predicates before generating a set of transitivity patterns.

Finding cycles in a graph can be accomplished by using a standard transitive closure algorithm with some simple augmentation to enforce the alternating edge constraint. A graph G cannot have more than $|G|!$ cycles because the length of a cycle, without duplicated nodes, cannot be greater than the number of the nodes in the graph. To convert a cycle of graph nodes into a cycle of predicates is also a straightforward task; one can simply rewrite each vertical edge in the cycle by the table name. For example, the cycle indicated by thick lines in Figure 3 is "$(t_2,X_1)(t_2,X_5)(t_4,X_5)(t_4,X_3)(t_1,X_3)(t_1,X_1)(t_2,X_1)$" and it can be rewritten as a cycle of predicates as "$t_2(X_1,X_5), t_4(X_5,X_3), t_1(X_3,X_1)$," where $t_2(X_1,X_5)$ is a rewrite of the vertical edge $(t_2,X_1)(t_2,X_5)$, and $t_4(X_5,X_3)$ is a rewrite of $(t_4,X_5)(t_4,X_3)$, and so on. Using this method, we can generate all cycles of predicates from the graph G, as listed in Figure 4.

$$t_2(X_1X_5)t_4(X_5X_3)t_1(X_3X_1)$$
$$t_2(X_1X_4)t_3(X_4X_2)t_1(X_2X_1)$$
$$t_2(X_5X_1)t_1(X_1X_2)t_3(X_2X_4)t_2(X_4X_5)$$
$$t_2(X_4X_5)t_4(X_5X_3)t_1(X_3X_1)t_2(X_1X_4)$$
$$t_1(X_3X_1)t_2(X_1X_4)t_3(X_4X_2)t_1(X_2X_3)$$
$$t_3(X_2X_4)t_2(X_4X_5)t_4(X_5X_3)t_1(X_3X_2)$$
$$t_1(X_2X_3)t_4(X_3X_5)t_2(X_5X_1)t_1(X_1X_2)$$
$$t_2(X_1X_4)t_3(X_4X_2)t_1(X_2X_3)t_4(X_3X_5)t_2(X_5X_1)$$
$$t_1(X_1X_2)t_3(X_2X_4)t_2(X_4X_5)t_4(X_5X_3)t_1(X_3X_1)$$

Figure 4: All predicate cycles found in the example DB

From the list of all possible cycles of predicates, we can now generate a complete set of transitivity metapatterns by generalizing table names and reference names and introducing an implication in each cycle. In our current example database, the result is the following set of metapatterns:

$$P_1(Y_1,Y_2) \wedge Q_1(Y_2,Y_3) \Rightarrow R_1(Y_1,Y_3) \quad \text{(MP-4)}$$
$$P_2(Y_1,Y_2) \wedge Q_2(Y_2,Y_3) \wedge W_2(Y_3,Y_4) \Rightarrow R_2(Y_1,Y_4) \quad \text{(MP-5)}$$
$$P_3(Y_1,Y_2) \wedge Q_3(Y_2,Y_3) \wedge W_3(Y_3,Y_4) \wedge V_3(Y_4,Y_5) \Rightarrow R_3(Y_1,Y_5) \quad \text{(MP-6)}$$

For example, MP-4 is a generalization of the first two predicate cycles in Figure 4; MP-5 is a generalization of cycles 3 through 7; and MP-6 is a generalization of the last two cycles. This set is complete because it includes all possible transitivity metapatterns in our example database.

Discovering and Evaluating New Patterns

Generating all transitivity metapatterns is not the end of our story. Depending on the strength or interestness of the patterns that are found with these metapatterns, the discovery system should generate more metapatterns that are deemed to be plausible. To do so, let us first examine how discovered patterns are evaluated.

When a metapattern is selected for execution, the system first instantiates it into a set of specific patterns that are possible in the current databases, and then evaluates them to check if they have sufficient support from the actual data. Given the information produced in the process of generating metapatterns, instantiating a metapattern is a straightforward procedure. It simply replaces the variables in the metapattern with specific table names and column names. For example, the predicate variables P_1, Q_1, and R_1 in MP-4 can be bound to table names t_1, t_2, t_3, and t_4 (these are the table names involve in the first two cycles of predicates in Figure 4), and the object variables Y_1, Y_2, and Y_3, can be bound to the reference variables X_1, X_2, X_3, X_4, and X_5, which in turn can be bound to corresponding columns according to the SCT in Figure 3, as follows:

$$t_2(X_1X_5)t_4(X_5X_3) \rightarrow t_1(X_3X_1)$$
$$t_1(X_3X_1)t_2(X_1X_5) \rightarrow t_4(X_5X_3)$$
$$t_4(X_5X_3)t_1(X_3X_1) \rightarrow t_2(X_1X_5)$$
$$t_2(X_1X_4)t_3(X_4X_2) \rightarrow t_1(X_2X_1)$$
$$t_1(X_2X_1)t_2(X_1X_4) \rightarrow t_3(X_4X_2)$$
$$t_3(X_4X_2)t_1(X_2X_1) \rightarrow t_2(X_1X_4)$$
$$\vdots$$

Notice that not all instantiated patterns are supported by the data in the database. We say a pattern is "interesting" only if its significance is above some user specified thresholds. In our approach, each pattern p is evaluated by two values that are computed against the databases: the *base* value p_b which reflects how much the left-hand side of p is supported by the actual data in the database, and the *strength* value p_s which reflects how much the right-hand side is supported by the data satisfying the left-hand side.

The base value is computed as $p_b = \frac{|LHS|}{|DOM(LHS)|}$, where LHS is the set of tuples in the database for which the left-hand side of p is true, and $DOM(LHS)$

is the *domain* of the tables in the left-hand side. A domain of two or more tables is defined as the union of the values of the fields (columns) through which these tables are joined. For example, if two tables S and T are joined by fields $S.a$ and $T.b$, and the field $S.a$ has values $\{v_1, v_2, v_3\}$, and the field $T.b$ has values $\{v_1, v_2, v_4\}$, then the domain of S and T, $DOM(S.a, T.b)$, in this join is the set of $\{v_1, v_2, v_3, v_4\}$, and the cardinality of this domain is $|DOM(S.a, T.b)| = 4$.

The strength value is computed as $p_s = \frac{|RHS|}{|LHS|}$, where LHS is defined the same as above, and RHS is the set of tuples in the LHS that satisfy the right-hand side of p.

A pattern's base and strength values, p_b and p_s, are compared with two user specified thresholds b and s. When both base and strength values are above their thresholds, the pattern is accepted. If the base value is the only one above its threshold, then the pattern is considered plausible. Such a pattern still has enough tuples (for it has a high enough base value) to be constrained to increase the strength value, and is recorded for further search (considered in the next section). A pattern is discarded when both base and strength are below the thresholds. As an example, suppose that the user specified thresholds are $b = 0.1$ and $s = 0.7$, and the following three sample patterns, with their base and strength values, can be found in the example database in Figure 2:

$$t_1(X_2X_1)t_3(X_4X_2) \rightarrow t_2(X_4X_1)\ [0.17, 0.7]$$
$$t_3(X_4X_2)t_1(X_2X_3)t_4(X_5X_3) \rightarrow t_2(X_4X_5)\ [0.02, 0.5]$$
$$t_2(X_4X_1)t_1(X_3X_1)t_4(X_5X_3) \rightarrow t_2(X_4X_5)\ [0.15, 0.4]$$

Among these patterns, the first one will be accepted because it has high enough base and strength values. The second one will be discarded because both its base and strength are lower than the thresholds. The third one will be kept as a plausible pattern because it has high enough base although its strength is low. Notice that for any given database, users may need several trial-and-errors to find suitable thresholds. (We are investigating methods to determine these parameters in a more principled way.)

Generating Metapatterns Based on Plausible Patterns

We have seen that with the initial set of metapatterns, a large set of actual patterns may be generated from the database. Some of these patterns are accepted, some discarded and some are still plausible. Interestingly, the plausible patterns provide the basis for dynamically generating more metapatterns. In particular, if a metapattern is associated with many plausible patterns, it will be used to generate more metapatterns by adding additional (meta)constraints to its left-hand side. If we consider these patterns as cycle of predicates, then with some added constraints the resultant patterns are cycles with extra or alternative branches. These types of patterns are beyond the simple transitivities.

Adding constraints to generate new metapatterns is accomplished as follows. Given a candidate metapattern, the system will add to its left-hand side a new (meta)constraint of the form $S(X, W)$, where S is a predicate variable and W is an object variable, while X must be a variable that already exists in the metapattern in order to link the constraint in. For example, one can add $S_1(Y_2, O)$, where O is an object variable, to MP-4 to get:

$$P_1(Y_1, Y_2) \wedge Q_1(Y_2, Y_3) \wedge S_1(Y_2, O) \Rightarrow R_1(Y_1, Y_3) \quad \text{(MP-7)}$$

or add $S_3(Y_2, Y_3)$ to MP-6 to get:

$$P_3(Y_1, Y_2) \wedge Q_3(Y_2, Y_3) \wedge W_3(Y_3, Y_4) \wedge V_3(Y_4, Y_5) \wedge S_3(Y_2, Y_3) \Rightarrow R_3(Y_1, Y_5) \quad \text{(MP-8)}$$

The motivation for this generation is to have the system search for an actual constraint, instantiated from S, that can yield patterns that have higher strengths.

The added meta-constraint can be instantiated to either a table that connects (i.e., share a reference name in SCT) to at least one predicate in the pattern or any of the build-in predicates (e.g., *equal*) with some variables that are already in the pattern. It is interesting to notice that the extended metapatterns enable the system to discover patterns that are beyond transitivities. For example, instantiating the metapattern MP-7, if P_1 and R_1 are bound to "ancestor", Q_1 to "parent", S_1 to "gender", and O to "male", then the following "male-ancestor" pattern may be discovered

$$ancestor(Y_1, Y_2), parent(Y_2, Y_3), gender(Y_2, male) \rightarrow ancestor(Y_1, Y_3)$$

In general, adding a new constraint to the left-hand side of a pattern will reduce the size of LHS, thus the number of plausible patterns will decrease as the length of their left-hand side increases. Furthermore, we only consider to add a constraint to a pattern when it reduces the number of tuples that satisfy the left-hand side of the pattern, so this process of adding constraints will eventually terminate.

Experimental Results

The described system has been applied to two simply databases to show that it can learn relational patterns directly from databases without requiring humans to pre-label the data as positive or negative examples of some pre-specified target concepts. The first database contains a small network used by the FOIL system (Quinlan 1990). From this database, our system learned, without pre-labeling the examples, the same recursive concepts as FOIL and more:

$$linkedTo(X_1X_3) \rightarrow canReach(X_1X_3)\ [1.0, 1.0]$$

$$canReach(X_1X_2) \wedge linkedTo(X_2X_3) \rightarrow canReach(X_1X_3)\ [0.14, 1.0]$$

$$canReach(X_1X_3) \rightarrow linkedTo(X_1X_3)\ [1.0, 0.5]$$

$$canReach(X_1X_3) \wedge linkedTo(X_2X_3) \rightarrow canReach(X_1X_2)\ [0.63, 0.4]$$

Although the last two patterns are not true in general, but they are interesting enough patterns in this database and have sufficient support from the data.

The second database contains a set of families used by Hinton's neural network (Hinton 1986). From this database, our system discovered many interesting relations among people, such as

$$husband(X_2, X_1) \rightarrow wife(X_1, X_2) \text{ [1.0, 1.0]}$$

$$niece(X_{65}, X_{78}), brother(X_{78}, X_{66}) \rightarrow nephew(X_{65}, X_{66}) \text{ [0.22, 1.0]}$$

$$daughter(X_{56}, X_{57}) brother(X_{57}, X_{60}) \rightarrow son(X_{56}, X_{60}) \text{ [0.33, 1.0]}$$

$$brother(X_{63}, X_{64}) \rightarrow sister(X_{64}, X_{63}) \text{ [1.0, 1.0]}$$

Again, the last one is not true in general, but a valid pattern in this particular database (i.e., all the families in the database have siblings of opposite gender).

Related Work

The most relevant work to this research includes supervised learning of relation patterns (e.g., (Quinlan 1990; Muggleton & Feng 1990)) and applications of ILP to KDD (e.g., (Dzeroski 1995)). However, most of these approaches learn rules that are rigid logical statements with no allowance for uncertainty, and often assume databases are correct and noise-free. While the patterns discovered by our method have associated probabilities that allows uncertainties and noise. One notable exception in ILP is the CLAUDIEN system (Raedt & Bruynooghe 1993; Laer, Dehaspe, & Raedt 1994), which uses the notion of *clausemodels* that share the same sprit of metapattern and can learn from positive examples only. However, CLAUDIEN requires clausemodels to be given while our system can generate metapatterns automatically. It would be interesting to see if the method described here can also be used to generate clausemodels for CLAUDIEN.

Conclusions and Future Work

In this paper, we have described a method for automatically generating metapatterns from meta-information of databases. With such a method, a metapattern-based, integrated data mining system can become both autonomous and interactive. There are clearly much room for more research in this area. One natural direction is to investigate methods for generating metapatterns that have other types of inductive actions on the right-hand side. The other one is to scale up this approach to large real-world databases. The current application under development is a set of six logistics databases, which have 104 tables, more than 2,000 columns, and over one million tuples. Following a valuable suggestion made by an anonymous reviewer, we will also include function dependency as another important criterion for testing significant connections.

Acknowledgments

We would like to thank Weixiong Zhang for his comments and suggestions on this paper. Special thanks to Eastman Chemical Company, Motorola Inc., and USC/Information Sciences Institution for providing tasks and resources. This work is also supported in part by the National Science Foundation under Grant No. IRI-9529615. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarilty reflect the views of the National Science Foundation.

References

Dzeroski, S. 1995. Inductive logic programming and knowledge discovery in databases. In *Advances in Knowledge Discovery and Data Mining.* MIT Press. chapter 5.

Fu, Y., and Han, J. 1995. Meta-rule-guided mining of association rules in relational databases. In *DOOD95 Workshop on the Integration of Knowlege Discovery with Deductive and Object Oriented Databases.*

Hinton, G. 1986. Learning distributed representations of concepts. In *Proceedings of the 8th Annual Conference of the Cognitive Science Society.*

Kero, B.; Russell, L.; Tsur, S.; and Shen, W. 1995. An overview of data mining technologies. In *DOOD95 Workshop on the Integration of Knowlege Discovery with Deductive and Object Oriented Databases.*

Laer, W. V.; Dehaspe, L.; and Raedt, L. D. 1994. Applications of a logical discovery engine. In *Proceedings of AAAI Workshop on Knowledge Discovery in Databases.* AAAI Press.

Muggleton, S., and Feng, C. 1990. Efficient induction of logic programs. In *Proceedings of the 1st Conference on Algorithmic Learning Theory.* Tokyo, Japan: Ohmsha.

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

Raedt, L. D., and Bruynooghe, M. 1993. A theory of clausal discovery. In *The Proceedings of the 13th IJCAI.*

Shen, W. M., and Leng, B. 1996. A metapattern-based automated discovery loop for integrated data mining. *IEEE Transactions on Data and Knowledge Engineering.* To appear in the special issue on data mining and knowledge discovery from databases.

Shen, W.; Ong, K.; Mitbander, B.; and Zaniolo, C. 1995. Metaqueries for data mining. In *Advances in Knowledge Discovery and Data Mining.* MIT Press. chapter 15.

Shen, W. 1992. Discovering regularities from knowledge bases. *International Journal of Intelligent Systems* 7(7):623–636.