CrossMine: Efficient Classification Across Multiple Database Relations[*]

Xiaoxin Yin
UIUC
xyin1@uiuc.edu

Jiawei Han
UIUC
hanj@cs.uiuc.edu

Jiong Yang
UIUC
jioyang@cs.uiuc.edu

Philip S. Yu
IBM T. J. Watson Resch. Ctr.
psyu@us.ibm.com

Abstract

Most of today's structured data is stored in relational databases. Such a database consists of multiple relations which are linked together conceptually via entity-relationship links in the design of relational database schemas. Multi-relational classification can be widely used in many disciplines, such as financial decision making, medical research, and geographical applications. However, most classification approaches only work on single "flat" data relations. It is usually difficult to convert multiple relations into a single flat relation without either introducing huge, undesirable "universal relation" or losing essential information. Previous works using Inductive Logic Programming approaches (recently also known as Relational Mining*) have proven effective with high accuracy in multi-relational classification. Unfortunately, they suffer from poor scalability w.r.t. the number of relations and the number of attributes in databases.*

In this paper we propose CrossMine, an efficient and scalable approach for multi-relational classification. Several novel methods are developed in CrossMine, including (1) tuple ID propagation*, which performs semantics-preserving virtual join to achieve high efficiency on databases with complex schemas, and (2) a selective sampling method, which makes it highly scalable w.r.t. the number of tuples in the databases. Both theoretical backgrounds and implementation techniques of CrossMine are introduced. Our comprehensive experiments on both real and synthetic databases demonstrate the high scalability and accuracy of CrossMine.*

* The work was supported in part by National Science Foundation under Grant No. 02-09199, the Univ. of Illinois, and an IBM Faculty Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

1. Introduction

Most of today's structured data is stored in relational databases. In many real world applications, such as evaluation of credit card applications and fraud detection, information stored in multiple relations needs to be used in decision making. Many important classification approaches, such as neural networks [9] and support vector machines [2], can only be applied to data represented in single, "flat" relational form. Multiple relations in a database are usually connected via *semantic links* such as *entity-relationship links* of an ER model used in the database design [5]. Data stored in the same relation often have closer semantic relationship than those reachable via remote links. It is counterproductive to simply "convert" multiple relational data into a single flat data relation because such conversion may lead to the generation of a huge universal relation [5] but lose some essential semantic information carried by the semantic links in the database design.

To tackle the multi-relational classification problem in relational databases, there are two major challenges: one is efficiency and scalability, and the other is the accuracy of classification. When building a classifier for a database with many relations, the search space is usually very large, and it is unaffordable to perform exhaustive search. On the other hand, the semantic links usually become very weak after passing through a long chain of links. Therefore, a multi-relational classifier needs to handle both efficiency and accuracy problems. Until now, there is still lack of accurate, efficient, and scalable multi-relational classification methods to handle large databases with complex schemas.

The most widely used category of approaches to multi-relational classification is Inductive Logic Programming (ILP) [10, 8]. The ILP approaches achieve good classification accuracy in multi-relational classification. However, most ILP approaches are not scalable w.r.t. the number of relations and the number of attributes in the database. Thus they are usually inefficient for databases with complex schemas. The ILP classification approaches aim at finding hypotheses of certain format that can predict the class labels of examples, based on the background knowledge. The hypotheses are often sets of rules, each consisting of a set of predicates. Take FOIL (*first-order inductive learner*) [15] as an example. FOIL constructs a set of conjunctive rules that distinguish positive examples from negative ones. To build a rule, it repeatedly searches for the best predicate and appends it to the rule. It evaluates all possible predicates to find the best one. To evaluate a predicate, it needs to append it to the current rule and find out the number of positive and negative tuples satisfying this rule. This process is very time-consuming when the number of relations is large or the number of possible predicates is large.

In a database for multi-relational classification, there is one target relation R_t, whose tuples are called *target tuples*. Each target tuple is associated with a class label. Other relations are non-target relations and may contain relevant information that helps classification. To build a good multi-relational classifier, one needs to find good predicates in each non-target relation R that help distinguish positive and negative target tuples. Many ILP approaches are not efficient because they evaluate many predicates separately, and it takes one or more join operations (or equivalents) to evaluate each predicate. One method to reduce the computational cost is to first join the target relation with other relations and then evaluate the predicates on one result relation. Then many predicates can be evaluated simultaneously by scanning the result relation only once. However, to repeatedly find good predicates to build rules, one needs to join many relations in different ways, which is expensive in both time and space.

In this paper we propose CrossMine, a scalable and accurate approach for multi-relational classification. Its basic idea is to propagate the tuple IDs (together with their associated class labels) in the target relation to other relations. In the relation to which the IDs are propagated, each tuple t is associated with a set of IDs, which represent the target tuples that are joinable with t. Tuple ID propagation is a flexible method. Besides propagating IDs from target relation to non-target relations, one can propagate the IDs transitively to additional non-target relations to search for good predicates among many relations. The idea of tuple ID propagation is to virtually join the relations with minimal cost, and then find good predicates in the joined relation. CrossMine obtains high scalability by avoiding the high cost of physical joins.

CrossMine uses rules for classification. It uses a sequential covering algorithm, which repeatedly constructs rules and removes positive examples covered by each rule. To construct a rule, it repeatedly searches for the best predicate and appends it to the current rule. During the search process, CrossMine limits the search space to relations related to the target relation or related to relations used in the rule. In this way the strong semantic links can be identified and the search process is controlled in promising directions. On the other hand, the search space of CrossMine is much larger than typical ILP approaches. By tuple ID propagation, CrossMine actually considers two predicates at the same time (one for join and another for value constraint). Moreover, CrossMine performs *look-one-ahead*, and considers up to four predicates at a time. CrossMine achieves both high efficiency and high accuracy by controlling the search space and identifying strong semantic links.

In many sequential covering algorithms, the negative examples are never removed in the rule building process, which makes the algorithm inefficient for databases with large numbers of tuples. It is common that before building a rule, there are much less positive examples than negative ones, which causes the classifier to spend a large amount of time to build low-quality rules. To address this issue, CrossMine employs a selective sampling method to reduce the number of negative tuples when the numbers of positive and negative tuples are unbalanced. This helps CrossMine to achieve high scalability w.r.t. the number of tuples in databases. Our experiments show that the sampling method decreases the running time significantly and only slightly lowers the accuracy.

The remaining of the paper is organized as follows. In Section 2 we introduce the related work. Section 3 introduces the idea of tuple ID propagation and its theoretical background. We describe the algorithm and implementation issues in Section 4. Experimental results are presented in Section 5 and the study is concluded in Section 6.

2. Related Work

The most important category of approaches in multi-relational classification is ILP. The formal definition of the ILP problem is as follows. Given background knowledge B, a set of positive examples P, and a set of negative examples N, find a hypothesis H, which is a set of Horn clauses such that:

- $\forall p \in P : H \cup B \models p$ (completeness)
- $\forall n \in N : H \cup B \not\models n$ (consistency)

The well known ILP systems include FOIL [15], Golem [12], and Progol [11]. FOIL is a top-down learner, which builds rules that cover many positive examples and few negative ones. Golem is a bottom-up learner, which performs generalizations from the most specific rules. Progol uses a combined search strategy. TILDE [1], a more recent approach, uses the idea of C4.5 [14] and inductively constructs decision trees. TILDE is more efficient than most traditional ILP approaches due to the divide-and-conquer nature of decision tree algorithm.

Besides ILP, probabilistic approaches [7, 16, 13] are also popular for multi-relational classification and modeling. The most important one is the probabilistic relational models (PRM's) [7, 16] which is an extension of Bayesian networks for handling relational data. PRM's can integrate the advantages of both logical and probabilistic approaches for knowledge representation and reasoning. In [13] an approach is proposed to integrate ILP and statistical modeling for document classification and retrieval.

Another way for modeling or classifying relational data is through frequent pattern or association rule mining. In [17] an approach is proposed for frequent pattern mining in graphs, which can be applied on multi-relational data. In [4] the authors propose an approach for association rule mining in relational databases.

We take FOIL as a typical example of ILP approaches and show its working procedure. FOIL is a sequential covering algorithm that builds rules one by one. After building each rule, all positive target tuples satisfying that rule are removed. Predicates are added one by one when building rules. At each step, every possible predicate is evaluated and the best one is appended to the current rule. To evaluate a predicate p, p needs to be appended to the current rule r to get a new rule r'. Then it constructs a new dataset which contains all target tuples satisfying r', together with the relevant non-target tuples on the join path specified by r'. And p is evaluated based on the number of positive and negative target tuples satisfying r'. For databases with complex schemas, the search space is huge and there are many possible predicates at each step. To build rules, FOIL needs to repeatedly construct datasets by physical joins to find good predicates, which is very time-consuming. The inefficiency of this method is also verified by our experiments.

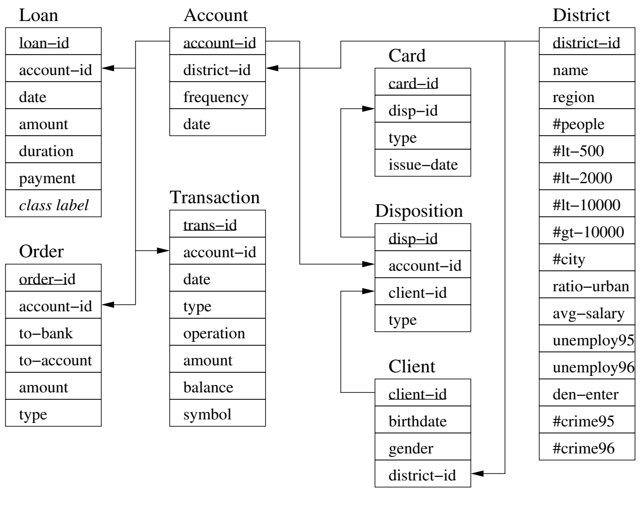


Figure 1. The financial database from PKDD CUP 99.

3. Tuple ID Propagation

In this section we explain the idea of tuple ID propagation and method of finding good predicates with such a propagation. In essence, tuple ID propagation is a method for virtually joining non-target relations with the target one. It is a flexible and effective method, and is much less costly than physical join in both time and space.

3.1. Basic Definitions

A database consists of a set of relations, one of which is the target relation, with class labels associated with its tuples. The other relations are non-target relations. Each relation may have one primary key and several foreign keys, each pointing to the primary key of some other relation. We consider the following types of joins:

1. Join between a primary key k and some foreign key pointing to k.
2. Join between two foreign keys k_1 and k_2, which point to the same primary key k. (For example, the join between Loan.account-id and Order.account-id.)

We ignore other possible joins because they do not represent strong semantic relationships between entities in the database. Figure 1 shows an example database. Arrows go from primary-keys to corresponding foreign-keys. The target relation is $Loan$. Each target tuple is either positive or negative, indicating whether the loan is paid on time. The class labels are usually either assigned by experts or derived via some sophisticated computation rules, such as based on the payment history of a customer.

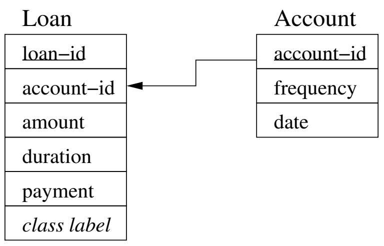
We first define the goodness of predicates. A predicate is a constraint on certain attribute in certain relation. For example, predicate "$Loan(L, ?, ?, ?, 12, ?)$" means that the duration of loan L is 12 months, and predicate "$Loan(L, A, ?, ?, ?, ?)$" means that loan L has value A on account-id attribute.

Let us take the sample database in Figure 2 as an example. Suppose the current rule is empty, and our goal is to find the best predicate. We use *foil gain* [15] to measure the goodness of a predicate.

Definition 1 (Foil gain). *For a rule r, we use $P(r)$ and $N(r)$ to denote the number of positive and negative examples satisfying r. Suppose the current rule is r. We use $r + p$ to denote the rule constructed by appending predicate p to r. The foil gain of predicate p is defined as follows,*

$$I(r) = -\log \frac{P(r)}{P(r) + N(r)} \tag{1}$$

$$foil\_gain(p) = P(r + p) \cdot [I(r) - I(r + p)] \tag{2}$$



Account		
account-id	frequency	date
124	monthly	960227
108	weekly	950923
45	monthly	941209
67	weekly	950101

Loan					
loan-id	account-id	amount	duration	payment	*class*
1	124	1000	12	120	+
2	124	4000	12	350	+
3	108	10000	24	500	−
4	45	12000	36	400	−
5	45	2000	24	90	+

Figure 2. A sample database (The last column of Loan relation contains class labels.)

Intuitively $foil\_gain(p)$ represents the total number of bits saved in representing positive examples by appending p to the current rule.

With foil gain one can measure the goodness of each predicate w.r.t. the target relation $Loan$. For a predicate "$Account(A, ?, monthly, ?)$", suppose rule $r = Loan(L, +) :- Loan(L, A, ?, ?, ?, ?)$, $Account(A, ?, monthly, ?)$. We say a tuple t in $Loan$ satisfies r if and only if **any** tuple in $Account$ that is joinable with t has value "monthly" in the attribute of *frequency*. In this example, there are two tuples (with account-id 124 and 45) in $Account$ that satisfy the predicate "$Account(A, ?, monthly, ?)$". So there are four tuples (with loan-id 1, 2, 4, and 5) in $Loan$ that satisfy this rule.

3.2. Search for Predicates by Joins

Consider the sample database in Figure 2. To compute the foil gain of predicates in a non-target relation, such as $Account$, we need to first set the current rule r to "$Loan(L, +) :- Loan(L, A, ?, ?, ?, ?)$", and then for each predicate p in $Account$ relation, find out all positive and negative tuples satisfying $r + p$. For example, if p = "$Account(A, ?, monthly, ?)$", then rule $r + p$ is "$Loan(L, +) :- Loan(L, A, ?, ?, ?, ?)$, $Account(A, ?, monthly, ?)$", which can be converted into an SQL query:

SELECT L.loan-id
FROM Loan L, Account A
WHERE L.account-id = A.account-id AND A.frequency = 'monthly'

$P(r + p)$, $N(r + p)$, and hence the foil gain of p can be computed from the result of this query. If the database is stored in main memory as a set of arrays, one can simulate the process of executing an SQL query to find out the target tuples satisfying $r + p$.

Many ILP approaches for multi-relational classification use similar methods to evaluate predicates. The disadvantage of this method is that it takes nontrivial computation to evaluate each predicate, and it needs to evaluate many predicates to find the best one. This is costly for databases with complex schemas.

It should be noted that much computation is performed repeatedly in the above process. The join of the target relation with each non-target relation is performed many times in the evaluation of such predicates. A more efficient approach is to do the join once and compute the foil gain of all predicates. For the sample database in Figure 2, the two relations can be joined, as shown in Figure 3. With the joined relation, the foil gain of every predicate in both relations can be computed. To compute the foil gain of all predicates on a certain attribute, one only needs to scan the corresponding column in the joined relation once. It can also handle continuous attribute as in [14]. To find the best predicate on attribute $Account.date$, one can first sort that column, then iterate from the smallest value to the largest value, and for each value d, compute the foil gain of two predicates "date $\leq d$" and "date $\geq d$".

Loan ⋈ Account							
l-id	a-id	amount	dur	pay	freq	date	*class*
1	124	1000	12	120	monthly	960227	+
2	124	4000	12	350	monthly	960227	+
3	108	10000	24	500	weekly	950923	−
4	45	12000	36	400	monthly	941209	−
5	45	2000	24	90	monthly	941209	+

Figure 3. The join of $Loan$ and $Account$.

Although join can be used for evaluation, it is quite expensive to repeatedly search for predicates by performing joins. Since one needs to evaluate numerous predicates on many different relations, one needs to join the relations in many different ways. When the rule gets long, the joined relation can be very wide and contains a huge number of tuples, among which many are redundant (as shown in relational schema normalization studies) [5]. It is time consuming to perform computation on such a relation. In the following sections, another approach called *tuple ID propagation* is proposed to overcome this problem.

3.3. Tuple ID Propagation

Loan					
loan-id	account-id	amount	duration	payment	*class*
1	124	1000	12	120	$+$
2	124	4000	12	350	$+$
3	108	10000	24	500	$-$
4	45	12000	36	400	$-$
5	45	2000	24	90	$+$

Account				
account-id	frequency	date	IDs	*class labels*
124	monthly	960227	1, 2	2+, 0−
108	weekly	950923	3	0+, 1−
45	monthly	941209	4, 5	1+, 1−
67	weekly	950101	–	0+, 0−

Figure 4. Example of tuple ID propagation.

Suppose the primary key of the target relation is an attribute of integers, which represents the ID of each target tuple. Consider the sample database shown in Figure 4, which has the same schema as in Figure 2. Instead of performing physical join, the IDs and class labels of target tuples can be propagated to the $Account$ relation. The procedure is formally defined as follows.

Definition 2 (Tuple ID propagation). *Suppose two relations R_1 and R_2 can be joined by attributes $R_1.A$ and $R_2.A$. Each tuple t in R_1 is associated with a set of IDs in the target relation, represented by $idset(t)$. For each tuple u in R_2, we set $idset(u) = \bigcup_{t \in R_1, t.A=u.A} idset(t)$.*

The following theorem and its corollary show the correctness of tuple ID propagation and how to compute foil gain from the propagated IDs.

Theorem 1 *Suppose two relations R_1 and R_2 can be joined by attribute $R_1.A$ and $R_2.A$, and R_1 is the target relation, with primary key $R_1.id$. All the tuples in R_1 satisfy the current rule (all the others have been eliminated). The current rule contains a predicate "R_1($R_1.id$, $R_1.A, \cdots$)", which enables the join of R_1 with R_2. With tuple ID propagation from R_1 to R_2, for each tuple u in R_2, $idset(u)$ represents all target tuples joinable with u, using the join path specified in the current rule.*

Proof. *From definition 2, we have $idset(u) = \bigcup_{t \in R_1, t.A=u.A} idset(t)$. That is, $idset(u)$ represents the target tuples joinable with u using the join path specified in the current rule.*

Corollary 1 *Suppose two relations R_1 and R_2 can be joined by attribute $R_1.A$ and $R_2.A$, R_1 is the target relation, and all the tuples in R_1 satisfy the current rule (all others have been eliminated). If R_1's IDs are propagated to R_2, then the foil gain of every predicate in R_2 can be computed using the propagated IDs on R_2.*

Proof. *Given the current rule r, for a predicate p in R_2, such as $R_2.B = b$, its foil gain can be computed based on $P(r)$, $N(r)$, $P(r+p)$ and $N(r+p)$. $P(r)$ and $N(r)$ should have been computed during the process of building the current rule. $P(r+p)$ and $N(r+p)$ can be computed in the following way: (1) find all tuples t in R_2 that $t.B = b$; (2) with the propagated IDs on R_2, find all target tuples that can be joined with any tuple found in (1) (using the join path specified in the current rule); and (3) count the number of positive and negative tuples found in (2).*

For example, suppose the current rule is "$Loan(L, +)$:− $Loan(L, A, ?, ?, ?, ?)$". For predicate "$Account(A, ?, monthly, ?)$", we can first find out tuples in $Account$ relation that satisfy this predicate, which are {124, 45}. Then we can find out tuples in $Loan$ relation that can be joined with these two tuples, which are {1, 2, 4, 5}. We maintain a global table of the class label of each target tuple. From this table, we know that tuples {1, 2, 4, 5} contain three positive and one negative examples. With this information we can easily compute the foil gain of predicate "$Account(A, ?, monthly, ?)$".

Please notice that one cannot compute the foil gain only from the class labels of each tuple in the $Account$ relation (see Figure 4). Suppose a tuple t in the $Loan$ relation can be joined with two tuples in the $Account$ relation. Using only the number of positive and negative tuples associated with every tuple in the $Account$ relation to compute the foil gain, the tuple t will be counted twice, and one cannot get the exact number of positive and negative tuples satisfying a rule. Only with the propagated IDs can one find out the exact set of target tuples that satisfy a rule.

Besides propagating IDs from the target relation to relations directly joinable with it, one can also propagate IDs transitively by propagating the IDs from one non-target relation to another, according to the following theorem.

Theorem 2 *Suppose two non-target relations R_2 and R_3 can be joined by attribute $R_2.A$ and $R_3.A$, and all the tuples in R_2 satisfy the current rule (all others have been eliminated). For each tuple v in R_2, $idset(v)$ represents the target tuples joinable with v (using the join path specified by the current rule). By propagating IDs from R_2 to R_3 through the join $R_2.A = R_3.A$, for each tuple u in R_3, $idset(u)$ represents target tuples that can be joined* *with u (using the join path in the current rule, plus the join $R_2.A = R_3.A$).*

Proof. *Suppose a tuple u in R_3 can be joined with v_1, v_2, $\cdots$, v_m in R_2, using join $R_2.A = R_3.A$. Then $idset(u) = \bigcup_{i=1}^{m} idset(v_i)$. A target tuple t is joinable with any one of v_1, v_2, $\cdots$, v_m if and only if $t.id \in \bigcup_{i=1}^{m} idset(v_i)$. Therefore, a target tuple t is joinable with u (using the join path in the current rule, plus the join $R_2.A = R_3.A$) if and only if $t.id \in idset(u)$.*

A corollary similar to corallary 1 can be proved for theorem 2. That is, by tuple ID propagation between non-target relations, one can also compute the foil gain based on the propagated IDs.

Tuple ID propagation is a way to perform virtual join. Instead of physically joining the relations, they are virtually joined by attaching the tuple IDs of the target relation to the tuples of a non-target relation, using a certain join path. In this way, the semantic links between the target relation and non-target relations can be found, and the foil gain of predicates can be computed as if physical join is performed.

Tuple ID propagation is a flexible and efficient method. IDs (and their associated class labels) can be easily propagated from one relation to another. By dong so, predicates in different relations can be evaluated with little redundant computation. The required space is also small because the IDs do not take much additional storage space.

ID propagation, though valuable, should be enforced with semantic constraints in mind. There are two cases that such propagation could be counter-productive: (1) propagate via large fan-outs, and (2) propagate via long weak links. The first case happens if the there are too many tuples that can be produced via propagation. Suppose after the IDs are propagated to a relation R, it is found that every tuple in R can be joined to many target tuples and every target tuple can be joined to many tuples in R. Then the semantic link between R and the target relation is usually very weak because the link is very unselective. For example, propagation among people via birth-country links may not be productive. The second case happens if the propagation goes through long weak links, e.g., linking a student with his car dealer's pet (via car, and then dealer) may not be productive either. From the consideration of both efficiency and accuracy, our system discourages propagation via links that (1) have very large fan-out, (2) not on active relations (this will be explained later).

4. The CrossMine Algorithm

In this section we present the algorithm CrossMine for building rules by tuple ID propagation. A sequential covering algorithm is developed that repeatedly builds rules and removes positive tuples satisfying the rule. To build a rule, it repeatedly searches for the best predicate and adds it to the current rule. This algorithm is selected because it guarantees the quality of each rule by always keeping a large number of negative examples, and moreover, its greedy nature makes it efficient in large databases. CrossMine also uses a sampling method to improve scalability w.r.t. the number of tuples in the database.

Algorithm 1 Find-Rules

Input: a relational database D with a target relation R_t.

Output: a set of rules for predicting class labels of target tuples.

```
Procedure
    rule set $R \leftarrow emptyset$;
    do
        rule $r \leftarrow$ Find-A-Rule();
        add $r$ to $R$;
        remove all positive target tuples satisfying $r$;
    while(there are more than 10% positive target tuples left);
    return $R$;
```

Figure 5. Algorithm *Find-Rules*

4.1. Rule Representation

The algorithm aims at finding rules that can distinguish positive target tuples from negative ones. Each rule is a list of predicates, associated with a class label. The form of rules used here is different from that by the traditional ILP approaches. Instead of using conventional predicates, *complex predicate* are use here as elements of rules. A complex predicate $\hat{p}$ contains two parts:

1. *prop-path*: indicates how to propagate IDs. For example, "$Loan.account\_id \rightarrow Account.account\_id$" indicates propagating IDs from the $Loan$ relation to the $Account$ relation using the join "$Loan.account\_id = Account.account\_id$".
2. *constraint*: indicates the constraint on the relation where the IDs are propagated to. For example, "$Account.frequency = monthly$" indicates that tuples in the $Account$ relation should have value "monthly" on attribute *frequency*.

The prop-path of a complex predicate may be empty if we already have the right tuple IDs on the relation to which the constraint is applied.

A complex predicate is usually equivalent to two conventional predicates. For example, the rule "$Loan(L, +)$:– $Loan(L, A, ?, ?, ?, ?)$, $Account(A, ?, monthly, ?)$" can be represented by "$Loan(+)$:– $[Loan.account\_id \rightarrow Account.account\_id$, $Account.frequency = monthly]$".

Algorithm 2 Find-A-Rule

Input: a relational database D with a target relation R_t.

Output: a rule for predicting class labels of target tuples.

Procedure

```
rule $r \leftarrow$ empty-rule;
set $R_t$ to active;
do
    Complex predicate $p \leftarrow$ Find-Best-Pred();
    if $foil\_gain(p) <$ MIN_FOIL_GAIN;
    then break;
    else
        $r \leftarrow r + p$;
        remove all target tuples not satisfying $r$;
        update IDs on every active relation;
        if $p.constraint$ is on an inactive relation
        then set that relation active;
while($r.length <$ MAX_RULE_LENGTH);
return $r$;
```

Figure 6. Algorithm ***Find-A-Rule***

4.2. The CrossMine Algorithm

Given a relational database with one target relation, CrossMine builds a classifier containing a set of rules, each of which contains a list of complex predicates and a class label. The overall idea is to repeatedly build rules. After each rule is built, remove all positive target tuples satisfying it. The algorithm is shown in Figure 5.

To build a rule, one repeatedly searches for the best complex predicate and appends it to the current rule, until the stop criterion is met. A relation is *active* if it appears in the current rule, or it is the target relation. Every active relation is required to have the correct propagated IDs on every tuple before searching for the next best predicate. The algorithm is shown in Figure 6.

The following procedure is used to find the best predicate: (1) for every active relation $\hat{R}$, find the best complex predicate whose constraint applies on $\hat{R}$ (no ID propagation involved), and (2) for every relation $\bar{R}$ that can be joined with some active relation $\hat{R}$, propagate IDs from $\hat{R}$ to $\bar{R}$, and find the best complex predicate on $\bar{R}$. Consider the database in Figure 1. Originally only $Loan$ is active. Suppose the first best complex predicate is "[$Loan.account\_id \rightarrow Account.account\_id$, $Account.frequency = monthly$]". Now $Account$ becomes active as well. And we will try to propagate the tuple IDs from $Loan$ or $Account$ in every possible way to find the next best predicate.

The idea behind the algorithm of building a rule is as follows. Starting from the target relation R_t, find the best complex predicate $\hat{p}$, which propagates IDs from R_t to another relation $\bar{R}$. Then start from either R_t or $\bar{R}$ to find the next complex predicate. This algorithm is greedy in nature. It extends the rule using only those predicates in either the active relations or the relations directly joinable with an active relation.

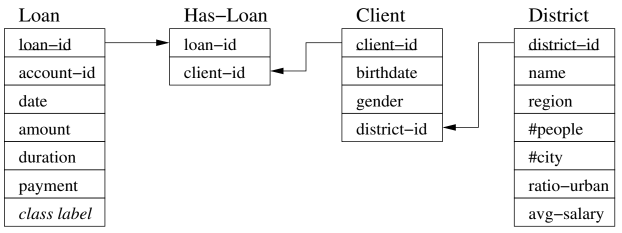


Figure 7. Another sample database.

The above algorithm may fail to find good predicates in databases containing some relations that are used to join with other relations, such as the database shown in Figure 7. In this database there is no meaningful predicate in the Has_Loan relation. Therefore, the rules built will never involve any predicates on the $Client$ relation and the $District$ relation.

This problem can be solved using the *look-one-ahead* method. When searching for the best predicate, after IDs have been propagated to a relation $\bar{R}$, if $\bar{R}$ contains a foreign-key pointing to relation $\bar{R}'$, IDs are propagated from $\bar{R}$ to $\bar{R}'$, and used to search for good predicates in $\bar{R}'$. By this method, in the example in Figure 7, one can find rules such as "$Loan(+)$:− [$Loan.loan\_id \rightarrow Has\_Loan.loan\_id$, $Has\_Loan.client\_id \rightarrow Client.client\_id$, $Client.birthdate <$ 01/01/60]".

With the correct IDs on a relation $\bar{R}$, one can scan $\bar{R}$ once to compute the number of positive and negative target tuples satisfying every predicate in $\bar{R}$, using the approach in [6]. For continuous attributes, one can use the method in Section 3.2 to search for good predicates. The algorithm for searching for the best complex predicate is shown in Figure 8.

The above algorithms show the procedure of building rules in CrossMine. The basic idea of building a rule is to start from the target relation, keep appending predicates in active relations or relations related to some active relation, until the stopping criterion is met. The running time of CrossMine is not much affected by the number of relations in the database, because the size of the search space is mainly determined by the number of active relations and the number of joins on each active relation. This is also verified in our experiments on synthetic databases.

To achieve high accuracy in multi-relational classification, an algorithm should be able to find most of the useful predicates in the database, and builds good rules with

Algorithm 3 Find-Best-Pred

Input: a relational database D with a target relation R_t, and current rule r.

Output: the complex predicate with most foil gain.

```
Procedure
  Complex predicate $p_{max} \leftarrow empty$;
  for each active relation $\hat{R}$
    Complex predicate $p \leftarrow$ best complex predicate in $\hat{R}$;
    if $foil\_gain(p) > foil\_gain(p_{max})$
    then $p_{max} \leftarrow p$;
  for each relation $\bar{R}$
    for each key/foreign-key $k$ of $\bar{R}$
      if $\bar{R}$ can be joined to some active relation $\hat{R}$ with $\bar{R}.k$
      then
            propagate IDs from $\hat{R}$ to $\bar{R}$;
            $p \leftarrow$ best complex predicate in $\bar{R}$;
            if $foil\_gain(p) > foil\_gain(p_{max})$
            then $p_{max} \leftarrow p$;
            for each foreign-key $k' \neq k$ of $\bar{R}$
                  propagate IDs from $\bar{R}$ to relation $\bar{R}'$
                    that is pointed to by $\bar{R}.k$;
                  $p \leftarrow$ best complex predicate in $\bar{R}'$;
                  if $foil\_gain(p) > foil\_gain(p_{max})$
                  then $p_{max} \leftarrow p$;
  return $p_{max}$;
```

Figure 8. Algorithm *Find-Best-Pred*

them. In most commercial databases following E-R model design there are two types of relations: *entity* relation and *relationship* relation. Usually each entity relation is reachable from some other entity relations via join paths going through relationship relations. Suppose an entity relation R contains useful information for classification. There are usually many join paths between R and the target relation R_t, some representing important semantic links. It is likely that R can be reached from some other useful entity relations through relationship relations. Therefore, by using the method of look-one-ahead, it is highly probable that one can utilize the information in R.

Most ILP approaches also perform heuristical search when building rules. However, the search spaces of those approaches are usually much smaller than that of CrossMine. By using complex predicates, CrossMine considers two predicates at a time (one for join and another for value constraint). By using look-one-ahead, it can consider up to four predicates together in rule generation. This enables CrossMine to find good predicates and build more accurate classifiers than traditional ILP approaches.

On the other hand, CrossMine is rather different from joining a large number of relations indiscriminately, such as the "universal relation" approach. Instead, it limits the search process (i.e., tuple ID propagation) among only active relations with at most one look-ahead. Thus the search space is more confined, following more promising and active links than indiscriminate joins, and thus lead to both high efficiency and classification accuracy.

4.3. Tuple Sampling

From Algorithm 1 we can see that during the procedure of building rules, the number of positive tuples keeps decreasing and the number of negative tuples remains unchanged. Usually each rule covers a certain proportion of the remaining positive tuples (usually 5% to 20%). Therefore, the first several rules can often cover the majority of the positive tuples. However, even if most of positive tuples have been covered, it still takes a similar amount of time to build a rule because all and the large number of negative tuples remain.

Let $r.sup^+$ and $r.sup^-$ be the number of positive and negative tuples satisfying a rule r. Let $r.bg^+$ and $r.bg^-$ be the number of positive and negative tuples from which r is built. The accuracy of r can be estimated using the method in [3], which is shown in the following equation:

$$Accuracy(r) = (r.sup^+ + 1)/(r.sup^+ + r.sup^- + c) \quad (3)$$

where c is the number of classes.

In the algorithm described above, $r.bg-$ always equals to the number of negative tuples. When $r.bg+$ is small, even if $r.bg-$ is large, the quality of r cannot be guaranteed. That is, if $r.bg+$ is small, one cannot be confident that $Accuracy(r)$ is a good estimate for the real world accuracy of r. Therefore, although much time is spent in building these rules, the quality of the rules is usually much lower than that of the rules with high bg^+ and bg^-.

Based on this observation, the following method is proposed to improving its effectiveness. Before a rule is built, we require that the number of negative tuples is no greater than NEG_POS_RATIO times the number of positive tuples. Sampling is performed on the negative tuples if this requirement is not satisfied. We also require that the number of negative tuples is smaller than MAX_NUM_NEGATIVE, which is a large constant.

Here we analyze the improvement on efficiency by sampling. Our experiments show that, when only a small portion of positive tuples remain, each rule generated usually covers an even smaller portion of remaining positive tuples. The possible reason is that, there are usually many "noisy" positive tuples that cannot be covered by any good rule. The consequence is that the number of rules generated usually increases with the number of target tuples. Suppose there are m tuples in the database, and about half of them are positive. We assume the number of rules is $f(m)$, which is a sublinear function of m. Suppose it takes $\Theta(m)$ time to find a rule. Then the algorithm without sampling takes $\Theta(m \cdot f(m))$ time. If sampling is used, the time for building a rule is proportional to the number of remaining positive tuples. Because the first several rules can often cover the majority of positive tuples, the total number of tuples decreases sharply after finding several rules. Thus the total running time is close to $\Theta(m)$.

When sampling is used, the accuracy of rules should be estimated in a different way. Suppose before building rule r, there are P positive and N negative tuples. N' negative tuples are randomly chosen by sampling ($N' < N$). After building rule r, suppose there are p positive and n' negative tuples satisfying r. We need to estimate n, the number of negative tuples satisfying r. The simplest estimation is $n \approx n' \frac{N}{N'}$. However, this is not a safe estimation because it is quite possible that r luckily excludes most of the N' negative examples but not the others. We want to find out a number n, so that the probability that $n' \leq n \frac{N'}{N}$ is 0.9. Or to say, it is unlikely that $\frac{n}{N} \leq \frac{n'}{N'}$.

As we know, N' out of N negative tuples are chosen by sampling. Assume we already know that n negative tuples satisfy r. Consider the event of a negative tuple satisfying r as a random event. Then n' is a random variable obeying binomial distribution, $n' \sim B(N', \frac{n}{N})$. n' can be considered as the sum of N' random variable of $B(1, \frac{n}{N})$. When N' is large, according to central limit theorem, we have $\frac{n'}{N'} \sim N(\frac{n}{N}, \frac{\frac{n}{N}(1-\frac{n}{N})}{N'})$. For a random variable $X \sim N(\mu, \sigma^2)$, $P(X \geq \mu - 1.28\sigma) \approx 0.9$. So we require

$$\frac{n'}{N'} = \frac{n}{N} - 1.28\sqrt{\frac{\frac{n}{N}(1-\frac{n}{N})}{N'}} \qquad (4)$$

Let $x = \frac{n}{N}$ and $d = \frac{n'}{N'}$. Equation (4) is converted into

$$\left(1 + \frac{1.64}{N'}\right)x^2 - \left(2d + \frac{1.64}{N'}\right)x + d^2 = 0 \qquad (5)$$

Equation (5) can be easily solved with two solutions x_1 and x_2, corresponding to the positive and negative squared root in equation (4). The greater solution x_2 should be chosen because it corresponds to the positive squared root. If there are $x_2 N$ negative tuples satisfying the rule before sampling, then it is unlikely that there are less than n' tuples satisfying the rule after sampling. Therefore, we use $x_2 N$ as the safe estimation of n. From the estimated n, we can estimate the accuracy of r based on equation (3).

The above presents the algorithm for building rules. Now we simply describe the classification process. To build rules of a certain class A, we set tuples in class A to be positive and others negative. We estimate the accuracy of each rule according to equation (3). After the rules for each class have been built, predictions can be made on target tuples. To predict the class label of a target tuple t, the most accurate rule that is satisfied by t is found, and the class label of that rule is used as the prediction.

5. Experimental Results

We have performed comprehensive experiments on both synthetic databases and real databases to show the accuracy and scalability of CrossMine. We compare CrossMine with FOIL [15] and TILDE [1] in every experiment, where the source code of FOIL and binary code of TILDE are from their authors. CrossMine and FOIL are run on a 1.7GHz Pentium 4 PC running on Windows 2000 Professional. TILDE is run on a Sun Blade 1000 workstation. Ten-fold experiments are used unless specified otherwise.

We use the following parameters for CrossMine. MIN_FOIL_GAIN = 2.5. MAX_RULE_LENGTH = 6. NEG_POS_RATIO = 1. MAX_NUM_NEGATIVE = 600, and we have found that the accuracy and running time of CrossMine are not sensitive to these parameters.

5.1. Synthetic Databases

To evaluate the scalability of CrossMine, a set of synthetic relational databases are generated. These databases mimic the real world relational databases. Our data generator takes the parameters shown in Table 1 to generate a database. The three columns of Table 1 represent the parameter name, description, and default value.

Name	Description	Def.
$\lvert R\rvert$	# relations	x
T_{min}	Min # tuples in each relation	50
T	Expected # tuples in each relation	y
A_{min}	Min # attributes in each relation	2
A	Expected # attributes in each relation	5
V_{min}	Min # values of each attribute	2
V	Expected # values of each attribute	10
F_{min}	Min # foreign-keys in each relation	2
F	Expected # foreign-keys in each relation	z
$\lvert r\rvert$	# rules	10
L_{min}	Min # complex predicates in each rule	2
L_{max}	Max # complex predicates in each rule	6
f_A	Prob. of a predicate on active relation	0.25

Table 1. Parameters of data generator.

To generate the database, we first generate a relational schema with $|R|$ relations, one being the target relation. The number of attributes of each relation obeys exponential distribution with expectation A and is at least A_{min}. One of the attributes is the primary-key. The number of values of each attribute (except the primary key) obeys exponential distribution with expectation V and is at least V_{min}. Besides these attributes, each relation has a few foreign-keys, pointing to the primary-keys of other relations. The number of foreign-keys of each relation obeys exponential distribution with expectation F and is at least F_{min}.

After the schema is generated, we generate rules that are lists of complex predicates. The number of complex predicates in each rule obeys uniform distribution between L_{min} and L_{max}. Each complex predicate has probability f_A to be on an active relation and probability $(1-f_A)$ to be on an inactive relation (involving a propagation). The class label of each rule is randomly generated, but the number of positive rules and that of negative rules differ by at most 20%.

The generated tuples are added to the database. The target relation has exactly T tuples. Each target tuple is generated according to a randomly chosen rule. In this way we also need to add tuples to non-target relations to satisfy the rule. After all target tuples are generated, we add more tuples to non-target relations. For each non-target relation R, the number of tuples obeys exponential distribution with expectation T and is at least T_{min}. If R already has enough tuples, we leave it unchanged. Otherwise we randomly generate tuples and add them to R until it has enough tuples.

We use "$Rx.Ty.Fz$" to represent a synthetic database with x relations, expected y tuples in each relation, and expected z foreign-keys in each relation.

For a multi-relational classification approach, we are most interested in its scalability w.r.t. the size of database schema, the number of tuples in each relation, and the number of joins involving each relation. Therefore, experiments are conducted on databases with different number of relations, different number of tuples in each relation, and different number of foreign-keys in each relation. In each experiment, the running time and accuracy of CrossMine, FOIL, and TILDE are compared.

To test the scalability w.r.t. the number of relations, five databases are created with 10, 20, 50, 100, and 200 relations respectively. In each database, the expected number of tuples in each relation is 500 and the expected number of foreign-keys in each relation is 2. Figure 9 shows the running time of the three approaches. Ten-fold experiments are used in most tests, and the average running time of each fold is shown in the figure. If the running time of an algorithm is close to or greater than 10 hours, only the first fold is tested in our experiments. We stop an experiment if the running time is much greater than 10 hours.

From the experimental results, one can see that CrossMine is thousands of times faster than FOIL and TILDE in most cases. Moreover, its running time is not affected much by the number of relations. FOIL and TILDE are not scalable with the number of relations. The running time of FOIL increases 9.6 times when the number of relations increases from 10 to 50, whereas the running time of TILDE increases 17.3 times. The accuracy of the three approaches are shown in Figure 10. One can see that CrossMine is more accurate than FOIL and TILDE.

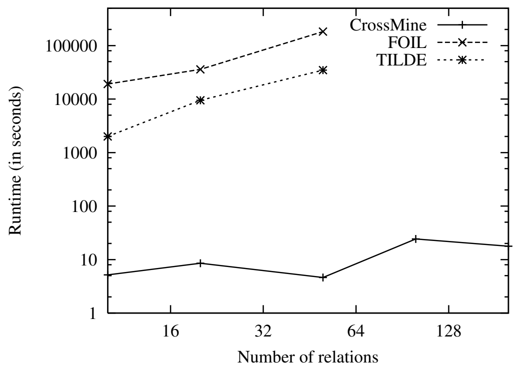


Figure 9. Runtime on R*.T500.F2.

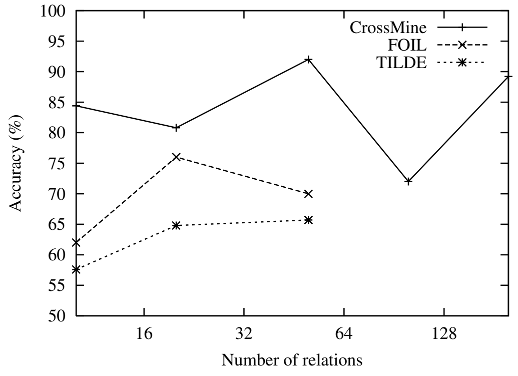


Figure 10. Accuracy on R*.T500.F2.

To test the scalability w.r.t. the number of tuples, five databases are created with the expected number of tuples in each relation being 200, 500, 1000, 2000, and 5000, respectively. The number of relations is 20 and the expected number of foreign-keys is 2. In this experiment, the performance of CrossMine with sampling is also tested to show the effectiveness of sampling. Figure 11 shows the running time of the four approaches.

One can see that CrossMine is more scalable than FOIL and TILDE. The running time of CrossMine increases 8.8 times when the number of tuples increases from 200 to 1000, while those of FOIL and TILDE increase 30.6 times and 104 times, respectively. Using sampling, CrossMine become more scalable (running time decreases to one third of non-sampling version when the number of tuples is 5000).

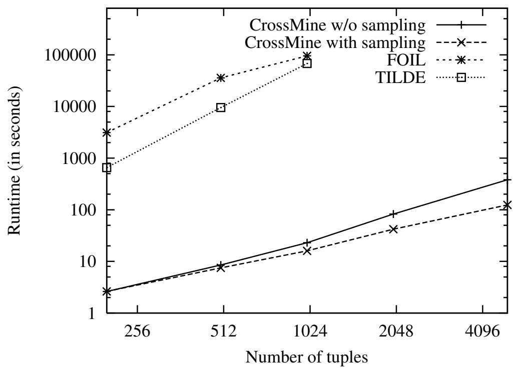


Figure 11. Runtime on R20.T*.F2.

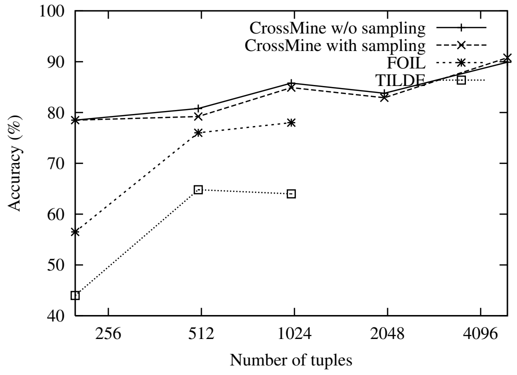


Figure 12. Accuracy on R20.T*.F2.

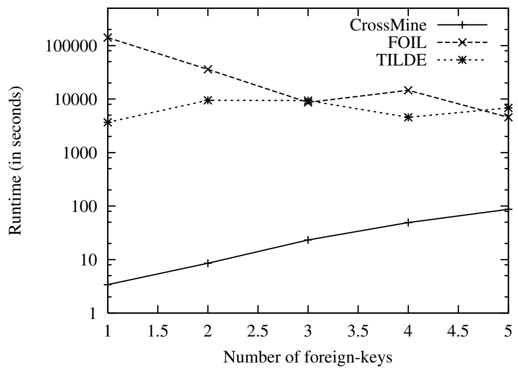


Figure 13. Runtime on R20.T500.F*.

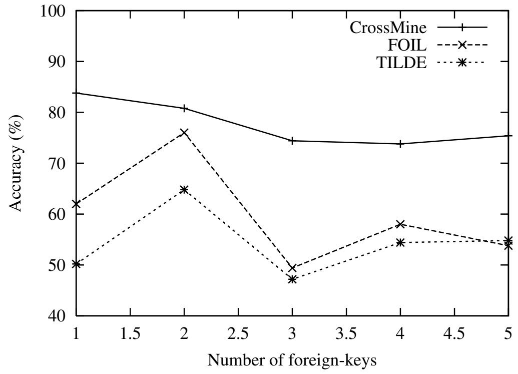


Figure 14. Accuracy on R20.T500.F*.

The accuracy of the three approaches is shown in Figure 12. CrossMine is more accurate than FOIL and TILDE, and the sampling method only slightly lowers the accuracy.

Finally, we test the scalability w.r.t. the number of foreign-keys. Again, five databases are created with the expected number of foreign-keys in each relation being 1 to 5. The number of relations is 20 and the expected number of tuples in each relation is 500. The running time of the three approaches are shown in Figure 13 and the accuracy are shown in Figure 14. One can see that CrossMine is not very scalable w.r.t. the number of foreign-keys, although it is still much more efficient than FOIL and TILDE. Fortunately, in most commercial databases the number of foreign-keys in each relation is pretty limited. And CrossMine is very efficient when this number is not large.

5.2. Real Databases

Experiments are also conducted on two real databases to compare the efficiency and accuracy of CrossMine, FOIL and TILDE. The first database is the financial database used in PKDD CUP 1999. Its schema is shown in Figure 1. We modify the original database by shrinking the $Trans$ relation which was extremely huge, and removing some positive tuples in $Loan$ relation to make the numbers of positive tuples and negative tuples more balanced. The final database contains eight relations and 75982 tuples in total. The Loan relation contains 324 positive tuples and 76 negative ones. The performances on this database is shown in Table 2.

The second database is the Mutagenesis database, which is a frequently used ILP benchmark. It contains four relations and 15218 tuples. The target relation contains 188 tuples, in which 124 are positive and 64 are negative. The Mutagenesis database is pretty small and the sampling method has no influences to CrossMine. The performances is shown in Table 3.

From the experiments one can see that CrossMine achieves good accuracy and efficiency. It is much more efficient than traditional ILP approaches, especially on databases with complex schemas.

Approach	Accuracy	Runtime
CrossMine w/o sampling	89.8%	17.4 sec
CrossMine with sampling	87.5%	13.9 sec
FOIL	74.0%	3338 sec
TILDE	81.3%	2429 sec

Table 2. Performances on the financial database of PKDD CUP'99.

Approach	Accuracy	Runtime
CrossMine	87.7%	1.92 sec
FOIL	79.7%	1.65 sec
TILDE	89.4%	25.6 sec

Table 3. Performances on the Mutagenesis database.

6. Conclusions and Future Work

Multi-relational classification is an important issue in data mining and machine learning involving large, real databases. It can be widely used in many disciplines, such as financial decision making, medical research, and geographical applications. Traditional ILP approaches are usually inefficient and unscalable for databases with complex schemas because they evaluate a huge number of rules when selecting predicates. In this paper we propose CrossMine, an efficient approach for multi-relational classification. It uses tuple ID propagation to reduce the computational cost dramatically, which makes CrossMine highly scalable w.r.t. the size of database schemas. Moreover, CrossMine uses a negative training data sampling method to reduce the cost of building rules from small numbers of positive examples, which makes CrossMine more scalable w.r.t. the number of tuples in each relation. The sampling method causes insignificant decrease in accuracy. In the process of building rules, CrossMine performs broader search than traditional ILP approaches by considering up to four predicates at a time. This enables CrossMine to identify good predicates and build more accurate rules. Experiments show that CrossMine is highly efficient comparing with the traditional ILP approaches, and it achieves high accuracy. These features make it appropriate for multi-relational classification in real world databases.

There are several possible extensions to CrossMine. Currently, CrossMine uses a sequential covering algorithm to build rules. Its efficiency might be improved by using the decision tree algorithm, which uses the idea of divide-and-conquer. Although CrossMine performs a quite broad search to build good rules, it is still a greedy algorithm and searches only a tiny part of the whole search space. It is interesting to study the application of other classification approaches (e.g. SVM, Neural Networks) on multi-relational environment to achieve better accuracy.

References

[1] H. Blockeel, L. De Raedt, and J. Ramon. Top-down induction of logical decision trees. In *Proc. 1998 Int. Conf. Machine Learning (ICML'98)*, Madison, WI, Aug. 1998.

[2] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–168, 1998.

[3] P. Clark and R. Boswell. Rule induction with CN2: Some recent improvements. In *Proc. 1991 European Working Session on Learning (EWSL'91)*, Porto, Portugal, Mar. 1991.

[4] L. Dehaspe and H. Toivonen. *Discovery of Relational Association Rules*. Springer-Verlag, 2000.

[5] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.

[6] J. Gehrke, R. Ramakrishnan, and V. Ganti. Rainforest: A framework for fast decision tree construction of large datasets. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, New York, NY, Aug. 1998.

[7] L. Getoor, N. Friedman, D. Koller, and B. Taskar. Learning probabilistic models of relational structure. In *Proc. 2001 Int. Conf. Machine Learning (ICML'01)*, Williamtown, MA, 2001.

[8] N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.

[9] T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[10] S. Muggleton. *Inductive Logic Programming*. Academic Press, New York, NY, 1992.

[11] S. Muggleton. Inverse entailment and progol. In *New Generation Computing, Special issue on Inductive Logic Programming*, 1995.

[12] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proc. 1990 Conf. Algorithmic Learning Theory*, Tokyo, Japan, 1990.

[13] A. Popescul, L. Ungar, S. Lawrence, and M. Pennock. Towards structural logistic regression: Combining relational and statistical learning. In *Proc. Multi-Relational Data Mining Workshop*, Alberta, Canada, 2002.

[14] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[15] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In *Proc. 1993 European Conf. Machine Learning*, Vienna, Austria, 1993.

[16] B. Taskar, E. Segal, and D. Koller. Probabilistic classification and clustering in relational data. In *Proc. 2001 Int. Joint Conf. Artificial Intelligence*, Seattle, WA, 2001.

[17] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Proc. 2002 Int. Conf. on Data Mining (ICDM'02)*, Maebashi, Japan, Dec. 2002.