CrossMine: Efficient Classification Across Multiple Database Relations[⋆]

Xiaoxin Yin[1], Jiawei Han[1], Jiong Yang[1], and Philip S. Yu[2]

[1] University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{xyin1, hanj, jioyang}@uiuc.edu
[2] IBM T.J. Watson Research Center, Yorktown Heights, N.Y. 10598, USA
psyu@us.ibm.com

Abstract. Most of today's structured data is stored in relational databases. Such a database consists of multiple relations that 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 including financial decision making and medical research. 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 "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 fail to achieve high scalability w.r.t. the number of relations in databases because they repeatedly join different relations to search for good literals.

In this paper we propose CrossMine, an efficient and scalable approach for multi-relational classification. CrossMine employs *tuple ID propagation*, a novel method for virtually joining relations, which enables flexible and efficient search among multiple relations. CrossMine also uses aggregated information to provide essential statistics for classification. A selective sampling method is used to achieve high scalability w.r.t. the number of tuples in the databases. Our comprehensive experiments on both real and synthetic databases demonstrate the high scalability and accuracy of CrossMine.

1 Introduction

Relational databases are the most popular format for structured data, and is thus the richest source of knowledge in the world. There are many real world applications involving decision making process based on information stored in relational databases, such as credit card fraud detection and loan application.

⋆ The work was supported in part by National Science Foundation under Grants IIS-02-09199/IIS-03-08215, 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.

Approaches that can perform in-depth analysis on relational data is of crucial importance in such applications. Therefore, multi-relational data mining has become a field with strategic importance.

There have been many important approaches for classification, such as neural networks [11] and support vector machines [6]. They can only be applied to data represented in single, “flat” relations. 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 [8]. Data stored in the same relation often have closer semantic relationship than those reachable via remote links. It is counter-productive to simply “convert” multi-relational data into a single flat data relation because such conversion may lead to the generation of a huge universal relation [8] but lose some essential semantic information carried by the semantic links in the database design.

Inductive Logic Programming (ILP) [12,10] is the most widely used category of approaches to multi-relational classification. There are many ILP approaches [1,3,4,5,13,14,15,18], which use very different philosophies in identifying hypotheses that fit the background knowledge. The ILP approaches achieve good classification accuracy. Unfortunately, most of them are not highly scalable w.r.t. the number of relations and the number of attributes in databases, thus are usually inefficient for databases with complex schemas.

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. To build a good multi-relational classifier, one needs to find good literals in each non-target relation R that help distinguish positive and negative target tuples. The target relation can usually join with every non-target relation via multiple join paths. Thus in a database with reasonably complex schema, there are a large number of join paths that need to be explored, each leading to dozens of literals in a certain relation. In order to identify the best literals and construct good clauses, many ILP approaches repeatedly join the relations along different join paths and evaluate literals based on the joined relation. This is very time consuming, especially when the joined relation contains much more tuples than the target one.

There are two major challenges in multi-relational classification: 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 linkages 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.

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) from 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 convenient and flexible method for virtually joining different relations, with as low cost as possible. Tuple IDs can be easily propagated between any two relations, which enables CrossMine to search freely in multiple relations for good literals and clauses. CrossMine obtains high efficiency and scalability by tuple ID propagation.

CrossMine uses a sequential covering algorithm, which repeatedly constructs clauses and removes positive examples covered by each clause. To construct a clause, it repeatedly searches for the best literal and appends it to the current clause. During the searching process, CrossMine limits the search space to relations related to the target relation or related to relations used in the clause. 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 larger than typical ILP approaches. By using tuple ID propagation and *look-one-ahead*, CrossMine considers literal sequences of length up to three at a time. It achieves both high efficiency and high accuracy by controlling the search space and identifying strong semantic links.

Unlike most previous approaches on multi-relational classification that only use simple literals, CrossMine uses both simple literals and literals involving aggregations on attribute values. For example, in the database of a CS department, a student's average grade or number of publications might be very important features for judging the academic performance of a student. The aggregations provide statistics about the target tuples, which often provide essential information for classification.

In many sequential covering algorithms, the negative examples are never removed in the clause building process, which makes the algorithm inefficient for databases with large numbers of tuples. It is common that before building a clause, there are much less positive examples than negative ones, which causes the algorithm to spend a large amount of time to build low-quality clauses. 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 achieve high scalability w.r.t. the number of tuples in databases. Our experiments show that the sampling method decreases the running time significantly but only slightly sacrifices the accuracy.

The remaining of the paper is organized as follows. In Section 2 we introduce the related work. The problem definition is presented in Section 3. Section 4 introduces the idea of tuple ID propagation and its theoretical background. We describe the algorithm and implementation issues in Section 5. Section 6 describes the negative tuple sampling technique. Experimental results are presented in Section 7. We made discussions in Section 8 and the study is concluded in Section 9.

2 Related Work

The most important category of approaches in multi-relational classification is ILP [12,10], which is defined 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 [18], Golem [14], and Progol [13]. FOIL is a top-down learner, which builds clauses that cover many positive examples and few negative ones. Golem is a bottom-up learner, which performs generalizations from the most specific clauses. Progol uses a combined search strategy. Some recent approaches TILDE [3], Mr-SMOTI [1], and RPTs [15] use the idea of C4.5 [17] and inductively construct decision trees from relational data. These approaches are usually more efficient than traditional ILP approaches due to the divide-and-conquer nature of decision tree algorithm.

Efficiency and scalability are two major issues in ILP. In [4] an approach was proposed to handle data stored on disks. In [5] the authors proposed an approach that can evaluate packs of queries which can be handled together. This approach is similar to CrossMine because both of them can utilize common prefix of different clauses. But CrossMine can propagate tuple IDs freely among different relations, which is more convenient in building clauses.

Besides ILP, probabilistic approaches [19,16] are also popular for multi-relational classification and modelling. Probabilistic relational models [19] is an extension of Bayesian networks for handling relational data, which can integrate the advantages of both logical and probabilistic approaches for knowledge representation and reasoning. In [16] an approach is proposed to integrate ILP and statistical modelling for document classification and retrieval.

We take FOIL as a typical example of ILP approaches and show its working procedure. FOIL is a sequential covering algorithm that builds clauses one by one. Each clause is built by repeatedly adding literal. At each step, every possible literal is evaluated and the best one is appended to the current clause. To evaluate a literal p, p needs to be appended to the current clause c to get a new clause c'. Then it constructs a new dataset which contains all target tuples satisfying c', evaluates p based on the number of positive and negative target tuples satisfying c'. For databases with complex schemas, the search space is huge and there are many possible literals at each step. Thus FOIL needs to repeatedly construct datasets by physical joins to find good literals, which is very time-consuming. This is also verified by our experiments.

3 Preliminaries

3.1 Basic Definitions

A database D consists of a set of relations, one of which is the target relation R_t, 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. The following types of joins are considered in CrossMine:

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.)

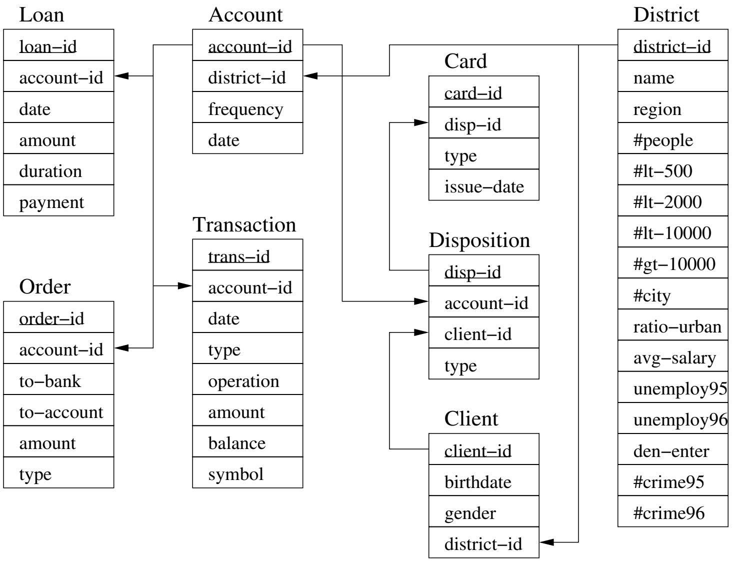


Fig. 1. The financial database from PKDD CUP 99

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.

CrossMine is a clause-based classifier on relational data. In general, each clause consists of a list of literals and the predicted class. Each literal is either a simple literal on the value of an attribute, or an aggregation literal on the aggregated value of an attribute.

3.2 Literals

In general, a literal is a constraint on a certain attribute in a certain relation. For example, literal "$l_1 = Loan(L, \_\,, \_\,, \_\,, >= 12, \_\,)$" means that the duration of loan L is no less than 12 months. In relational databases a literal is often defined based on a certain join path. For example, "$l_2 = Loan(L, A, \_\,, \_\,, \_\,, \_\,), Account\,(A, \_\,, monthly, \_\,)$" is defined on the join path $Loan \bowtie Account$, which means that the associated account of a loan has frequency "monthly".

There are two types of attributes: categorical attributes and numerical attributes. There are three types of literals:

1. **Categorical literal:** A categorical literal is defined on a categorical attribute. It is a constraint that this attribute must take a certain value, such as l_2 in the above example.

2. **Numerical literal:** A numerical literal is defined on a numerical attribute. It contains a certain value and a comparison operator, such as l_1, in the above example.
3. **Aggregation literal:** An aggregation literal is similar to a numerical literal, but is defined on the aggregated value of an attribute. It contains an aggregation operator, a certain value, and a comparison operator. For example, $l_3 = Loan(L, A, \_, \_, \_, \_), Order(\_, A, \_, \_, sum(amount) >= 1000, \_)$ is an aggregation literal, which requires the sum of amount of all orders related to a loan is no less than 1000. The following aggregation operators can be used: count, sum, avg.

3.3 Clauses

CrossMine is a clause-based classifier, which aims at finding clauses that distinguish positive examples from negative ones. Each clause contains a list of literals, associated with a class label. To integrate the join path into the clauses, CrossMine uses a form of clauses that is different from the traditional ILP approaches. Instead of using conventional literal, *complex literal* is used here as the element of clauses. A complex literal $\hat{l}$ contains two parts:

1. *prop-path*, i.e., propagation path, which 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 condition "$Loan.account\_id = Account.account\_id$". [1]
2. *constraint*: which indicates the constraint on the relation which 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 constraint is actually a literal that is either categorical, numerical, or involves aggregation.*

A complex literal is usually equivalent to two conventional literals. For example, the clause "$Loan(L, +) :- Loan(L, A, \_, \_, \_, \_), Account(A, \_, monthly, \_)$" can be represented by "$Loan(+) :- [Loan.account\_id \rightarrow Account.account\_id, Account.frequency = monthly]$".

A clause contains a list of literals. A target tuple satisfies a clause if and only if it satisfies every literal of the clause. To judge whether a target tuple t satisfies a clause c, one needs to join t with tuples in other relations according to the join path of c. We will introduce how to efficiently find out all target tuples satisfying a clause later.

We use the database in Figure 2 as an illustrative example. Suppose clause $c = Loan(+) :- [Loan.account\_id \rightarrow Account.account\_id, Account.frequency = monthly]$. We say a tuple t in *Loan* satisfies c 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 literal "$Account(A, \_, monthly, \_)$". So there are four tuples (with loan-id 1, 2, 4, and 5) in *Loan* that satisfy this clause.

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

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
124	monthly	960227
108	weekly	950923
45	monthly	941209
67	weekly	950101

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

3.4 Evaluation of Literals and Clauses

To generate a clause, CrossMine starts at an empty clause, keeps selecting the best literal and add it to the current clause. At each step, we need to evaluate every literal and select the best one. *Foil gain* is used [18] to measure the goodness of a literal.

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

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

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

Intuitively $foil\_gain(l)$ represents the total number of bits saved in representing positive examples by appending l to the current clause. It indicates how much the predictive power of the clause can be increased by appending l to it.

After generating a clause c, we need to evaluate c by estimating its accuracy. Suppose there are N^+ positive and N^- negative tuples satisfying c in the training set. The accuracy of c can be estimated using the method in [7], which is shown in the following equation:

$$Accuracy(c) = (N^+ + 1)/(N^+ + N^- + C) \tag{3}$$

where C is the number of classes.

4 Tuple ID Propagation

In this section we present the idea of tuple ID propagation and method of finding good literals with that. In essence, tuple ID propagation is a method for virtually joining non-target relations with the target relation. It is a convenient method that enables flexible search in relational databases, and is much less costly than physical join in both time and space.

4.1 Search for Literals by Joins

Consider the sample database in Figure 2. Suppose we want to compute the foil gain of literals in a non-target relation, such as *Account*. We need to find out for all positive and negative target tuples satisfying each literal l in the *Account* relation.

One approach is to join the two relations together and compute the foil gain of all literals, as shown in Figure 3. With the joined relation, the foil gain of every literal in both relations can be computed. To compute the foil gain of all literals 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 [17]. To find the best literal 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 literals "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	+

Fig. 3. The join of *Loan* and *Account*

It is quite expensive to use physical joins to evaluate literals for the following two reasons. First, in a database with complex schema, there are usually a large number of join paths that need to be explored. For example, in the database shown in Figure 1, *Loan* can join with *Account*, *Order*, *Transaction* and *Disposition*. Each of the four relations can join with several other relations, such as *Disposition* that can join with *Card*, *Client*, or back to *Account* and *Order*. Therefore one needs to repeatedly perform physical joins and create many joined relations. Second, there may be much more tuples in a joined relation than in the target relation. For example, a loan may join with several orders or dozens of transactions. Thus the joined relation may contain a large number of tuples when the join path is long.

The above two challenges prevent most traditional ILP approaches from efficiently searching among different relations. In the next section we will introduce *tuple ID propagation*, a technique that enables free search in relational databases. When searching for good literals, one can propagate tuple IDs from any relation that IDs have been propagated to, which requires much less computation and data transfer. The tuple IDs can be easily propagated between any two relations, which makes it possible to "navigate freely" among different relations.

4.2 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.

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−

Fig. 4. Example of tuple ID propagation

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

Lemma 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 clause (others have been eliminated). The current clause contains a literal "$R_1(R_1.id,\ R_1.A,\ \cdots)$", 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 clause.*

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 clause.*

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* *clause (others have been eliminated). If R_1's IDs are propagated to R_2, then the foil gain of every literal in R_2 can be computed using the propagated IDs on R_2.*

Proof. *Given the current clause c, for a literal l in R_2, such as $R_2.B = b$, its foil gain can be computed based on $P(c)$, $N(c)$, $P(c+l)$ and $N(c+l)$. $P(c)$ and $N(c)$ should have been computed during the process of building the current clause. $P(c+l)$ and $N(c+l)$ 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 clause); and (3) count the number of positive and negative tuples found in (2).*

For example, suppose "$Loan(L, +) :- Loan(L, A, \_, \_, \_, \_)$" is the current clause. For literal "$Account(A, \_, monthly, \_)$", we can first find out tuples in the *Account* relation that satisfy this literal, which are {124, 45}. Then we can find out tuples in the *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 literal "$Account(A, \_, monthly, \_)$".

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 lemma.

Lemma 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 clause (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 clause). 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 clause, 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 clause, plus the join $R_2.A = R_3.A$) if and only if $t.id \in idset(u)$.*

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

4.3 Analysis and Constraints

The idea of *label propagation* was proposed in [2], which propagates class labels along join paths for evaluating literals. This approach is effective for n-to-1 relationships. But for join paths that involve 1-to-n or n-to-n relationships, it cannot find the numbers of positive and negative target tuples satisfying each literal. For example, suppose there are 10 tuples in the *Loan* relation, 5 being positive and 5 being negative. 4 positive and 5 negative tuples are joinable with 1 account each, while the other positive tuple is joinable with 10 accounts. Suppose all above accounts satisfy a literal l. Then one can see that there are 5 positive and 5 negative target tuples satisfying l, indicating that l has low foil gain. However, if only class labels are propagated, we will not be able to distinguish class labels from different target tuples, and will say that there are 14 positive and 5 negatives tuples satisfying l, indicating that l has high foil gain. A real database usually contains many 1-to-n and n-to-n relationships, thus one needs to propagate IDs instead of labels when building classifiers.

Tuple ID propagation is a way to perform virtual join. Instead of physically joining 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 literals can be evaluated 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, literals 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. Moveover, a relation may be associated with multiple set of IDs corresponding to different join paths. This enables CrossMine to search for good literals freely across relations.

ID propagation, though valuable, should be enforced with certain constraints. 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. Therefore, our system discourages propagation if the current link has very large fan-out.

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 such links.

5 Clause Generation

In this section we present CrossMine's algorithm for generating clauses by tuple ID propagation. A sequential covering algorithm is developed that repeatedly builds clauses and removes positive tuples satisfying the clause. To build a clause, it repeatedly searches for the best literal and adds it to the current clause. This algorithm is selected because it guarantees the quality of each clause by always keeping a large number of negative examples, and moreover, its greedy nature makes it efficient in large databases.

5.1 Finding Best Literal

Suppose CrossMine is searching for the best literal in a certain relation R, and tuple IDs have been propagated to R so that one will know the target tuples joinable with each tuple in R. To find the best literal in R, CrossMine evaluates the literals in each attribute of R. Different algorithms are used for categorical and numerical attributes.

Suppose the best literal on a categorical attribute A_c is to be found. Suppose A_c has l values $a_1, \ldots, a_l$. For each value a_i, a literal $l_i = [R.A_c = a_i]$ is built. Then CrossMine scans the values of each tuple on A_c to find out the numbers of all positive and negative target tuples satisfying each literal l_i. With this information, the foil gain of each l_i can be computed and the best literal can be found.

Suppose the best literal on a numerical attribute A_n is to be found, and a sorted index for values on A_n has been built beforehand. CrossMine iterates from the smallest value of A_n to the largest value. When iterating to each value v_i, all tuples having value v_i are found, and their associated IDs are added into a pool. This pool of IDs represent all target tuples satisfying literal $[A_n \leq v_i]$. In this way, one can compute the foil gain of every literal of the form $[A_n \leq v_i]$ for every value v_i of A_n. Then CrossMine iterates from the largest value to the smallest value to evaluate the literals of the form $[A_n \geq v_i]$. In this way the best numerical literal can be found for A_n.

To search for the best aggregation literal for A_n, CrossMine first finds some statistics for each target tuple. By scanning the tuple IDs associated with tuples in R, for each target tuple t^*, CrossMine can find the tuples in R joinable with t^*, and calculate the count, sum, and average of the values of those tuples on A_n. Then CrossMine computes the foil gain of all aggregation literals using an approach similar to the approach for finding best numerical literals. In this way the best aggregation literal can be found.

5.2 Clause Generation Algorithms

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

To build a clause, one repeatedly searches for the best complex literal and appends it to the current clause, until the stop criterion is met. A relation is *active* if it appears in the current clause, 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 literal. The algorithm is shown in Figure 6.

The following procedure is used to find the best literal: (1) for every active relation $\hat{R}$, find the best complex literal 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 literal on

```
Algorithm 1. Find-Clauses
Input: a relational database $D$ with a target relation $R_t$.
Output: a set of clauses for predicting class labels of target tuples.

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

Fig. 5. Algorithm *Find-Clauses*

```
Algorithm 2. Find-A-Clause
Input: a relational database $D$ with a target relation $R_t$.
Output: a clause for predicting class labels of target tuples.

Procedure
  clause $c \leftarrow$ empty-clause;
  set $R_t$ to active;
  do
    Complex literal $l \leftarrow$ Find-Best-Literal();
    if $foil\_gain(l) <$ MIN_FOIL_GAIN;
    then break;
    else
      $c \leftarrow c + l$;
      remove all target tuples not satisfying $c$;
      update IDs on every active relation;
      if $l.constraint$ is on an inactive relation
      then set that relation active;
  while($c.length <$ MAX_CLAUSE_LENGTH);
  return $c$;
```

Fig. 6. Algorithm *Find-A-Clause*

$\bar{R}$. Consider the database in Figure 1. Originally only *Loan* is active. Suppose the first best complex literal 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 literal.

The idea behind the algorithm of building a clause is as follows. Starting from the target relation R_t, find the best complex literal $\hat{l}$, 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 literal. This algorithm is greedy in nature. It extends the clause using only those literals in either the active relations or the relations directly joinable with an active relation.

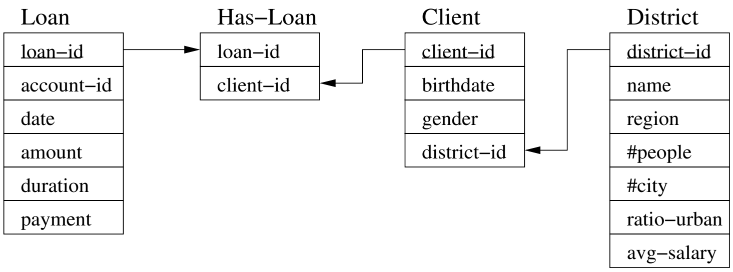


Fig. 7. Another sample database

The above algorithm may fail to find good literals 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 literal in the *Has_Loan* relation. Therefore, the clauses built will never involve any literals on the *Client* relation and the *District* relation.

This problem can be solved using the *look-one-ahead* method. When searching for the best literal, 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 literals in $\bar{R}'$. By this method, in the example in Figure 7, one can find clauses 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 literal in $\bar{R}$, using the approach in [9]. The algorithm for searching for the best complex literal is shown in Figure 8.

The above algorithms show the procedure of building clauses in CrossMine. The basic idea of building a clause is to start from the target relation, keep appending literals 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 literals in the database, and builds good clauses with 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

Algorithm 3. Find-Best-Literal
Input: a relational database D with a target relation R_t, and current clause c.
Output: the complex literal with most foil gain.

```
Procedure
  Complex literal $l_{max} \leftarrow empty$;
  for each active relation $\hat{R}$
    Complex literal $l \leftarrow$ best complex literal in $\hat{R}$;
    if $foil\_gain(l) > foil\_gain(l_{max})$
    then $l_{max} \leftarrow l$;
  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 $\bar{R}$;
            $l \leftarrow$ best complex literal in $\bar{R}$;
            if $foil\_gain(l) > foil\_gain(l_{max})$
            then $l_{max} \leftarrow l$;
            for each foreign-key $k' \neq k$ of $\bar{R}$
                  propagate IDs from $\bar{R}$ to relation $\bar{R}'$
                    that is pointed to by $\bar{R}.k$;
                  $l \leftarrow$ best complex literal in $\bar{R}'$;
                  if $foil\_gain(l) > foil\_gain(l_{max})$
                  then $l_{max} \leftarrow l$;
  return $l_{max}$;
```

Fig. 8. Algorithm *Find-Best-Literal*

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 clauses. However, the search spaces of those approaches are usually much smaller than that of CrossMine. By using complex literals, CrossMine considers two literals at a time (one for join and another for value constraint). By using look-one-ahead, it can consider up to three literals together in clause generation. This enables CrossMine to find good literals 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.

5.3 Predicting Class Labels with Clauses

After generating clauses, CrossMine needs to predict the class labels of unlabelled target tuples. CrossMine also needs to predict the class labels of the tuples in the training set to estimate the accuracy of each clause. Therefore, an efficient algorithm is needed for finding out all target tuples satisfying each clause.

CrossMine uses an efficient algorithm based on tuple ID propagation to find out all target tuples satisfying a certain clause c. Suppose $c = R_t(+)\ :-\ l_1, l_2, \ldots, l_k$. ($l_i$ $(1 \le i \le k)$ is a complex literal.) The main idea of the algorithm is to propagate the IDs of all target tuples along the prop-path of each literal l_i, and prune all IDs of target tuples not satisfying the constraint of l_i.

To illustrate this procedure, let us examine an example. Suppose $c = Loan(+)\ :-\ [Loan.account\_id \rightarrow Account.account\_id,\ Account.frequency = monthly],\ [Account.district\_id \rightarrow District.district\_id,\ avg\_salary > 80000]$. First, the IDs of all target tuples are propagated to the *Account* relation via the prop-path *Loan.account_id* $\rightarrow$ *Account.account_id.* All target tuples whose associated account has value "monthly" on attribute *frequency* are found, and the IDs of all the other tuples are pruned. Then the remaining IDs are propagated to the *District* relation via the prop-path *Account.district_id* $\rightarrow$ *District. district_id*, and target tuples satisfying the second literal are found, which are all tuples satisfying this clause.

Given a set of target tuples whose class labels need to be predicted, CrossMine first finds out the tuples satisfying each clause. For each target tuple t, the most accurate clause that is satisfied by t is found, and the class label of that clause is used as the predicted class. If multiple classes are presented in the training set, then for each class C, CrossMine takes tuples of C as positive tuples and all the other tuples as negative ones to build clauses for class C. The same algorithm is used for predicting the class labels of unseen tuples.

6 Tuple Sampling

From Algorithm 1 we can see that during the procedure of building clauses, the number of positive tuples keeps decreasing and the number of negative tuples remains unchanged. Each clause covers a certain proportion of the remaining positive tuples (usually 5% to 20%), thus the first several clauses 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 clause because all the negative tuples remain there.

Let $c.sup^+$ and $c.sup^-$ be respectively the number of positive and negative tuples satisfying a clause c. Let $c.bg^+$ and $c.bg^-$ be respectively the number of positive and negative tuples satisfying c when c is built. The accuracy of c can be estimated using the method in [7], which is shown in the following equation:

$$Accuracy(c) = (c.sup^+ + 1)/(c.sup^+ + c.sup^- + C) \tag{4}$$

where C is the number of classes.

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

Based on this observation, the following method is proposed to improve its effectiveness. Before a clause 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 clause generated usually covers an even smaller portion of the remaining positive tuples. The possible reason is that, there are usually many "special positive cases" that cannot be covered by any good clause. The consequence is that the number of generated clauses usually increases with the number of target tuples. When sampling is not used, the time for building each clause is proportional to the total number of target tuples. Thus the total runtime increases sharply as the number of tuples increases, because more clauses are needed and longer time is used for building each clause. When sampling is used, the time for building a clause is proportional to the number of remaining positive tuples. Because the first several clauses can often cover the majority of positive tuples, the total number of tuples decreases sharply after finding them, and the algorithm becomes highly scalable.

When sampling is used, the accuracy of clauses should be estimated in a different way. Suppose before building clause c, there are P positive and N negative tuples. N' negative tuples are randomly chosen by sampling ($N' < N$). After building clause c, suppose there are l positive and n' negative tuples satisfying c. We need to estimate n, the number of negative tuples satisfying c. The simplest estimation is $n \approx n'\frac{N}{N'}$. However, this is not a safe estimation because it is quite possible that c 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 c. Consider the event of a negative tuple satisfying c 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

$$\frac{n'}{N'} = \frac{n}{N} - 1.28\sqrt{\frac{\frac{n}{N}(1-\frac{n}{N})}{N'}} \tag{5}$$

Let $x = \frac{n}{N}$ and $d = \frac{n'}{N'}$. Equation (5) is converted into

$$\left(1 + \frac{1.64}{N'}\right)x^2 - \left(2d + \frac{1.64}{N'}\right)x + d^2 = 0 \tag{6}$$

Equation (6) can be easily solved with two solutions x_1 and x_2, corresponding to the positive and negative squared root in equation (5). 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 clause before sampling, then it is unlikely that there are less than n' tuples satisfying the clause after sampling. Therefore, we use $x_2 N$ as the safe estimation of n. From the estimated n, we can estimate the accuracy of c based on equation (4).

7 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 [18] and TILDE [3] 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.

The following parameters are used in our experiments for testing CrossMine: MIN_FOIL_GAIN = 2.5, MAX_CLAUSE_LENGTH = 6, NEG_POS_RATIO = 1, and MAX_NUM_NEGATIVE = 600. Moreover, we have found that the accuracy and running time of CrossMine are not sensitive to these parameters.

7.1 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.

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. All attributes are categorical, and 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 clauses that are lists of complex literals. The number of complex literals in each clause obeys uniform distribution between L_{min} and L_{max}. Each complex literal has probability f_A to be on an active relation and probability $(1 - f_A)$ to be on an inactive relation (involving a propagation). Only categorical literals are used. The class label of each clause is randomly generated, but the number of positive clauses and that of negative clauses differ by at most 20%.

Table 1. Parameters of data generator

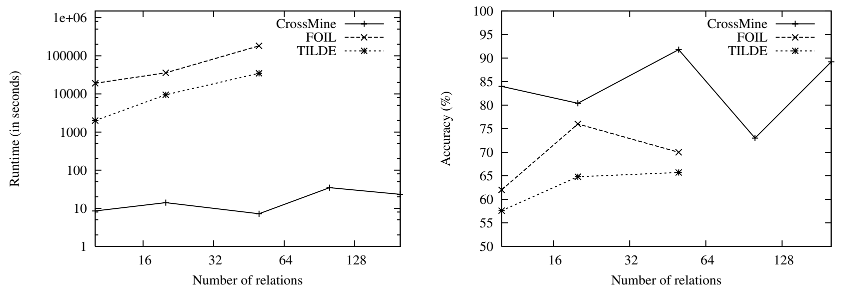
Name	Description	Def.
$\lvert R \rvert$	# relations	x
T_{min}	Min # tuples in each relation	50
T	Expected # tuples in each relation	y
A_{min}	Min # attributes in each relation	2
A	Expected # attributes in each relation	5
V_{min}	Min # values of each attribute	2
V	Expected # values of each attribute	10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

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 clause. In this way we also need to add tuples to non-target relations to satisfy the clause. 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.

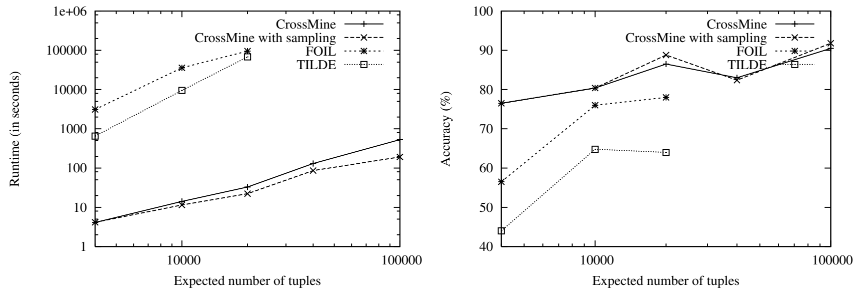
To test the scalability w.r.t. the number of relations, five databases are created with 10, 20, 50, 100, and 200 relations respectively. In each database, the expected number of tuples in each relation is 500 and the expected number of foreign-keys in each relation is 2.

Figure 9 (a) 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



a) Time vs. #relation b) Accuracy vs. #relation

Fig. 9. Runtime and accuracy on R*.T500.F2



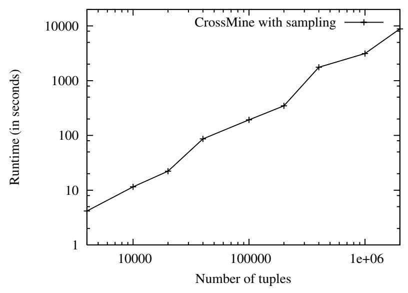
a) Time vs. # tuple b) Accuracy vs. # tuple

Fig. 10. Runtime and accuracy on R20.T*.F2

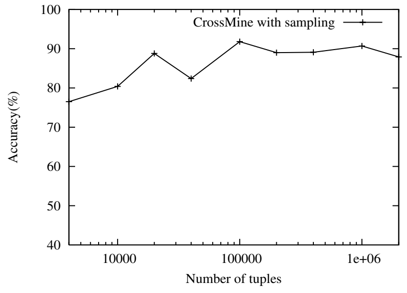
running time of FOIL increases 9.6 times when the number of relations increases from 10 to 50, whereas the running time of TILDE increases 17.3 times. The accuracy of the three approaches are shown in Figure 9 (b). One can see that CrossMine is more accurate than FOIL and TILDE.

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. There are twenty relations in each dataset, thus the expected number of tuples range from 4K to 100K. The expected number of foreign-keys in each relation is 2. In this experiment, the performance of CrossMine with sampling is also tested to show the effectiveness of sampling. Figure 10 (a) 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 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. With tuple sampling, CrossMine becomes more scalable (running time decreases to one third of non-sampling version when the number of tuples is 5000). The accuracy of the three approaches is shown in Figure 10 (b). CrossMine is more accurate than FOIL and TILDE, and the sampling method only slightly sacrifices the accuracy.

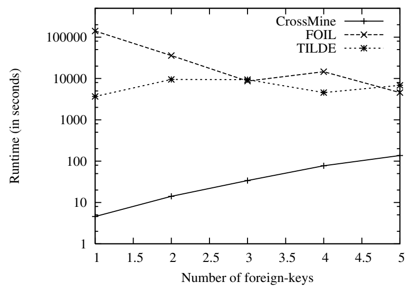


a) Time vs. # tuple

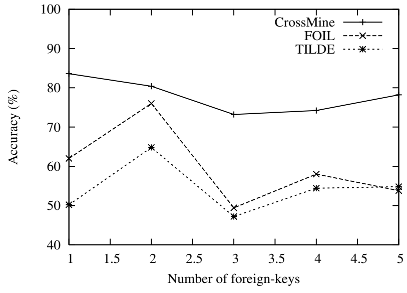


b) Accuracy vs. # tuple

Fig. 11. Runtime and accuracy on large datasets



a) Time vs. #foreign-key



b) Accuracy vs. #foreign-key

Fig. 12. Runtime and accuracy on R20.T500.F*

We also test CrossMine (with negative sampling) on large datasets to show its high scalability. We generate nine datasets with expected number of tuples in each relation from 200 to 100K. Since there are twenty relations in each dataset, the expected numbers of tuples range from 4K to 2M. The running time and accuracy of CrossMine are shown in Figure 11 (a) and (b). It can be seen that CrossMine is highly scalable for large datasets.

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 12 (a) and the accuracy are shown in Figure 12 (b). 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 quite limited. And CrossMine is very efficient when this number is not large.

7.2 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

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

Approach	Accuracy	Runtime
CrossMine w/o sampling	89.5%	20.8 sec
CrossMine with sampling	88.3%	16.8 sec
FOIL	74.0%	3338 sec
TILDE	81.3%	2429 sec

Table 3. Performances on the Mutagenesis database

Approach	Accuracy	Runtime
CrossMine	89.3%	2.57 sec
FOIL	79.7%	1.65 sec
TILDE	89.4%	25.6 sec

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 the *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. All three types of literals are considered in this experiment.

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.

8 Discussions

In this paper it is assumed that the dataset can fit in main memory, so that random access can be performed on tuples in different relations. In some real applications the dataset cannot fit in main memory. Instead, the data are stored in a relational database in the secondary storage. However, this will not affect the scalability of CrossMine. In this section we show that all the operations of CrossMine can be performed efficiently on data stored on disks.

8.1 Tuple ID Propagation

Tuple ID propagation is the most basic operation of CrossMine. When data is in main memory, a set of tuple IDs associated with a relation R are stored in a separate array. When data cannot fit in main memory, we can store a set of tuple IDs as an attribute of R. Since CrossMine limits the fan-out of tuple ID propagation (Section 4.3), the number of IDs associated with each tuple is limited, thus the IDs can be stored as a string of fixed or variable length.

In CrossMine, only joins between keys or foreign-keys are considered (Section 3.1). An index can be created for every key or foreign key. When propagating IDs from R_1 to R_2, only the tuple IDs and the two joined attributes are needed. If one of them can fit in main memory, this propagation can be done efficiently. Otherwise, a join operation can be performed between R_1 and R_2 to find joinable tuples and propagated IDs.

8.2 Evaluating Literals

Suppose tuple IDs have been propagated to a relation R, and the best literal on R need to be identified. If all attributes of R are categorical, then the numbers of positive and negative target tuples satisfying every literal can be calculated by one sequential scan on R. With this sequential scan, we can also generate simple statistics (sum, average, etc.) for every target tuple and every numerical attribute. The best aggregation literal can be found by these statistics. For a numerical attribute A, suppose a sorted index has been built on A. Then a sorted scan on A is needed to find the best literal on A. If this index and the tuple IDs can fit in main memory, this can be done efficiently.

9 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. Many traditional ILP approaches are inefficient and unscalable for databases with complex schemas because they evaluate a huge number of clauses when selecting literals. 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. In the process of building clauses, CrossMine performs search in wider space than traditional ILP approaches by considering up to three literals at a time. This enables CrossMine to identify better-quality literals and build more accurate clauses. 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. Although CrossMine searches a wider space to select better-quality clauses than most ILP approaches, it is still a greedy algorithm and searches only a small part of the whole search space. Moreover, it is interesting to study how to integration CrossMine methodology with other classification methods (such as SVM, Neural Networks, and k-nearest neighbors) in the multi-relational environment to achieve even better accuracy and/or scalability.

References

1. A. Appice, M. Ceci, and D. Malerba. Mining model trees: a multi-relational approach. In *Proc. 2003 Int. Conf. on Inductive Logic Programming*, Szeged, Hungary, Sept. 2003.
2. J. M. Aronis, F. J. Provost. Increasing the Efficiency of Data Mining Algorithms with Breadth-First Marker Propagation. In *Proc. 2003 Int. Conf. Knowledge Discovery and Data Mining*, Newport Beach, CA, 1997.
3. H. Blockeel, L. De Raedt, and J. Ramon. Top-down induction of logical decision trees. In *Proc. 1998 Int. Conf. Machine Learning*, Madison, WI, Aug. 1998.
4. H. Blockeel, L. De Raedt, N. Jacobs, and B. Demoen. Scaling up inductive logic programming by learning from interpretations. *Data Mining and Knowledge Discovery*, 3(1):59-93, 1999.
5. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135-166, 2002.
6. C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Mining and Knowledge Discovery*, 2:121–168, 1998.
7. P. Clark and R. Boswell. Rule induction with CN2: Some recent improvements. In *Proc. 1991 European Working Session on Learning*, pages 151–163, Porto, Portugal, Mar. 1991.
8. H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002.
9. 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*, New York, NY, Aug. 1998.
10. N. Lavrac and S. Dzeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.
11. T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.
12. S. Muggleton. *Inductive Logic Programming*. Academic Press, New York, NY, 1992.
13. S. Muggleton. Inverse entailment and progol. In *New Generation Computing, Special issue on Inductive Logic Programming*, 1995.
14. S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proc. 1990 Conf. Algorithmic Learning Theory*, Tokyo, Japan, 1990.
15. J. Neville, D. Jensen, L. Friedland, and M. Hay. Learning Relational Probability Trees. *Proc. 2003 Int. Conf. Knowledge Discovery and Data Mining*, Washtington, DC, 2003.
16. 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.
17. J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
18. J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In *Proc. 1993 European Conf. Machine Learning*, Vienna, Austria, 1993.
19. 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.