Tomasz Imielinski and Heikki Mannila

A Database Perspective on Knowledge Discovery

The concept of data mining as a querying process and the first steps toward efficient development of knowledge discovery applications are discussed.

Database mining is not simply another buzzword for statistical data analysis or inductive learning. Database mining sets new challenges to database technology: new concepts and methods are needed for query languages, basic operations, and query processing strategies. The most important new component is the ad hoc nature of knowledge and data discovery (KDD) queries and the need for efficient query compilation into a multitude of existing and new data analysis methods. Hence, database mining builds upon the existing body of work in statistics and machine learning but provides completely new functionalities.

The current generation of database systems are designed mainly to support business applications. The success of Structured Query Language (SQL) has capitalized on a small number of primitives sufficient to support a vast majority of such applications. Unfortunately, these primitives are not sufficient to capture the emerging family of new applications dealing with knowledge discovery.

Most current KDD systems offer isolated discovery features using tree inducers, neural nets, and rule discovery algorithms. Such systems cannot be embedded into a large application and typically offer just one knowledge discovery feature. The same is true for most online analytical processing (OLAP) tools. We call such systems *first-generation* database mining systems. The current situation is very similar to the situation in database management systems in the early 1960s, when each application had to be built from scratch, without the benefit of dedicated database primitives provided later by SQL and relational database APIs (Application Programming Interface). In fact, today's techniques of data mining would more appropriately be described as "file mining" since they assume a loose coupling between a data-mining engine and a database. In most cases, this interface has a form of two simple commands: "read from" and "write to", just as Cobol programs interacted with large data files 30 years ago. Additionally, the KDD field currently suffers from lack of identity: there is no consensus on whether KDD is an area of its own.

Some claim knowledge discovery is simply machine learning with large data sets, and that the database component of the KDD is essentially maximizing performance of mining operations running in the top of large persistent data sets and involving expensive I/O. It is, of course, important work to close the gap between the inductive learning tools and the database systems currently available. However, although improving performance is an important issue, it is probably not sufficient to trigger a qualitative change in system capabilities.

Moreover, many database management system (DBMS) performance enhancements important for database mining are also desirable in general. Such features include parallel query execution, in-memory evaluation as well as optimized support for sampling and aggregate query operations such as "Count," "Average," and "Variance." These features are of general interest and their importance to database mining can be viewed only as a side effect of basic research in DBMS. Moreover, incremental improvements of existing DBMSs to better suit KDD applications will most likely be insufficient, since the current DBMSs were primarily targeted at the different classes of applications.

A historical analogy is perhaps in order; one may argue that performance improvement of I/O operations alone would have never triggered the DBMS research field nearly 30 years ago. Query languages, query optimization, and transaction processing were the driving ideas behind the tremendous growth of database field in the last three decades. To be more specific, it was the ad hoc nature of querying that created a challenge to build general-purpose query optimizers. If queries were predefined and their number was limited it would be sufficient to develop highly tuned, stand-alone, library routines. We believe database mining can follow a similar path of development.

New Research Frontier

The following are two research scenarios: a short-term vision—that is strictly performance driven, and well under way—and another, long term, that is only beginning now and possibly defines a new frontier for database research.

Research Program—Short Term

The short-term research program calls for efficient algorithms implementing machine learning tools on the top of large databases and utilizing the existing DBMS support.

The implementation of classification algorithms (say, C4.5) or neural networks on top of a large database requires tighter coupling with the database system and intelligent use of indexing techniques [4, 6]. For example, training a classifier on a large training set stored in a database may require multiple passes through the data using different orderings between attributes. This can be implemented by utilizing DBMS support for aggregate operations, indexes and database sorting (order by). Clustering may require efficient implementations of nearest neighbor algorithms on the top of large databases. Finally, generation of association rules can be performed in many different

TERRY WIDENER

ways, depending on the amount of main memory available. There have been a growing number of papers on this subject at recent VLDB and SIGMOD conferences. Some of these techniques are finding their way into products such as IBM's Intelligent Miner.

Research Program—Long Term

Database mining should learn from the general experience of DBMS field and follow one of the key DBMS paradigms: building optimizing compilers for ad hoc queries and embedding queries in application programming interfaces. Thus, the focus should be on increasing programmer productivity for KDD application development. In fact, there is a growing need for Knowledge and Data Discovery Management Systems (KDDMS), or *second-generation* database mining systems, to manage KDD applications just as DBMSs successfully manage business applications.

$$\begin{aligned}
&X.\text{Diagnosis} = \text{"heart disease"} \wedge X.\text{Sex} = \text{"male"} \\
&\wedge X.\text{City} = Y \wedge Y.\text{Population} > 500{,}000 \\
&\Rightarrow X.\text{Age} > 50\ [600, 0.55] \\
&X.\text{Diagnosis} = \text{"heart disease"} \wedge X.\text{Age} < 50 \\
&\Rightarrow X.\text{BMI} > 27\ [300, 0.80]
\end{aligned}$$

Figure 1. Examples of rules

Queries in KDDMS, however, have to be much more general than SQL; similarly, the queried objects have to be far more complex than records (tuples) in relational database. To achieve this, we define KDD objects, KDD queries, and bring back the concept of closure of a query language as a basic design paradigm. By a KDD object we mean a rule, a classifier, or a clustering.

Rules, which are defined more precisely in the next section, are essentially probabilistic formulas or multidimensional correlations (see Figure 1). Classifiers are defined using, for example, classification trees, neural networks or multidimensional regression. Clusterings refer to collections of sets of objects, in which each set consists of objects grouped together by proximity with respect to a predefined distance metric.

By a KDD query we mean a predicate which returns a set of objects that can either be KDD objects or database objects such as records or tuples. The KDD objects typically will not exist a priori, thus querying the KDD objects requires (rule, classification, cluster) generation at run time. KDD objects may also be pregenerated and stored in a "inductive" database, such as metadata. For example, a rule base can store pregenerated associations [10], or rules [8]. In such cases querying can be reduced to retrieval. In general, a KDDMS should be able to persistently store and manage the KDD objects as well as provide the ability to query them. Thus, querying has two major roles: generation of new KDD objects and retrieval of the ones which were generated before.

In relational databases, the result of a query is a relation that can be queried further. In fact, a relational query "does not care" if its argument is the original database or if it is a result of another query. This is typically referred to as a closure principle, and it is one of the most important design principles (although SQL does not fully follow it).

We also would like a KDD query language to satisfy such a closure principle. Thus, a result of a KDD query may be an argument of another compatible type of KDD query. We would like relational queries to form a proper subset of KDD queries. In principle, then, a KDD query can be nested within a regular relational query. This is illustrated in Figure 2. Some examples of KDD queries:

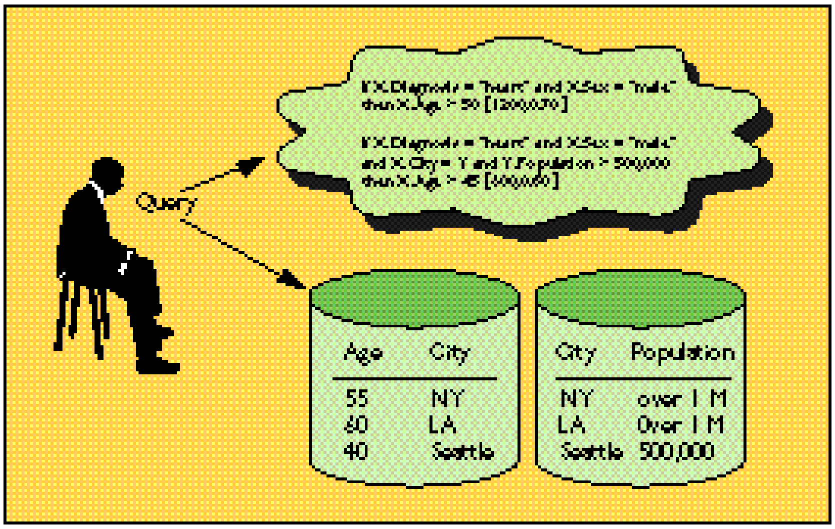
- Generate a classifier (say, by a decision tree) trained on a user-defined training set (specified through a database query) with user-defined attributes and user-specified classification categories. Then find all records in the database wrongly classified using that classifier and use that set as a training data for another classifier.
- Generate the strongest rule (according to some predefined criterion) with user-specified attributes occurring in the body and in the consequent. Find tuples that violate this rule and generate rules satisfied by such set of tuples.
- Generate all rules with consequent attribute values computed by an SQL query. This illustrates that KDD queries may not be completely known at a compile time.
- Find tuples that belong to the largest cluster in a clustering constructed according to user-specified distance metrics.

KDD queries may cross the border between tuples (database objects) and KDD objects several times possibly using multiple layers of nesting. KDD queries can also be embedded in a host programming environment to provide application programming interfaces for knowledge and data discovery applications just as SQL queries can be embedded in host languages such as C or Cobol.

The proposed research program is as follows: First a KDD query language has to be formally defined (prompting questions about desired expressive power, and so forth), then query optimization tools would be developed to compile queries into reasonably efficient execution plans. These execution plans will include existing inductive learning and statistical data analysis algorithms and may include new inductive tools as well. Note how this plan essentially mirrors the development of query languages and query optimization in relational databases.

Thus, in our view, database mining is not simply another buzzword for statistical data analysis or inductive learning on large data sets. The key new component is the ad hoc nature of KDD queries and the need for efficient query compilation into a multitude of existing and new data analysis methods. Hence, database mining builds upon the existing body of work in statistics and machine learning, but provides completely new functionalities.

KDD query optimization will be more challenging than relational query optimization due to the higher expressive power of KDD queries. (Some general approaches such as pushing selections down the query trees can be used here as well. In this case, some selections may be pushed prior to the rule or classification tree generation.)



If X.Diagnosis = "heart" and X.Sex = "male"
then X.Age > 50 [1200,0.70]
If X.Diagnosis = "heart" and X.Sex = "male"
and X.City = Y and Y.Population > 500,000
then X.Age > 45 [500,0.60]
Query
Age City
55 NY
60 LA
40 Seattle
City Population
NY over 1 M
LA Over 1 M
Seattle 500,000

Figure 2. Data mining as rule querying

Another difficulty is that the border between querying and discovery is fuzzy; one might even say that there is no such a thing as real "discovery." Discovery is just a matter of the expressive power of a query language.[1] Thus, discovery is a very fuzzy term and is often misused to describe the results of standard SQL querying.

In the following section, we present the emerging body of work in which rules are considered primarily as KDD objects. This work presents a first step in realizing the more ambitious program just described.

Rule Querying and Rule Manipulation

As a running example, consider a database about occurrences of certain diseases. Suppose we have three classes of objects in the database: Patients (with attributes or operations Age, Sex, City, Diagnosis, Height, Weight, ClaimAmt (Claim Amount) and so forth, a computed attribute BMI (body mass index), and City (with attributes State, Population, etc.))

An example of a propositional rule in this database would be

X.Diagnosis = "heart disease" And X.Sex = "male"
==> X.Age > 50 [1200,0.70]

indicating there are at least 1,200 heart attack cases and that at least 70% of these occurred to persons over 50. Thus, propositional rules involve one (implicit) table or predicate and do not involve variables. Propositional rules can, in principle, use arbitrary methods instead of just materialized attributes.

Predicate rules are Horn clauses (implications) with possibly different predicates occurring in the body and in the consequent of a rule. Additionally, these rules involve statistical quantifiers that capture support and confidence of a rule. An example of such a rule could state that 80% of employees who are married to their managers are working on federally supported projects. Such a rule will involve multiple predicates such as "Married" and "Project."

Any database implicitly defines the collection of all propositional or predicate rules in it. While this collection is too large to enumerate, we use it in a conceptual way.

Generating rules that satisfy predefined predicates is viewed as querying. As an example, the user may

[1]A recent article in the *Wall Street Journal* referred to standard SQL querying of a large warehouse of financial information as database mining and the query about employees who earn more than their managers as "discovering" important class of employees.

state that he or she wants to see all rules about a patient with heart disease such that the consequent of the rule says something about the age of the patient, there are at least 1,000 cases which the rule body applies, and the confidence of the rule is at least 65%. In the M-SQL language of Imielinski and Virmani [10] this rule query would be represented as follows:

```
Select
 From Mine(T): R
 Where R.Consequent = {(Age =*)
  R.Support > 1000
  R.Confidence > 0.65
```

R renames Mine(T) and Mine(T) is an operator that takes a class T and generates all propositional rules about T. Obviously, Mine(T) is actually never enumerated.

Attributes of R such as Consequent, Body (not present here), Support, and Confidence refer to the Consequent and Body of the rule, and its support and confidence numbers. The condition (Age = *) indicates that any value of the attribute Age is allowed to occur in the consequent. The rule about patients with heart disease would belong to the answer of this query.

Thus, rule discovery is viewed as yet another, perhaps more expressive, type of querying. Rules that constitute answers to rule queries are no more (or less) "discovered" than tuples in the answers to SQL queries.

But rules are not necessarily the final product of the KDD applications. Thus, a proper API which embeds a rule query language in a more expressive, general purpose, host programming environment is necessary. This requires the ability to manipulate and iterate over collections of rules by making them first-class programming objects, similar to those described in [10]. A true KDDMS would have to be equipped with such an API.

As an example, consider an application that predicts values of unknown data elements (null values) in the database on the basis of the other "known" fields. This requires generation of rules that have the attribute to be predicted occurring as a rule consequent; application of such rules to the current database state, and the substitution of the the nulls with predicted data values. This can be written as a host language (say C or C++) program with embedded rule queries and possibly standard database queries as well [10].

Thus, rule querying can be embedded in a host programming language, or it can also be used for free-form, interactive querying. The answer to a rule query will prompt the user to launch another, perhaps incrementally different query or even generate a massive number of new queries against the database. Therefore the querying process generally consists of a sequence of queries, the next one related to, or built upon each preceding one. This requires additional features. For example, the user can take advantage of the previous query, by slightly modifying it or by taking a rule which is a part of the answer and mining around it. Notice, also, that queries themselves may require the system to perform complex search tasks that may lead to nondeterministic computations. Thus, queries serve only as general and often loose specifications for the system to provide varying degrees of user control over the discovery process. (In this way even a fully "automated" discovery can be viewed as query evaluation for a very loose and broadly specified query.)

For example, finding the rule about patients with heart disease, the user may want to mine around the rule to get a subpopulation of these patients that satisfy this rule in an even stronger sense.

Rule Query Languages

Several rule query languages have been proposed so far. In general, rule query languages include two categories: those targeted at propositional rules, such as the rules defined in the previous section, and query languages dealing with predicate rules. In this section we briefly review the main features.

In [9] a rule query language to query predicate rules with statistical quantifiers is described.

Rule queries allow different types of closure operators. (Not to be confused with the earlier defined closure principle, the result of the rule query can contain rules that look similar, but are different from the query template.) For example, a closure operation might allow additional conditions of the form (x.m = v) to be added to the rule. Other closure operators allow substituting the set of values for a given variable (* operator) and adding extra predicates to the rule body. This leads to rules that are more specific than the rule constituting the query pattern. Additionally, the user may specify the desirable support (which reflects the statistical significance of the rule) and confidence (which reflects the rule strength).

Shen, et al. [13] consider the logic programming language LDL++ as a metapattern querying language. The higher-order features of LDL++ allow for metalevel querying where, for instance, the predicates in the rules are only loosely specified (such as "Family Characteristic" metapredicate which stands for a collection of possible predicates dealing with family data). As an example of the metapattern notation, a pattern

P(X Y) And Q(Y Z) ==> R(X Z)

looks for predicate rules expressing transitive relationships between binary predicates P, Q, and R.

On the other hand, there is no support for statistical quantification expressing imprecise rules in terms of confidence and support. It is also not clear how features such as mining around a rule can be represented in LDL++-based mining, without actual modification of LDL++ itself. Regardless, LDL++ seems to be a good starting point for rule querying languages.

Inductive logic programming methods [5] also deal with goal-oriented rule discovery; another term for user-specified querying.

Finally, the M-SQL approach of Imielinski and Virmani [10] extends SQL with rule mining features to query propositional rules. Their work differs from the work here by not using the logic programming path but building upon a standard C++/SQL interface with possible persistent classes (such as those offered, for instance, by Object Store). They add one additional operator Mine to SQL, allowing rule generation from the database as well as retrieval from the rulebase, created to store some of the previously generated rules. Other query language proposals to query rules include [11]. These do not, however, seem to support the closure principle.

Finding Rules and Compiling Queries

Existing work on data mining already provides possibilities for realizing the goals mentioned in the previous sections. We now review some of the algorithms originally suggested for the discovery of "association rules" [1, 2]. We will give a classification of different methods and show how these algorithms can in fact be used as rule query evaluators.

There is a simple observation often used to make the running time of rule generation dependent on the number of rules produced. In order to generate a propositional rule of the form, `Body ==> Consequent`, we first measure support (number of tuples) satisfying the body of the rule—a conjunct of descriptors—then we measure support of the conjunct which is a conjunction of `Body` and `Consequent`. We divide the former by the latter to get the rule confidence.

In order to generate a massive number of rules that satisfy the rule query conditions, we first generate a number of frequent conjuncts along with their support. Rules are generated from such conjuncts using the rule formation step just mentioned.

The key aspect of all different algorithms based on so-called association mining is based on the observation that rule support can be used as an effective pruning criterion. Basically, if a given conjunct does not meet the minimum support constraints, there is no point in extending it any further by adding extra conditions since any resulting extension would not meet support.

There are two main types of algorithms for computing the collection of frequent conjuncts: bottom-up methods and indexing methods. In the bottom-up approach introduced in [1], the database is scanned several times. For each pass, a set of candidate conjunctions is generated, and then their support is evaluated. In the simplest approach, the first pass leads to 1-descriptor conjunctions, the second pass to 2-descriptor conjunctions, and so forth. Conjunctions that do not meet support requirements are discarded. At each level, one has to investigate only conjunctions known to have sufficient support for their subconjunctions. The process terminates when no more conjuncts meeting the query requirements can be generated.

In the indexing approach, first inverted lists (indexes) are formed for each individual attribute specified in the query. An inverted list is a linked list of all object identifiers for objects satisfying a given descriptor. The process then proceeds by merging the linked lists performing essentially index ANDing for larger and larger subsets of the original attribute set until no more support or attributes can be added.

As an example, consider again the sample rule query given. Finding the answer to this query requires evaluating all frequent conjuncts of the description language, and also discretizing the continuous attributes like Age. The pruning criterion on the frequency of the conjuncts can be used efficiently. Furthermore, analogously to the principle of "pushing selections inward" in the optimization of ordinary queries, one can compile the selection condition

R.consequent = (Age = *)

into the rule-finding method. Namely, one needs to find only frequent conjuncts such that a restriction on Age is included. This restriction can be embedded into the generation phase for new candidates. Similar optimizations can actually be done with a variety of conditions on the interesting rules.

Both the indexing and the bottom-up methods have been shown to work effectively on real data of millions of records [3, 12]. The time usage varies linearly with the size of the input data set, and also linearly with the size of the output. These methods are beginning to appear in data mining products. Their drawback is that if there is a long conjunction with sufficient support, all subconjunctions will necessarily be generated leading in the worst case to exponential complexity (in terms of the number of attributes). Agrawal, et al. [3] provide some comparisons of performance between the bottom-up and covering-set approaches.

These bottom-up and indexing methods were developed for finding simple association rules. One can use them for various other types of formalisms, too, including finding patterns from sequences of events.

Conclusion

It took database researchers more than 20 years to develop efficient query optimization and execution methods for relational query languages. In this article we have claimed that specification, compilation and execution of query languages for database mining is a challenge that requires a similar, long-term effort extending to the next decade and beyond.

In the same way business applications are currently supported using SQL-based API, the KKD applications need to be provided with application development support. In addition to the ability to query KDD objects, data mining query languages need to provide better support for such useful data mining operations as finding nearest neighbor (neighbors), clustering, or discretization and aggregate operations on data objects. Developments in multidimensional databases such as Essbase from Arbor Software are beginning to meet these goals. The recent introduction of the data cube basic operation [7] is an interesting suggestion in this direction.

We have argued that development of querying tools for data mining is one of the big challenges to the database community. This work requires interdisciplinary synergy between databases, machine learning, statistics, pattern recognition as well as high-performance computing. We have shown some of the preliminary work on rule query languages that does not yet integrate KDD objects other than rules. There is a definite need for complete KDD query languages in which classification trees, neural nets, clustering tools, and so forth, can fully interoperate with existing query languages as well as among themselves.

Although KDD objects are basic objects of data mining applications, they are hardly a final product, just as tuples or records are typically not a final product of database applications. Typically, KDD objects generated by queries will be embedded in more complex applications that may involve spreadsheets, further statistical analysis packages, graphical tools, and so forth. Therefore, it is very important to provide application developers with tools to build applications using a KDD query language as only one of the building blocks. Second-generation database mining systems should achieve these goals.

References

1. Agrawal, R. Imielinski, T., and Swami, A. Mining association rules between sets of items in large databases. In *Proceedings of ACM Sigmod '93*, May 1993, pp. 207–216.
2. Agrawal, R., Imielinski, T., and Swami, A. Database mining: A performance perspective. *J. IEEE: Special Issue on Learning and Discovery in Knowledge-Based Databases, 5*, 6 (Dec. 1993), 914–925.
3. Agrawal R., Mannila H., Srikant, R., Toivonen, H., and Verkamo, A. I. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, Eds., AAAI, 1996, 307–328.
4. Agrawal, R. and Shim, K. Developing tightly-coupled data mining applications on a relational database system. In *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining* (Montreal, Canada), Aug. 1996, pp. 287–290.
5. De Raedt, L. and Bruynooghe, M. A theory of clausal discovery. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence* (IJCAI–93), Morgan and Kaufmann, 1993, pp. 1058–1053.
6. Fayyad, U. M., Smyth, P., Weir, N., and Djorgovski, S. Automated analysis and exploration of large image databases: Results, progress, and challenges. *J. Intell. Info. Syst. 4*, (1995), 7-25.
7. Gray, J., Bosworth, A., Layman, A., and Pirahesh, H. Data {C}ube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Proceedings of the 12th International Conference on Data Engineering, ICDE'96* (New Orleans), Feb. 1996, pp. 152–159.
8. Hatonen, K., Klemettinen M., Mannila H., Ronkainen P., and Toivonen, H., Knowledge discovery from telecommunication network alarm databases. In *Proceedings of the 12th International Conference on Data Engineering ICDE'96* (New Orleans), Feb. 1996, pp. 115–122.
9. Imielinski, T. Hirsh, H. Query-based approach to database mining. Technical report, Rutgers University, Dec. 1993.
10. Imielinski, T., Virmani, A., and Abdulghani, A. Disovery board application programming interface and query language for database mining. In *Proceedings of KDD96* (Portland, Ore.), Aug. 1996, pp. 20–26.
11. Meo, R. Psaila, G. and Ceri, S. A new SQL-like operator for mining association rules. In *Proceedings of VLDB96*. To be published.
12. Savasere, A., Omiecinski, E., and Navathe, S. An efficient algorithm for mining association rules in large databases. In *Proceedings of VLDB95* (Sept. 1995), pp. 432–444.
13. Shen, W. M., Ong, K., Mitbander, B. and Zaniolo, C. Metaqueries for Data Mining. In *Advances in Knowledge Discovery and Data Mining*, U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, Eds., AAAI, 1996, pp. 375–398.

TOMASZ IMIELINSKI (imielins@cs.rutgers.edu) is a professor and chair of the Department of Computer Science, Rutgers University, New Brunswick, N.J.

HEIKKI MANNILA (mannila@cs.Helsinki.FI) is a professor of computer science at the University of Helsinki, Finland.