From: KDD-95 Proceedings. Copyright © 1995, AAAI (www.aaai.org). All rights reserved.

Active Data Mining

Rakesh Agrawal *Giuseppe Psaila**

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120
ragrawal@almaden.ibm.com *psaila@elet.polimi.it*

Abstract

We introduce an active data mining paradigm that combines the recent work in data mining with the rich literature on active database systems. In this paradigm, data is continuously mined at a desired frequency. As rules are discovered, they are added to a rulebase, and if they already exist, the history of the statistical parameters associated with the rules is updated. When the history starts exhibiting certain trends, specified as shape queries in the user-specified triggers, the triggers are fired and appropriate actions are initiated.

To be able to specify shape queries, we describe the constructs for defining shapes, and discuss how the shape predicates are used in a query construct to retrieve rules whose histories exhibit the desired trends. We describe how this query capability is integrated into a trigger system to realize an active mining system. The system presented here has been validated using two sets of customer data.

Index Terms. Active Data Mining, Shape Queries, Blurry Queries, Triggers, Time-Series Data.

Introduction

Data mining (Stonebraker *et al.* 1993) (also called knowledge discovery in databases (Piatetsky-Shapiro & Frawley 1991)) is the efficient discovery of previously unknown patterns in large databases, and is emerging as a major application area for databases (Gartner Group 1994) (Business Week 1994). In (Agrawal, Imielinski, & Swami 1993), three classes of data mining problems involving associations, sequences, and classification were introduced and it was argued that these problems can be uniformly viewed as requiring discovery of rules embedded in massive data. Attached to every discovered rule are some statistical parameters, such as confidence or support of the rule.

As the data mining technology is applied in the production mode (Stores 1994), the need for *active* mining arises. Figure 1 shows a schematic of the active data mining process. The basic idea is as follows. Rather than applying a mining algorithm to the whole data, the data is first partitioned according to time periods. The granularity of the time period is application-dependent. The amount of data available is large (generally in gigabytes and more) so that this partitioning does not lose significance of the rules discovered. The mining algorithm is now applied to each of the partitioned data set and rules are obtained for each time period. These rules are collected into a rulebase. In this rulebase, each statistical parameter of a rule will have a sequence of values, called the history of the parameter for that rule. We can now query the rulebase using predicates that select rules based on the shape of the history of some or all parameters.

The user can specify triggers (Dayal, Hanson, & Widom 1994) over the rulebase in which the triggering condition is a query on the shape of the history. As the fresh data comes in for the current time period, the mining algorithm is run over this data, and the rulebase is updated with the generated rules. This update causes the histories of the rules to be extended. (A history of a new rule is initialized with zero values for the past time periods.) This, in turn, may cause the triggering condition to be satisfied for some rules and the corresponding actions to be executed.

Such active systems can be used, for instance, to build early warning systems for spotting trends in the retail industry. For example, if we were mining association rules (Agrawal, Imielinski, & Swami 1993), we will have histories for the support and confidence of each rule [1]. Following the promotion

*Current address: Politecnico di Torino, Dip. Automatica e Informatica, C.so Duca degli Abruzzi 24, I-10129 TORINO, Italy.

[1] An association rule (Agrawal, Imielinski, & Swami 1993) is an expression of the form $A \Rightarrow C$, where both A and C are sets of literals. In a database of transac-

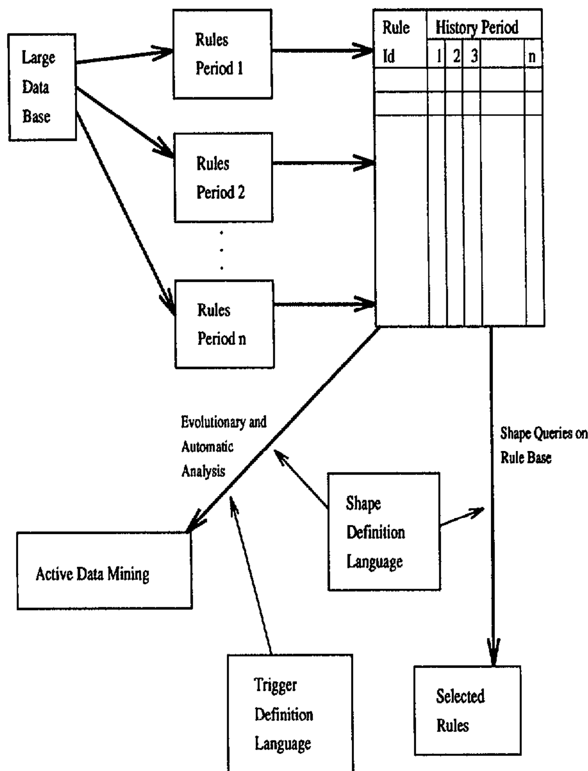

Figure 1: Active Data Mining Process

for an item X, the user may specify a notification trigger on the rule $X \Rightarrow Y$; the triggering condition being that the support history remains stable, but the confidence history takes the shape of a downward ramp. Firing of this trigger will signify that if the goal of promoting X was to drag the sale of Y, it was not fulfilled. The loyalists continued to buy X and Y together, but the new buyers cherry-picked X.

The rest of the paper is organized as follows. In Section Shape Definition, we give the constructs for defining shapes. In Section Queries, we present the query language that uses the shape predicates to retrieve rules whose histories exhibit the desired patterns. This query capability is integrated into a trigger system. Section Triggers presents the trigger definition facility, describes what we call the wave execution semantics of trigger execution, and discusses how it is used in the mining process. We conclude in Section Implementation and Experience with a discussion of our prototype implementation and experiences from using this system on two sets of customer data. We also give our directions for future work.

tions, where each transaction is a set of literals, the rule $A \Rightarrow C$ signifies that very often when A appears in a transaction, so does C. How often this happens is captured by the "confidence" parameter, and is indicative of the strength of the rule. The "support" parameter gives the fraction of transactions in the database in which the given rule is present, and is indicative of the prevalence of the rule.

Shape Definitions

A shape found in a history can be described by considering the transition of values assumed by the shape at the beginning and end of each unit time period. We let the user define classes of transitions and assign symbols to them. These symbols are called **elementary shapes**. We do not have a pre-canned set of elementary shapes; the user can add or delete shapes or change the definition of any of them. However, to keep the discussion concrete, assume a user-defined set of elementary shapes, consisting of **up**, **Up**, **down**, **Down**, **appears**, **disappears**, **stable**, and **zero**. The shape **up** could be a slightly increasing transition with a minimum and maximum variation of 0.05 and 0.19 respectively; **Up** could be a highly increasing transition with a minimum and maximum variation of 0.20 and 1.00 respectively; **appears** could be a transition from a zero value to a non-zero value; **stable** could be a transition in which the absolute difference between the initial and final value is not more than 0.04; etc.

Complex shapes can be derived by recursively combining elementary shapes and previously defined derived shapes, using the **shape operators**. We have presented these operators in (Agrawal *et al.* 1995) and given their formal semantics, expressive power, and implementation techniques. These operators are summarized in Table 1. We give syntax for each operator and how the corresponding derived shape is matched in a history H.

Using these operators, one can describe a wide variety of shapes found in a history, including "blurry" shapes where the user cares about the overall shape but does not care about specific details. The syntax for defining a shape is:

(shape *name(parameters) descriptor*).

For example, here is a definition of a **doublepeak**:

```
(shape spike(upcnt dncnt)
    (concat (atleast upcnt (any up Up))
            (atleast dncnt
                     (any down Down))))
(shape doublepeak(width ht1 ht2)
    (in width (inorder spike(ht1 ht1)
              spike(ht2 ht2))))
```

We first define **spike** to be a shape that has at least **upcnt** number of either **up** or **Up** transitions followed by at least **dncnt** number of either **down** or **Down** transitions. Then **doublepeak** is a shape **width** wide that has two non-overlapping spikes.

Note that `width` may be wider than the sum of the width of the two spikes and there may be noise on either side of them. As another example, the shape `bullish`:

```
(shape bullish(width upcnt dncnt)
    (in width
        (and
          (noless upcnt (any up Up))
          (nomore dncnt
                  (any down Down)))))
```

is defined to have at least `upcnt` ups (either `up` or `Up`) and at most `dncnt` downs (either `down` or `Down`) in `width` time periods. Finally, the shape `drift`:

```
(shape drift(width)
    (in width (precisely 0
                (any up Up down Down))))
```

has no ups or downs in `width` time periods.

Earlier languages based on regular expressions for finding patterns in sequences (see (Seshadri, Livny, & Ramakrishnan 1995) for a review of some of them) were not targetted at defining shapes. The difference in design focus influences which expressions are easy to write, understand, optimize, and evaluate. We encourage the reader to write the shape definitions given above as regular expressions to appreciate the difference.

Queries

With the machinery for defining shapes in hand, we are ready to specify how we can retrieve rules whose one or more histories contain the desired shapes. The syntax for defining a query is:

(query (*shape history-spec*))

Here, *shape* is the descriptor for the shape to be matched. The *history-spec* is of the form:

history-name start-time end-time

Here *history-name* specifies the name of the history in which the shape should be matched. The portion in which the matching occurs is constrained by the interval specified by *start-time* and *end-time*. Matching over the complete history can be specified by using the keywords `start` and `end` for *start-time* and *end-time* respectively.

The result of the execution of a query is the set of all rules that contain the desired shape in the specified history. In addition, the result also contains the list of subsequences of the history that matched the shape. If no subsequence matches the specified shape, the result is an empty set.

Here is an example of a query:

```
(shape ramp() (concat Up Up))
(query ((ramp) (confidence start end)))
```

Multiple Choice	(any P_1 P_2 ... P_n)
Match all subsequences of H that match at least one of the P_i shapes.	
Concatenation	(concat P_1 P_2 ... P_n)
First, match the shape P_1. If a matching subsequence s is found, match P_2 in the subsequence of H immediately following the last element of s. Accept the match if it is strictly contiguous to s, etc.	
Multiple Occurrences	(exact n P) (atleast n P) (atmost n P)
Match all subsequences of H that contain exactly (at least/at most) n contiguous occurrences of the shape P. In addition, the resulting subsequences must neither be preceded nor followed by a subsequence that matches P.	
Bounded Occurrences	(in *length* *shape-occurrences*)
Do "blurry" matching. Here *length* specifies the length of the shape in number of transitions. The *shape-occurrences* has two forms given below:	
shape-occurrences : logical combination using and and or.	(precisely n P) (noless n Q) (nomore n R)
Match all *length* long subsequences of H that contain precisely (no less than/ no more than) n occurrences of the shape P (Q/R). The n occurrences of P (Q/R) need not be contiguous in the matched subsequence; there may be overlap or arbitrary gap between any two.	
shape-occurrences : ordered shapes.	(inorder P_1 $P_2 \ldots P_n$)
Match all *length* long subsequences of H containing the shapes P_1 through P_n in that order. P_i and P_{i+1} may not overlap, but may have an arbitrary gap.	

Table 1: Shape Operators

We have defined a simple shape `ramp`, consisting of two consecutive `Up`s, and we want to retrieve all the rules whose confidence history contains a `ramp`.

Instead of the shape name, we could have alternatively written its definition in the above query. We also could have limited the range of confidence history in which the shape should be matched. Here is a modified query:

```
(query ((Concat Up Up)
        (confidence start 10)))
```

The user can also retrieve combinations of several shapes in different histories by using the logical operators and and or. Here is an example of a query that is looking for different shapes in the two histories of a rule — an `upramp` in support but a `dnramp` in confidence:

```
(shape upramp(len cnt)
    (in len (noless cnt (any up Up))))
(shape dnramp(len cnt)
```

```
    (in len
        (noless cnt (any down Down))))

(query
  (and
    (upramp(5 3) (support start 10))
    (dnramp(5 3) (confidence start 10))
  ))
```

Triggers

The query language we just described provides the capability to discover interesting information by analyzing rules and their histories in novel ways. Consider a user who is periodically collecting rules in the rulebase and wants to discover rules that are assuming critical (or interesting in some other way) behavior. For instance, the user may be interested in rules that have started exhibiting increasing trend. Rather than running queries every time the data for a new period comes and rules are added to the rulebase, it will be preferable to post these queries as triggers and let the system initiate appropriate actions (e.g. notification) when the trigger conditions are satisfied.

We use the ECA (Event Condition Action) model(Chakravarthy *et al.* 1989) as the basis for our trigger system. We lean on the rich literature in active databases (see (Dayal, Hanson, & Widom 1994) for an overview) and specialize it for our purpose. The interesting aspects of our trigger system are what can be specified as trigger conditions, the semantics of the trigger execution, and how it is used in the active mining process.

The syntax for specifying a trigger is:

```
(trigger trigger-name
    (events events-spec)
    (condition (shape history-spec))
    (actions actions-spec))
)
```

A trigger definition has three sections: *events*, *condition* and *actions*. Let us examine each of them.

The trigger system reacts to pre-defined and user-defined events. The pre-defined events describe an external update of the rulebase. These events are: `createrule` and `updatehistory`. They occur when a new rule is added to the rulebase and the history of rule is updated, respectively. A user-defined event is introduced to the system as:

```
(event event-name)
```

where *event-name* is the name of the event.

The *events-spec* in the events section specifies the events to which the trigger being defined reacts. Pre-defined and user-defined events and their logical combinations using the logical operators `or` and and can appear in the *events-spec*[2]. A trigger is considered *fired* if the event specification is true for at least one rule in the rulebase. That is, the specified event combination has occurred for some rule.

The condition section is syntactically and semantically similar to the `query` construct discussed in Section Queries. The difference is that the condition is evaluated only on rules present in the *affected set* (Widom & Finkelstein 1990) produced by the events section, instead of the whole rulebase. A condition selects the affected set of rules from the rulebase and performs the specified shape query on the relevant histories of those rules. The condition is true if the output set resulting from the query is not empty. In that case, the actions section is executed on the query output.

The *action-specs* in the actions section is a list of actions that are executed for all (and only those) rules that belong to the output set produced by the condition evaluation. An action can be an execution of a function, such as *notify* or *show*, which can be defined by the user or system supplied. An action can also be a user-defined event name, in which case an occurrence of the specified event is generated. An action does not change the state of the rulebase; the goal of model is only to notify that the properties expressed by the condition of a trigger holds for some rules and is accomplished by generating pre-defined and user-defined events that alert any possibly interested trigger.

Wave Execution Semantics Several semantics have been proposed for trigger systems in active databases (see (Simon, Kiernan, & de Maindreville 1992) for a discussion). The execution semantics for our trigger system follows what we call the *wave execution model.* This semantics is close to what is known as the deterministic semantics for Datalog-like rules (S.Ceri, G.Gottlob, & L.Tanca 1990). The attractiveness of this semantics is its simplicity and a good match for our application.

A *wave* is a set of event occurrences that come together to the active system. The trigger execution process starts when a new wave is ready. First, the event specification of every trigger is checked to determine if it will fire. Triggers for which this evaluation is true are selected for firing and their affected set (rules affected by *events-spec*) is produced. When the event specification has been

[2]We considered richer event specifications, such as those in (Chakravarthy *et al.* 1994) (Gatziu & Dittrich 1994) (Gehani, Jagadish, & Shmueli 1992), but decided against it. Our trigger system is meant to react to changes in the shapes of the history, for which we have a rich shape-specification language. We keep our events specification simple but use rich specifications in the condition section.

checked for all the triggers, the current wave has been used up and any event generated as a consequence of the triggers fired will belong to a new wave.

The selected triggers are fired now. For each fired trigger, its condition is evaluated only on rules in the affected set. If the condition is true, the output set is passed to the actions section. The actions section is immediately executed for each rule in the output set. If any event is generated as a consequence, it is added to the new wave.

After the conclusion of the condition evaluation and the eventual actions execution for all the fired triggers, the process is repeated considering the events generated as belonging to the new wave. The process terminates if the evaluation of all the event specifications determines that no trigger needs to be fired.

Example We now give a simple example to illustrate our trigger facility. Suppose that a user wants to be notified if the support for a rule is increasing but its confidence is decreasing at the same time. The following definitions show how the user can accomplish this goal:

```
(shape uptrend(width upcnt)
    (in width
        (noless upcnt (any up Up))))
(shape dntrend(width dncnt)
    (in width (noless
                dncnt (any down Down))))

(event upward)

(trigger detect_up
   (events updatehistory)
   (condition
    (uptrend(5 4)
     (support (- end 5) end)))
   (actions upward)
)
(trigger detect_dn
   (events upward)
   (condition
    (dntrend(5 4)
     (confidence (- end 5) end)))
   (actions notify)
)
```

We first specify what is meant by support is increasing and confidence is decreasing by defining two shapes: `uptrend` and `dntrend`. We introduce an event named `upward` to the system using the `event` construct. We then define the trigger `detect_up`. This trigger can be fired by the pre-defined event `updatehistory`. If this trigger is fired, the condition section of this event checks if the rules that were updated (affected set of the event `updatehistory`) contain `uptrend` in the last five periods of their support history. If this condition is evaluated as true for some rules, the user-defined event `upward` is generated for each of these rules.

The second trigger `detect_dn` reacts to the generation of the occurrences of the `upward` event and it checks for `dntrend` in the last five periods of the confidence history of only those rules for which the trigger has been fired (affected set of the `upward` event). Thus, the user is notified of only those rules that simultaneously had an `uptrend` in support and `dntrend` in confidence in the last five time periods.

Implementation and Experience

A prototype of the active data mining system described here has been implemented on the AIX system as part of the Quest project at IBM. The implementation uses an object-oriented design. A base class, called *LangObj*, declares methods that define the common interface for the derived classes; the most interesting method being *Evaluate* which is invoked by the query management subsystem to execute queries. The shape definition language objects—elementary shapes, shape operators—are represented as classes derived from the base class (e.g. *LangObj_concat* corresponding to the operator `concat`). The associated *Evaluate* method implements the matching strategy of the corresponding operator. To create a query object, the constructor is called with actual parameters pointing to objects corresponding to shape specifications. Thus, a tree of objects is created, with each object corresponding to an elementary or a derived shape. To execute this query, the query management subsystem calls the *Evaluate* method of the root of the tree object, providing the time sequence to be analyzed as the argument. Any non-leaf node of the tree recursively invokes the *Evaluate* method of the branches, until a leaf node is reached where the matching takes place. A similar approach has been used to implement triggers as well.

The prototype system was successfully tested against two large datasets. The first dataset from a mail-order company consisted of roughly 2.9 million transactions collected over five years. The second dataset from a market research company that provides marketing information to the retail industry consisted of three years of roughly 6.8 million point-of-sales transactions. In both the cases, we divided data on monthly basis and mined association rules (Agrawal, Imielinski, & Swami 1993) for each dataset. For rules discovered, we saved three parameter histories for each rule in the rulebase: support, confidence, and the product of sup-

port and confidence values. We specified shape triggers on rules corresponding to earlier parts of data and examined their firings with the addition of rules for the later periods. This experience leads us to believe that the proposed active data mining paradigm is very attractive for the production deployment of data mining technology.

Our current prototype is a stand-alone implementation. In future, we plan to integrate our query constructs and trigger functions with a SQL relational database system. We currently recompute the trigger conditions over the whole history to determine trigger firing. In a production system, it will be useful to have the facility for materializing the partial results of the current trigger queries and incrementally completing them as the histories are extended. We plan to investigate incremental computations in future.

References

Agrawal, R.; Psaila, G.; Wimmers, E. L.; and Zait, M. 1995. Querying shapes of histories. In *Proc. of the VLDB Conference.*

Agrawal, R.; Imielinski, T.; and Swami, A. 1993. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering* 5(6):914–925. Special Issue on Learning and Discovery in Knowledge-Based Databases.

Business Week. 1994. Database marketing.

Chakravarthy, S.; Blaustein, B.; Buchmann, A.; Carey, M.; Dayal, U.; Goldhirsch, D.; Hsu, M.; Jauhari, R.; Ladin, R.; Livny, M.; McCarthy, D.; McKee, R.; and Rosenthal, A. 1989. HiPAC: A research project in active time-constrained database management – final technical report. Technical Report XAIT-89-02, Xerox Advanced Information Technology.

Chakravarthy, S.; Krishnaprasad, V.; Anwar, E.; and Kim, S.-K. 1994. Composite events for active databases: Semantics, contexts, and detection. In *Proc. of the VLDB Conference*, 606–617.

Dayal, U.; Hanson, E.; and Widom, J. 1994. Active database systems. In Kim, W., ed., *Modern Database Systems: The Object Model, Interoperability and Beyond.* New York: ACM Press.

Gartner Group 1994. Data mining: The next generation of business intelligence? ATG Research Note T-517-246, Gartner Group Inc., Stamform, CT.

Gatziu, S., and Dittrich, K. 1994. Detecting composite events in active databases using petri nets. In *Proc. of the 4th Int'l Workshop on Research Issues in Data Engineering: Active Database Systems*, 2–9.

Gehani, N.; Jagadish, H.; and Shmueli, O. 1992. Composite event specification in an active databases: Model & implementation. In *Proc. of the VLDB Conference*, 327–338.

Piatetsky-Shapiro, G., and Frawley, W., eds. 1991. *Knowledge Discovery in Databases.* Menlo Park, CA: AAAI/MIT Press.

S.Ceri; G.Gottlob; and L.Tanca. 1990. *Logic Programming and Databases.* Springer Verlag.

Seshadri, P.; Livny, M.; and Ramakrishnan, R. 1995. SEQ: A model for sequence databases. In *Proc. of the IEEE Int'l Conference on Data Engineering.*

Simon, E.; Kiernan, J.; and de Maindreville, C. 1992. Implementing high level active rules on top of a relational DBMS. In *Proc. of the VLDB Conference*, 315–327.

Stonebraker, M.; Agrawal, R.; Dayal, U.; Neuhold, E. J.; and Reuter, A. 1993. The DBMS research at crossroads. In *Proc. of the VLDB Conference*, 688–692.

Stores. 1994. Quest: IBM develops market basket analysis system.

Widom, J., and Finkelstein, S. 1990. Set-oriented production rules in relational database systems. In *Proc. of the ACM SIGMOD Conference on Management of Data*, 259–270.