Mining Optimized Association Rules for Numeric Attributes

Takeshi Fukuda, Yasuhiko Morimoto, Shinichi Morishita, and Takeshi Tokuyama
IBM Tokyo Research Laboratory
1623-14, Shimo-tsuruma, Yamato City, Kanagawa Pref, 242, JAPAN
{fukudat,morimoto,morisita,ttoku}@trl.ibm.co.jp

Abstract

Given a huge database, we address the problem of finding association rules for numeric attributes, such as

$$(Balance \in I) \Rightarrow (CardLoan = yes),$$

which implies that bank customers whose balances fall in a range I are likely to use card loan with a probability greater than p. The above rule is interesting only if the range I has some special feature with respect to the interrelation between *Balance* and *CardLoan*. It is required that the number of customers whose balances are contained in I (called the *support* of I) is sufficient, and also that the probability p (called the *confidence ratio*) should be much higher than the average probability of the condition being met.

Our goal is to realize a system that finds such appropriate ranges automatically. We mainly focus on computing two *optimized ranges*: one that maximizes the support on the condition that the confidence ratio is at least a given threshold value, and another that maximizes the confidence ratio on the condition that the support is at least a given threshold number.

Using techniques from computational geometry, we present novel algorithms that compute the optimized ranges in linear time if the data are sorted. Since sorting data with respect to each numeric attribute is expensive in the case of huge databases that occupy much more space than the main memory, we instead apply randomized bucketting as the preprocessing method, and thus obtain an efficient rule-finding system.

Tests show that our implementation is fast not only in theory but also in practice. The efficiency of our algorithm enables us to compute optimized rules for all combinations of hundreds of numeric and Boolean attributes in a reasonable time.

PODS '96, Montreal Quebec Canada
© 1996 ACM 0-89791-781-2/96/06..$3.50

1 Introduction

Recent progress in technologies for data input through such media as bar-coded labels, credit cards, OCRs, and cash dispensers, have made it easier for finance and retail organizations to collect massive amounts of data and to store them on disk at a low cost. Such organizations are interested in extracting from these huge databases unknown information that inspires new marketing strategies. Current database systems are their primary means of realizing this aim, but in database and AI communities, there has been a growing interest in efficient discovery of interesting rules, which is beyond the power of current database functions [AGI+92, AIS93a, AIS93b, AS94, BFOS84, HCC92, NH94, PCY95, PS91, PSF91, Qui93, SAD+93].

Association Rules

Given a database universal relation, we consider the association rule that if a tuple meets a condition C_1, then it also satisfies another condition C_2 with a probability (called *confidence* in this paper). We will denote such an association rule (or rule, for short) between the presumptive condition C_1 and the objective condition C_2 by $C_1 \Rightarrow C_2$ [1].

We call a rule *exact* if its confidence is 100%. For the purpose of discovering exact or almost exact rules, Piatetsky-Shapiro [PS91] presents the KID3 algorithm. Important rules in scientific databases are likely to be exact. On the other hand, business database, such as a customer database or a transaction database, tends to reflect the uncontrolled real world, and the confidence of an interesting rule is usually much less than 100%.

Thus when we handle commercial databases, we should consider a broader class of rules whose confidences are greater than a specified minimum threshold, such as 30%. We call such rules *confident*. Agrawal, Imielinski, and Swami [AIS93b] study ways of discovering all confident rules. They focus on rules with conditions that are conjunctions of $(A = yes)$, where A is

[1] We use the symbol "$\Rightarrow$" in order to distinguish the relationship from logical implication, which is usually denoted by "$\rightarrow$".

a Boolean attribute, and present an efficient algorithm. They have applied the algorithm to basket-data-type retail transactions to derive interesting associations between items, such as

$$(Pizza = yes) \wedge (Coke = yes) \Rightarrow (Potato = yes).$$

Improved versions of the algorithm have also been reported [AS94, PCY95].

Optimized Association Rules

In addition to Boolean attributes, databases in the real world usually have numeric attributes, such as age and the balance of account in a database of bank customers. Thus, it is also an important issue to find association rules for numeric attributes. In this paper, we focus on finding a simple rule of the form

$$(Balance \in [v_1, v_2]) \Rightarrow (CardLoan = yes),$$

which expresses that customers whose balances fall in the range between v_1 and v_2 are likely to use card loan. If an instance of the range is given, the confidence of this rule can be computed with ease. In practice, however, we want to find a range that yields a confident rule. Such a range is called a *confident* range. Unfortunately, a confident range is not always unique, and for instance, we could find a confident range that contains a very small number of customers.

Let the *support* of a range be the ratio of the number of tuples in the range to the number of all tuples. A range is called *ample* if its support is no less than a given fixed threshold. We want to find a rule associated with a range that is both ample and confident.

In particular, we would like to find the confident range with maximum support, and we call the associated rule an *optimized support* rule. This range captures the largest cluster of customers that are likely to use card loan with a probability no less than the given minimum confidence threshold. Here, we refer to a data set associated with a range of the numeric attribute value as a *cluster*, for short.

Instead of the optimized support rule, it is also interesting to find the ample range that maximizes the confidence. We call the associated rule an *optimized confidence* rule. This range gives us a cluster of more than, for instance, 10% of customers that tend to use card loan with the highest confidence factor. In the case of promoting card loan by sending direct mails to a fixed number of new customers within a limited budget, this rule gives us a nice information on the target of customers.

Main Results

There are trivial ways of computing optimized support rules and optimized confidence rules in $O(N^2)$, where N is the number of all tuples. In this paper, we give a non-trivial linear time algorithm for each optimized rule, on the assumption that the data are sorted with respect to the numeric attribute. Each algorithm uses some techniques developed in computational geometry to achieve linear time complexity.

Given the sorted data, our algorithms are asymptotically optimal. Sorting the database, however, could create a serious problem if the database is much larger than the main memory, because sorting data for each numeric attribute would take an enormous amount of time.

To handle giant databases that cannot fit in the main memory, we need to find another way of computing optimized rules. For this purpose, we present algorithms for approximating optimized rules, using randomized algorithms [MR95]. The essence of these algorithms is that we generate thousands of almost equi-depth buckets[2], and then combine some of those buckets to create approximately optimized ranges. In order to obtain such almost equi-depth buckets, we first create a sample of data that fits into the main memory, thus ensuring the efficiency with which the sample is sorted. Then, we sort the sample and divide the sample into equi-depth buckets. We show that those buckets in the sample also give almost equi-depth buckets in the original data with high probability.

Tests show that our implementation is fast not only in theory but also in practice. Even for a small case, our algorithm is faster than a naive quadratic-time algorithm by an order of magnitude. The efficiency of our algorithm allows to compute a complete set of optimized rules for all combinations of hundreds of numeric and Boolean attributes in a reasonable time. We present some performance results.

We have been focusing on rather simple rules of the form $(A \in [v_1, v_2]) \Rightarrow C$, but our algorithms can be straightforwardly extended to generate rules of the form $(A \in [v_1, v_2]) \wedge C_1 \Rightarrow C_2$, where C_1 and C_2 are Boolean statements that do not contain any uninstantiated ranges on numeric attributes.

It would also be valuable to extend our framework to rules with two numeric attributes in the presumptive condition, and to find the region in the two-dimensional space of these attributes representing a nice association rule between these two numeric attributes and the conclusion. For instance, we would like to find a rule such as

$$(Age, Balance) \in X \Rightarrow (CardLoan = yes),$$

where X is a rectangle or a connected region in two-dimensional space of Age and $Balance$. Optimized rules can also be naturally defined in this extension. An approach is given in our companion paper [FMMT96].

[2] Buckets are called (almost) *equi-depth* if tuples are (almost) uniformly distributed into buckets

Related Works

Some other works also handle numeric attributes and try to derive rules. Piatetsky-Shapiro [PS91] studies how to sort the values of a numeric attribute, divide the sorted values into approximately equi-depth ranges, and use only those fixed ranges to derive *exact* rules. Other ranges except for the fixed ones are not considered in his framework. Our method is not only capable of outputting optimized ranges but is also more handy than Piatetsky-Shapiro's method, since we need not make candidate ranges beforehand. Recently Srikant and Agrawal [SA96] has improved Piatetsky-Shapiro's method by considering the combination of some consecutive ranges. The combined range could be the whole range the numeric attribute, which just produces a trivial rule. To avoid this, Srikant and Agrawal present an efficient way of computing a combined range whose size is at most a threshold given by the user.

Some techniques, related but not directly applicable to finding optimized association rules, have been developed for handling numeric attributes in the context of deriving decision trees that are used for classifying data into distinct groups. ID3 [Qui93], CART [BFOS84], *CDP* [AIS93b], and SLIQ [MAR96] perform binary partitioning of numeric attributes repeatedly until each range contains data of one specific group (or some groups, sometimes) with high probability, while *IC* [AGI+92] uses k decomposition. Since both partitionings tend to yield large decision trees, in order to reduce the size of the trees, methods such as pruning some branches and linking some ranges together have also been proposed.

2 Preliminaries

2.1 Association Rules

Let R be a relation. In order to describe conditions on tuples in R, we use some primitive conditions, and describe more complicated conditions by using them. For a Boolean attribute A, $A = yes$ and $A = no$ are primitive conditions. For a numeric attribute A, $A = v$ and $A \in [v_1, v_2]$ are examples of primitive conditions.

Example 2.1 Consider a relation for retail transactions. Each attribute of the relation is a Boolean one whose domain is $\{yes, no\}$, and represents an item, such as *Coke* or *Pizza*. $(Coke = yes) \wedge (Pizza = yes)$ is a condition, and a tuple that meets the condition represents a customer who purchased a coke and a pizza. ∎

Example 2.2 Consider a relation for data on a bank's customers. Suppose that each tuple contains the balance of account and services (card loan or automatic withdrawal, say) for one customer.

$$(Balance \in [15821, 26264]) \vee (CardLoan = yes)$$

is an example of a condition. ∎

The *support* of condition C is defined as the percentage of tuples that meet condition C, and is denoted by $support(C)$. For instance, in the retail relation given in Example 2.1, if $support(Coke = yes) = 10\%$, 10% of customers purchase a coke.

Let C_1 and C_2 be conditions on tuples. An *association rule* (or *rule* for short) has the form $C_1 \Rightarrow C_2$. The *confidence* of rule $C_1 \Rightarrow C_2$ is defined as $support(C_1 \wedge C_2)/support(C_1)$, which we will denote by $conf(C_1 \Rightarrow C_2)$. For instance, in the bank relation in Example 2.2, suppose that the confidence of the rule

$$(Balance \in [15821, 26264]) \Rightarrow (CardLoan = yes)$$

is 50%; then 50% of customers whose balances fall in the range use credit card loans.

2.2 Optimized Association Rules

Throughout this paper, we focus on mining association rules of the form $(A \in [v_1, v_2]) \Rightarrow C$. Suppose that A and C are fixed. A rule is *confident* if its confidence is not less than the given minimum confidence threshold. Among confident rules, an *optimized support rule* maximizes $support(A \in [v_1, v_2])$. A rule is *ample* if $support(A \in [v_1, v_2])$ is not less than the given minimum support threshold. Among ample rules an *optimized confidence rule* maximizes the confidence.

Example 2.3 Consider rules of the form:

$$(Balance \in [v_1, v_2]) \Rightarrow (CardLoan = yes)$$

Suppose that 50% is given as the minimum confidence threshold. We may have many instances of ranges that yield confident rules, such as:

Range	[1000, 10000]	[5000, 5500]	[500, 7000]
Support	20%	2%	15%
Confidence	50%	55%	52%

Among those ranges, [1000, 10000] is a candidate range for an optimized support rule. Next, given 10% as the minimum support threshold, we may also have many ample rules, such as:

Range	[1000, 5000]	[2000, 4000]	[3000, 8000]
Support	13%	10%	11%
Confidence	65%	50%	52%

The reader might feel it strange that although [1000, 5000] is a superset of [2000, 4000], the confidence of the rule of the former range is greater than that of the latter range, but observation will confirm that this situation could really occur. ∎

2.3 Buckets

Let t be a tuple of the given relation R, and let $t[A]$ denote the value of the attribute A of t. *Buckets* of the domain of A are a sequence of disjoint ranges

$$B_1, B_2, \ldots, B_M$$
$$(B_i = [x_i, y_i] \text{ and } x_i \leq y_i < x_{i+1})$$

such that A's values for all tuples are covered by the buckets; namely, for an arbitrary tuple $t \in R$ there exists a bucket B_j that contains $t[A]$. We say that a bucket B_i is *finest* if $B_i = [x, x]$ for a value x.

Example 2.4 If A represents age and the domain of A is non-negative integers bounded by 120, we can make 121 finest buckets $[i, i]$ for each $i = 0, 1, \ldots, 120$. When A shows balance of millions of customers in a bank, the domain of A may range from \$0 to $\$10^{10}$. In this case the number of finest buckets may amounts to millions. ∎

Linking consecutive buckets $B_s, B_{s+1}, \ldots, B_t$ creates a range $[x_s, y_t]$. Observe that if all buckets are finest, the combination of consecutive finest buckets gives the range of an optimized association rule. Given a large number (thousands, say) of buckets that may not be finest, an approximation of the range of an optimized rule can be obtained by joining consecutive buckets. Thus as ranges of rules, we only use those that are made of consecutive buckets.

We call the number of tuples in $\{t \in R \mid t[A] \in B_i\}$ the *size* of B_i, and denote it by u_i. B_i's are called *equi-depth* if the size of any B_i is the same. Let v_i denote the number of tuples in $\{t \in R \mid t[A] \in B_i, t \text{ meets } C\}$, and let N be the number of all tuples. $(\sum_{i=s}^{t} v_i)/(\sum_{i=s}^{t} u_i)$ gives the confidence of rule $(A \in [x_s, y_t]) \Rightarrow C$, and the support of $A \in [x_s, y_t]$ is $(\sum_{i=s}^{t} u_i)/N$.

For the purpose of computing u_i and v_i, for each tuple t we need to determine the bucket that $t[A]$ belongs to. One natural way of doing this check is to scan each tuple once and locate the bucket to which the tuple belongs by using a hash function or by building a ordered binary tree of finest buckets. This technique works fast for a huge database provided the number of finest buckets is small (recall the case of age in Example 2.4), but it may run very slowly if it has to handle millions of finest buckets, owing to the limited size of the main memory. Another natural method is to sort the given relation over A and divide the sorted data into finest buckets, but it takes enormous amount of time to sort a giant database that is much larger than the main memory.

The above discussion shows that the most difficult case is that in which the number of finest buckets is large and the size of the given database is huge. One such example may be the balances of millions of customers in a bank. In this case, for the sake of efficiency, we should avoid sorting a huge database and reduce the number of buckets to consider. Thus our approach is to generate a small number (say thousands) of buckets, which may not be finest, instead of making millions of finest buckets. We will make almost equi-depth buckets so that we can make good approximations of optimized rules. In the next section we present a way of making almost equi-depth buckets without sorting the data.

3 Making Equi-depth Buckets

We present a way of dividing N data into M buckets almost evenly.

3.1 Algorithm

Since we must avoid sorting data, as mentioned in the previous section, we use the following approximation algorithm:

Algorithm 3.1

1. Make an S-sized random sample from N data.
2. Sort the sample in $O(S \log S)$ time.
3. Scan the sorted sample and set the $i(S/M)$-th smallest sample to p_i for each $i = 1, \ldots, M-1$. Let p_0 be $-\infty$ and p_M be $+\infty$.
4. For each tuple x in the original N data, find i such that $p_{i-1} < x \leq p_i$ and assign x to the i-th bucket. This check can be done in $O(log M)$ time by using the binary search tree for the buckets. Thus, for all i, the size u_i of B_i can be computed in $O(N \log M)$ time.

∎

The complexity of this algorithm is

$$O(max(S \log S, N \log M)).$$

In practice $S \ll N$, and hence the complexity is $O(N \log M)$. We evaluated the performance of this algorithm with very large sets of data (containing up to ten million tuples) and found that the computation time grows almost linearly in proportion to the number of data. See Subsection 5.1.

3.2 Sample Size

How many samples are enough to generate almost equi-depth buckets ?

Let B_j be an arbitrary bucket among M buckets of N data generated by Algorithm 3.1. B_j is expected to contain N/M original data, and we may suppose that B_j actually contains N_j original data. We can precisely compute the following probability p_e for δ and M by using the binomial distribution:

$$p_e = Pr(|N_j - N/M| \geq \delta(N/M)).$$

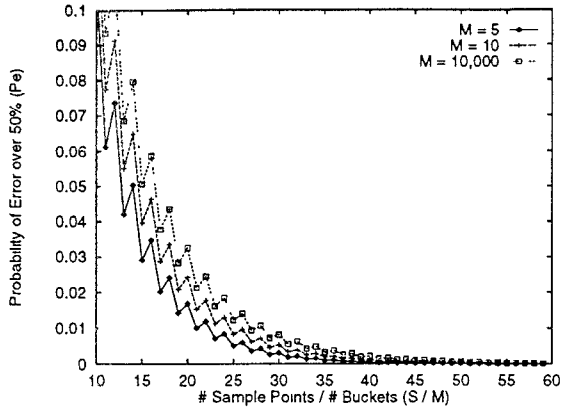

Figure 1: Sample size and probability of the error being over 50%

p_e does not depend on N. Figure 1 shows the relationship between p_e and S/M for $\delta = 0.5$ and $M = \{5, 10, 10000\}$. For every value of M, p_e goes down sharply when $S/M < 40$. It becomes smaller than 0.3% when $S/M = 40$, and it does not decrease much when $S/M > 40$. Thus, in our implementation we use $40 \cdot M$ as S.

In practice, a value of at most $M = 10^4$ is precise enough to allow us to derive approximate rules, and a $(4 \cdot 10^5)$-sized sample fits into the main memory. Subsection 5.1 presents performance results for Algorithm 3.1 and a naive method using Quick Sort.

3.3 Parallel Bucketing

The most time-consuming part of Algorithm 3.1 is Step 4, which scans the entire database to find buckets for all tuples. Because we want to know only the size of each bucket, we can easily perform Step 4 in parallel:

1. Randomly distribute the tuples in the database to processor elements (PEs) almost evenly.
2. At a coordinating PE, execute Steps 1, 2, and 3.
3. At each PE, perform Step 4; namely, scan the divided data and count the number of data in each bucket.
4. Gather the results from all PEs to the coordinating PE and compute their sum.

No communication is necessary during the counting process, and hence we expect that this algorithm will be scalable to the number of PEs.

4 Algorithms

As explained in Subsection 2.3, if all buckets are "finest" or if plenty of buckets are given, we can focus on rules whose ranges are combinations of consecutive buckets, and therefore we give algorithms for computing optimized rules among such rules. Precisely, given a sequence of buckets $B_1, B_2, \ldots, B_M$, we focus on rules of the form

$$(A \in [x_s, y_t]) \Rightarrow C,$$

where $[x_s, y_t]$ is a combination of consecutive buckets $B_s, B_{s+1}, \ldots, B_t$. Since any range $[x_s, y_t]$ is specified by a pair of indexes $s \leq t$, for simplicity, we denote $support(A \in [x_s, y_t])$ by $support(s,t)$ and denote $conf((A \in [x_s, y_t]) \Rightarrow C)$ by $conf(s,t)$ throughout this section. We also use the notations u_i and v_i for bucket B_i, defined in Subsection 2.3.

4.1 Optimized Confidence Rules

We call $s \leq t$ an *ample* pair if $support(s,t)$ is no less than the given minimum support threshold. Our goal is to find an ample pair $s \leq t$ that maximizes $conf(s,t)$.

To this end, consider the sequence of points $Q_k = (\sum_{i=1}^{k} u_i, \sum_{i=1}^{k} v_i)$ for $k = 1, \ldots, M$, and let Q_0 be $(0,0)$. Observe that the slope of the line $Q_m Q_n$ gives $conf(m+1, n)$, and the x-coordinate of Q_m minus the x-coordinate of Q_n is equal to $support(m+1, n)$. Our goal is transformed into the problem of finding an ample pair $m+1 \leq n$ that maximizes the slope of $Q_m Q_n$, which is $conf(m+1, n)$. We call m and n an *optimal confidence pair* (if more than one pair has the same maximum slope, select a pair that maximizes $support(m+1, n)$).

To solve this problem, we use a technique of handling convex hulls, for which we introduce some special terms. Let S be a set of distinct points. A *convex polygon* of S has the property that any line connecting any two points of S must itself lie entirely inside the polygon. The *convex hull* of S is the smallest convex polygon of S. Let v_{min} be the node in S with the minimum x-coordinate, and let v_{max} be the node in S with the maximum x-coordinate. Observe that v_{min} and v_{max} are on the convex hull of S. From v_{min} we can visit nodes on the convex hull of S in clockwise (counterclockwise) order until we hit v_{max}, and we call the set of nodes visited the *upper* (*lower*) hull of S.

Let U_m denote the upper hull of $\{Q_m, \ldots, Q_M\}$, and let $r(m)$ be

$$min\{i \mid m+1 \leq i \text{ is an ample pair}\}.$$

Now consider the tangent of Q_m and $U_{r(m)}$, and suppose that the tangent touches $U_{r(m)}$ at Q_t, as illustrated in Figure 2. Q_t is called the *terminating* point of the tangent (if the tangent touches more than one node of $U_{r(m)}$, select the node with the maximum x-coordinate as Q_t). It is easy to see that if $m \leq n$ is an optimal confidence pair, Q_n is the terminating point of the tangent of Q_m and $U_{r(m)}$. Thus, we need to find the tangent of Q_m and $U_{r(m)}$ with the maximum slope among all m. To this end, we will present an algorithm whose computational complexity is linear in the number of buckets.

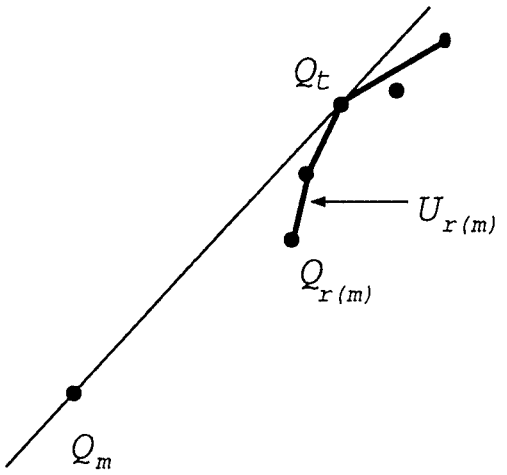

Figure 2: The inner tangent of Q_m and $U_{r(m)}$

Online Maintenance of Convex Hulls

The algorithm frequently scans the nodes on upper hull $U_{r(m)}$ in clockwise or counter-clockwise order for each $m = 1, \ldots, M$. To speed up this search, we present a way of accessing the two neighbors of each node on an arbitrary upper hull in a constant time.

Algorithm 4.1 Suppose that we are given a sequence of nodes $Q_0, Q_1, \ldots, Q_M$ that is the sorted list with respect to the x-coordinate value, and let U_i denote the upper hull of $\{Q_i, \ldots, Q_M\}$. Let S and D_i for each $i = 1, \ldots, M$ be empty stacks. Our goal is to make S such that the top-to-bottom order of nodes in S corresponds to the clockwise order of nodes on U_i for each $i = 0, \ldots, M$. This property of S enables us to access the neighbors of each node Q on U_i by looking at the upper node and the lower node of Q in S.

Preparatory Phase: We make S store U_i for each $i = M, \ldots, 0$. When $i = M$, push Q_M onto S, which trivially makes S store U_M. For each $i = M-1, \ldots, 0$, visit each node Q on U_{i+1} from Q_{i+1} in clockwise order (visit each node in S in top-to-bottom order), and find the Q_iQ with the maximum slope. This search is done by performing the following procedure:

> **Clockwise Search:** If the slope of Q_i and the top node of S is less than or equal to the slope of Q_i and the second top of S, the top node is no longer a node on U_i; in this case, pop the top node from S, push it onto D_i, and repeat this check. Otherwise, the slope of Q_i and the top node is maximum; in this case, push Q_i onto S.

D_is are used for recording the nodes deleted at each step. Since at most $M-1$ nodes are popped from S in the above check, the check is executed at most $2(M-1)$ times, and hence the time and space complexities of clockwise search are $O(M)$. After the termination, S stores U_0.

Restoration Phase: Conversely, for each $i = 0, \ldots, M-1$, pop the top node Q_i from S and push back all nodes of D_i in top-to-bottom order onto S, thus making S store U_{i+1}. ∎

Example 4.1 Consider nodes $Q_0, Q_1, \ldots, Q_9$ in Figure 3. The dotted line from Q_i to Q_9 shows the upper hull of $\{Q_i, \ldots, Q_9\}$. For each $i = 9, \ldots, 0$, Algorithm 4.1 generates the following states of S:

						Q_3			
					Q_4	Q_4	Q_2		Q_0
		Q_7	Q_6	Q_5	Q_5	Q_5	Q_5	Q_1	Q_1
	Q_8	Q_8	Q_8	Q_8	Q_8	Q_8	Q_8	Q_8	Q_8
Q_9	Q_9	Q_9	Q_9	Q_9	Q_9	Q_9	Q_9	Q_9	Q_9
9	8	7	6	5	4	3	2	1	0

							Q_4	Q_5	
			Q_7	Q_6			Q_3	Q_2	
D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

∎

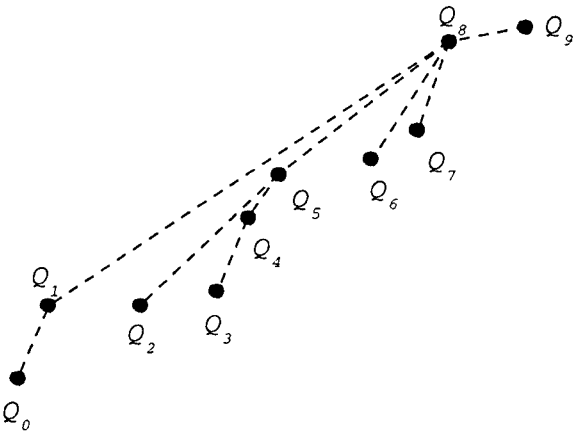

Figure 3: Upper Hulls

Computing Tangents

Now we are in a position to present the algorithm for computing the tangent of Q_m and $U_{r(m)}$ with the maximum slope among all m.

Algorithm 4.2 Given $Q_0, Q_1, \ldots, Q_M$, perform the preparatory phase of Algorithm 4.1. In order to have $U_{r(m)}$ for each $m = 1, \ldots, M$, we execute the restoration phase of Algorithm 4.1, which takes $O(M)$ time. We use variable L to store the tangent of Q_m and $U_{r(m)}$ for some m.

Base Step: Find the terminating point of the tangent of Q_0 and $U_{r(0)}$ by clockwise search; that is, visit each node Q on $U_{r(0)}$ from $Q_{r(0)}$ in clockwise order, find the Q_0Q with the maximum slope, and set the tangent to L.

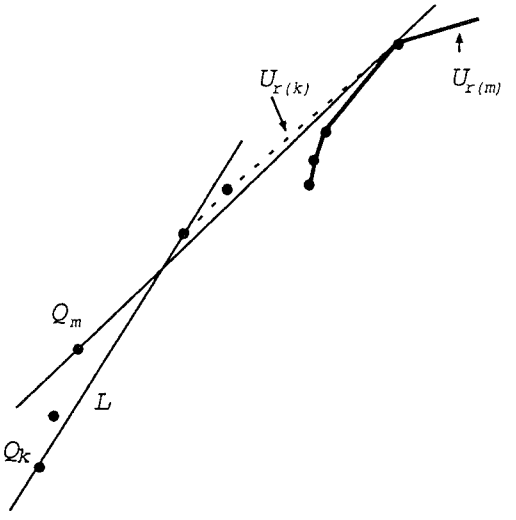

Figure 4: Leaving L untouched

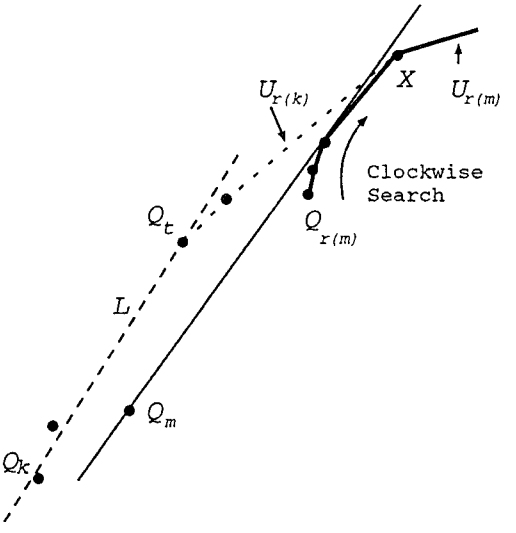

Figure 5: Clockwise Search

Inductive Step: For each $m = 1, \ldots, M$, while $U_{r(m)}$ is not empty, assume that L stores the tangent of Q_k and $U_{r(k)}$ for some $k < m$ and perform one of the following steps:

- If Q_m is above or on L, the slope of the tangent of Q_m and $U_{r(m)}$ is not greater than that of L. Figure 4 illustrates an example of this case. We do not compute the tangent of Q_m and $U_{r(m)}$, and leave L untouched.

- Otherwise, let Q_t be the terminating point of L, compute the tangent of Q_m and $U_{r(m)}$ by one of the following steps, and set the tangent to L:

  - If L does not touch $U_{r(m)}$ (Q_t is on the left-hand side of $Q_{r(m)}$), find the terminating point of Q_m and $U_{r(m)}$ by clockwise search; that is, visit each node Q on $U_{r(m)}$ from $Q_{r(m)}$ in clockwise order, and find the Q_mQ with the maximum slope. Figure 5 illustrates this case.

    Among all nodes both on $U_{r(k)}$ and $U_{r(m)}$, let X denote the node with the minimum x-coordinate. The above clockwise search only scans edges from $Q_{r(m)}$ to at most X; otherwise, X cannot be on $U_{r(k)}$. Since edges between $Q_{r(m)}$ and X are hidden inside $U_{r(k)}$, those edges have never been scanned in this algorithm.

  - Otherwise, L touches $U_{r(m)}$ at Q_t. Find the terminating point of Q_m and $U_{r(m)}$ by counter-clockwise search; that is, visit each node Q on $U_{r(m)}$ from Q_t in counter-clockwise order, and find the Q_mQ with the maximum slope. Figure 6 illustrates this case.

    Note that edges between $Q_{r(m)}$ and Q_t have never been scanned before in this algorithm.

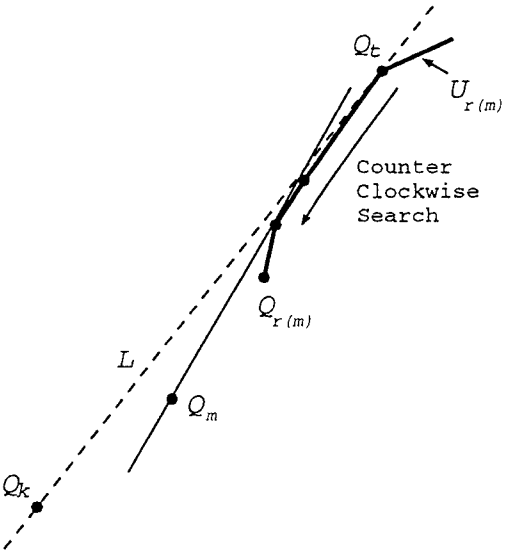

Figure 6: Counter-clockwise Search

Final Step: Among all tangents that have been set to L, select the ones with the maximum slope. ∎

Theorem 4.1 *Algorithm 4.2 computes all optimal confidence pairs in $O(M)$ time.*

Proof: Let S denote the set of edges on $U_{r(m)}$ for all m. Both the clockwise search and the counter-clockwise search in the algorithm scan each edge in S at most once. Since the number of edges in S is at most $M - 1$, the algorithm computes tangents with the maximum slope in $O(M)$ time. ∎

4.2 Optimized Support Rules

Let θ be the given minimum confidence threshold. Our goal is to find a value of $s \leq t$ that satisfies $conf(s,t) \geq \theta$ and maximizes $support(s,t)$, which is $\sum_{i=s}^{t} u_i$ (call s and t an *optimal support pair*). Let us call s *effective* if $conf(j, s-1) < \theta$ for every $j < s$.

Lemma 4.1 *If $s \leq t$ is an optimal support pair, s is effective.*

Proof: Otherwise, there exists j such that

$$conf(j, s-1) \geq \theta.$$

Since $s \leq t$ is an optimal support pair, $conf(s,t) \geq \theta$, and hence $conf(j,t) \geq \theta$, which contradicts the optimality of s and t. ∎

From the above lemma, we will find all effective indices and choose an optimal support pair. Let w be $max_{j<s} \sum_{i=j}^{s-1}(v_i - \theta u_i)$ for each index s. Then, note that s is effective iff $w < 0$. The following algorithm computes w for all indices in $O(M)$ by scanning buckets forwards, and gives the set of all effective indices.

Algorithm 4.3

```
1 is effective;
w := 0;
for each s := 2 to M begin
    if(w < 0 and v_{s-1} - θu_{s-1} ≥ 0)
        then w := v_{s-1} - θu_{s-1};
        else w := w + (v_{s-1} - θu_{s-1});
    if(w < 0) then s is effective;
end
```

∎

Let $top(s)$ denote the largest index t such that $s \leq t$ and $conf(s,t) \geq \theta$. The final step is to choose s maximizing $\sum_{i=s}^{top(s)} u_i$.

Lemma 4.2 *If $s < s'$ are effective, $top(s) \leq top(s')$.*

Proof: Since s' is effective, $conf(s, s'-1) < \theta$. From the definition of $top(s)$, $conf(s, top(s)) \geq \theta$. Then, it follows that $conf(s', top(s)) \geq \theta$, which implies that $top(s) \leq top(s')$. ∎

Thanks to this property, we only need to scan backwards through the list of effective indices $(s(1), \ldots, s(q))$ and the list of all indices $(1, \ldots, M)$ alternately to find $top(s(i))$. We can do this by means of the following algorithm:

Algorithm 4.4

```
i := M
for each j from q to 1
    if(conf(s(j), i) < θ) then i := i - 1;
    else begin top(s(j)) := i; j := j - 1; end
```

∎

In the above algorithm, we pre-compute a cumulative table $F(j) = \sum_{i=j}^{s} v_i - \theta \sum_{i=j}^{s} u_i$. Since

$$conf(s(j), i) < \theta \text{ iff } F(i) - F(s(j)-1) < 0,$$

we can check $conf(s(j), i) < \theta$ in a constant time. Thus both Algorithms 4.3 and 4.4 run in $O(M)$ time, we have:

Theorem 4.2 *All optimal support pairs can be computed in $O(M)$ time.*

In the algorithm literature, Bentley [Ben84] introduced a linear time algorithm (Kadane's Algorithm) that computes a range I maximizing $\sum_{i \in I} x_i$ against an array of real numbers x_i. If every x_i is non-negative, trivially $[1, M]$ gives the solution, so we assume that some elements are negative. Let $a(j)$ denote $\max\{\sum_{i \in [s,t]} x_i \mid s \leq t \leq j\}$, then the interval of $a(M)$ is our answer. For computing $a(j)$ we introduce another auxiliary data $b(j)$ that denotes $\max\{\sum_{i \in [s,j]} x_i \mid s \leq j\}$ Then, the following relations hold.

$$\begin{aligned} b(j+1) &= \max\{0, b(j)\} + x_{j+1} \\ a(j+1) &= \max\{b(j+1), a(j)\} \end{aligned}$$

Set 0 to $b(0)$. Then, a simple dynamic programming gives linear time solution.

For a confidence threshold θ, call $\sum_{i \in I}(v_i - \theta u_i)$ the *gain* of range I. If $v_i - \theta u_i$ is set to x_i, Kadane's Algorithm computes the range that maximizes the gain, which is not equivalent to the range of the optimized support rule. The essence of Kadane's Algorithm is dynamic programming, which unfortunately does not work for finding optimized rules.

4.3 Generalization of Optimized Rules

Our algorithms can be straightforwardly extended to compute rules of the general form

$$(A \in [v_1, v_2]) \wedge C_1 \Rightarrow C_2,$$

where C_1 and C_2 are Boolean statements that do not contain any variables on numeric attributes. Let u_i be the size of $\{t \in R \mid t[A] \in B_i, t \text{ meets } C_1\}$, let v_i be the size of $\{t \in R \mid t[A] \in B_i, t \text{ meets } C_1 \text{ and } C_2\}$, and apply our algorithms to this case.

5 Performance Results

The proposed algorithms have been written in C++. We evaluated their performance on an IBM Power Series 850 with a CPU clock rate of 133 MHz and 96 MB of main memory, and running AIX 4.1.

5.1 Making Buckets

We randomly generated test data with 8 numeric attributes and 8 Boolean attributes — 72 bytes per tuple. The test data resided in the AIX file system on

a 3.5" 1.2GB IDE drive. As a test case, we divided the test data into 1000 buckets with respect to each numeric attribute and counted the number of tuples in every bucket for each Boolean attribute. We compared the performance of our bucketing algorithm, Algorithm 3.1, with the following two methods. One method, which we call *Naive Sort*, sorts data for each numeric attribute by using Quick Sort. The other one, which we call *Vertical Split Sort*, first splits data vertically to generate a smaller table with tuple identifier and each numeric attribute, and then sort the temporary table.

Figure 7 shows the execution time for data of tuples ranging from $5 \cdot 10^5$ to $5 \cdot 10^6$. For large data sets with more than one million tuples, Algorithm 3.1 outperforms the naive method by more than an order of magnitude, and it also beats Vertical Split Sort by a factor of 2 to 4. Furthermore, the execution time of Algorithm 3.1 grows almost linearly in the number of data.

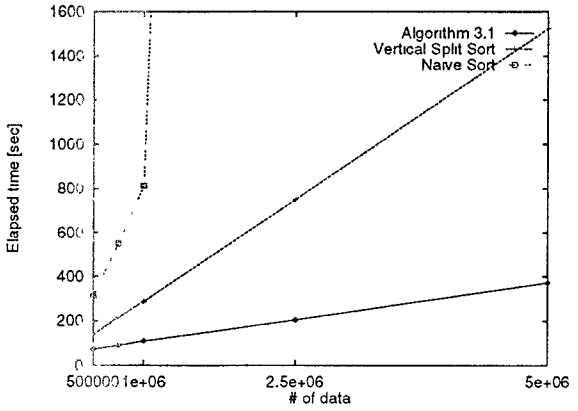

Figure 7: Performance of Bucketing Algorithms

5.2 Finding Optimized Rules

We evaluated the performance of the algorithms for finding optimized rules, comparing the results with those of a naive method that computes, in quadratic time, the confidence and support of all ranges in order to find an optimal one.

Figures 8 and 9 respectively show the execution times for finding optimized confidence rules with a 5% support threshold and optimized support rules with a 50% confidence threshold for buckets whose sizes are from 100 to 1,000,000. Our algorithm for finding optimized confidence rules beats the naive method by more than an order of magnitude for more than 500 buckets. For finding optimized support rules, our algorithm is also faster than the naive method by more than an order of magnitude for data of more than 100 buckets. Even for small data sets, both of our algorithms outperform the naive ones. The execution time of both algorithms increases linearly in the number of buckets.

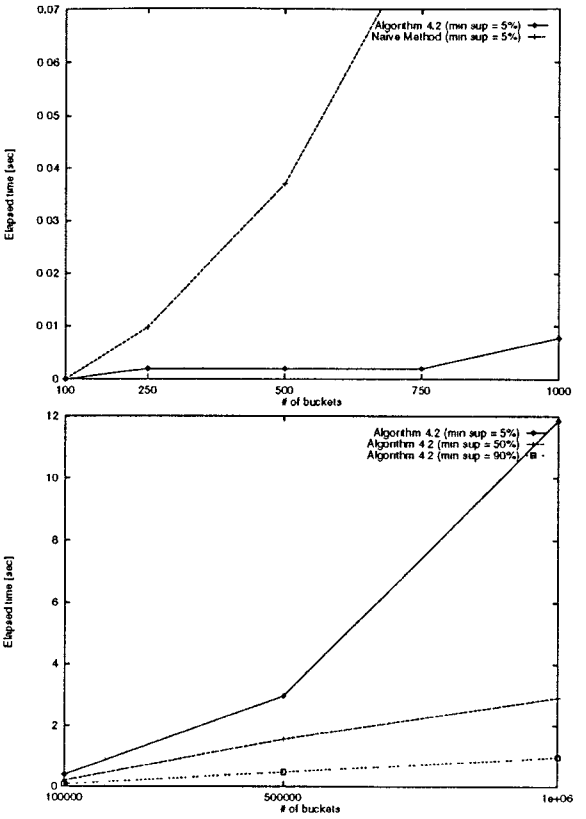

Figure 8: Finding Optimized Confidence Rules

References

[AGI+92] R. Agrawal, S. Ghosh, T. Imielinski, B. Iyer, and A. Swami. An interval classifier for database mining applications. In *Proceedings of the 18th VLDB Conference*, pages 560–573, 1992.

[AIS93a] Rakesh Agrawal, Tako Imielinski, and Arum Swami. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993.

[AIS93b] Rakesh Agrawal, Tako Imielinski, and Arum Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 207–216, May 1993.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th VLDB Conference*, pages 487–499, 1994.

[Ben84] Jon Bentley. Programming pearls. *Communications of the ACM*, 27(27):865–871, September 1984.

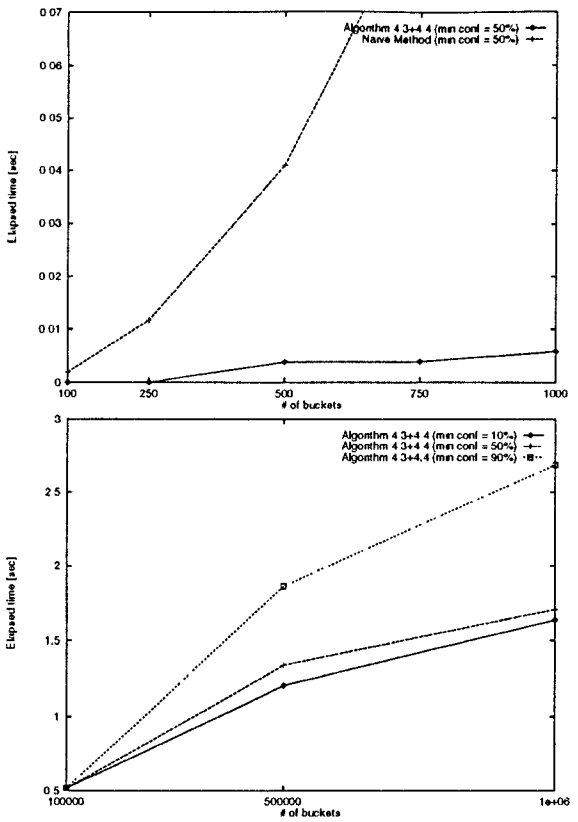

Figure 9: Finding Optimized Support Rules

[BFOS84] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth, 1984.

[FMMT96] Takeshi Fukuda, Yasuhiko Morimoto, Shinichi Morishita, and Takeshi Tokuyama. Data mining using two-dimensional optimized association rules: Scheme, algorithms, and visualization. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, June 1996.

[HCC92] Jiawei Han, Yandong Cai, and Nick Cercone. Knowledge discovery in databases: An attribute-oriented approach. In *Proceedings of the 18th VLDB Conference*, pages 547–559, 1992.

[MAR96] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. Sliq: A fast scalable classifier for data mining. In *Proceedings of the Fifth International Conference on Extending Database Technology*, 1996.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[NH94] Raymond T. Ng and Jiawei Han. Efficient and effective clustering methods for spatial data mining. In *Proceedings of the 20th VLDB Conference*, pages 144–155, 1994.

[PCY95] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash-based algorithm for mining association rules. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 175–186, May 1995.

[PS91] G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. In *Knowledge Discovery in Databases*, pages 229–248, 1991.

[PSF91] G. Piatetsky-Shapiro and W. J. Frawley, editors. *Knowledge Discovery in Databases.* AAAI Press, 1991.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[SA96] Ramakrishnan Srikant and Rakesh Agrawal. Mining quantitative association rules in large relational tables. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, June 1996.

[SAD+93] Michael Stonebraker, Rakesh Agrawal, Umeshwar Dayal, Erich J. Neuhold, and Andreas Reuter. DBMS research at a crossroads: The vienna update. In *Proceedings of the 19th VLDB Conference*, pages 688–692, 1993.