Algorithms for Data Mining

Tadao Takaoka
Department of Computer Science and Software Engineering
University of Canterbury
Christchurch, New Zealand

1. Introduction

Data mining is to extract useful information from a vast amount of data, typically from a large database. Here useful information means some interesting information that can be found only going through a large database with a computer, which a human can never scan through with bare eyes and hands. The database can be that of sales data of a supermarket, image data such as X-ray images for medical records, etc. Interesting information may be customers' purchasing behavior in the sales database, or some abnormality in medical images. As the size of the database is measured by gigabytes stored in disk space, algorithms that deal with the data must be not only fast, but also need to make minimum access to the disk, as access to a disk is expensive in terms of accessing time. We use artificial examples for explanation purposes. In the section of experiments, however, we show results based on actual sales data.

We can analyze, or mine, a database from various angles. Mining an association rule in a sales database, for an example of data mining, is an activity of finding a relationship of purchased items looking like "A customer who buys cereal is likely to buy milk." Knowing such rules can help management decision in many ways. For example, a manager can promote these items together to increase customers' satisfaction.

In the following we give an outline of data mining using a specific example. Suppose we have the following records of sales at a food supermarket, which include purchased items as well as several attributes of customers.

Example 1.

customer	items	total amount spent
1	ham, cheese, cereal, milk	$42
2	bread, cheese, milk	$22
3	ham, bread, cheese, milk	$37
4	bread, milk	$12
5	bread, cereal, milk	$24
6.	ham, bread, cheese, cereal	$44

customer	name	gender	age	annual income	address
1	Anderson	female	33	$20000	suburb A
2	Bell	female	45	$35000	suburb A
3	Chen	male	28	$25000	suburb B
4	Dickson	male	50	$60000	suburb B
5	Elias	male	61	$65000	suburb A
6.	Foster	female	39	$45000	suburb B

In the above example, four out of five customers who bought “bread” also bought “milk”, so there is an association rule “bread → milk”. Since four out of five customers bought milk with bread, its probability is 80%, or 0.8. We call this value “confidence” or “confidence level”.

The leftmost numbers 1, 2, 3, 4, 5 and 6 correspond to customers, and “ham”, “cheese”, etc., correspond to items those customers purchased. Actually as the same customers may return several times, those numbers may be regarded as transaction numbers. Customers can be identified by names. For simplicity, we regard them as the same. The first table includes actual transactions, and the second includes attributes of those customers. Based on the database such as above, we want to analyze and predict a purchaser’s behavior. The size of the database is measured by the number of transactions, N, which is six in the above example, purchased items, customer attributes, etc. In real world applications, N is a few million. This kind of analysis based on purchased items is sometimes called “basket analysis” as it is like items in the basket of a customer are analyzed using a point-of-sales equipment and a computer. The following are typical data mining methods developed in the past decade.

Mining association rules. Suppose we want to identify some rule that can say if a customer buys item X, he is likely to buy item Y. We measure this likelihood, denoted by X → Y and called an association rule, by the formula of confidence given by

$$\text{conf}(X \rightarrow Y) = \text{support}(X, Y)/\text{support}(X)$$

X is called the antecedent and Y the consequence. Here support(X, Y) is the number of transactions that include both X and Y, and support(X) is that for X. This definition is similar to that of apriori probability in probability theory. In general, X and Y can be sets of items. Also we need to have some support for the occurrences of X and Y in transactions. The problem here is how to efficiently find a rule X → Y with the maximum confidence and at least the minimum support. The minimum support is given either by a fraction or absolute value. We need a minimum support because we can not take an action based on a high confidence rule with just a few exceptional examples. As we need to deal with gigabytes for the database, fast algorithms are essential. There is a legend that a rule “nappy → beer” was found in the United States. Also in a UK super market, it was found that high income customers tended to buy luxurious cheese with some confidence, but rather low support. If they stop selling this kind of cheese due to low sales amount, they could lose those valuable customers. Note that the lower the support, the more candidate transactions we need to handle to find promising association rules.

In the above example, all of the three customers who bought “cheese” also bought “ham”, and thus conf(ham → cheese) = 3/3 = 1, the highest confidence with support(ham, cheese) = 1/2 if the minimum support is 1/2 in fraction, or 3 in absolute value. If the minimum support is 2/3, we can identify the highest confidence with the rule of “bread → milk” with conf(bread → milk) = 4/5. After we identify that “ham → cheese” is very likely, we can set up several strategies. One is to put the sales position of “cheese” side by side with that of “ham”. Another strategy may be to send advertisement for “cheese” to customers who bought “ham”.

In this chapter we introduce a fast algorithm for finding promising association rules developed by Agrawal, et al [1] and Mannila et. al. [14], and also discuss possible extensions such as negative and hierarchical rules. General discussions on association rules are given in the textbooks [10], [6], [11] and [23].

Mining association rules with numerical attributes. Using numerical attributes of customers, we want to estimate purchasers' behavior. This problem was originated by Srikant and Agrawal [17]. Specifically, we use numerical attributes in the antecedent. In our example, let us use the condition "age < 40" for the antecedent. Then conf(age < 40 → ham) = 1. If we set the range to "age ≤ 50", however, the confidence becomes 3/5. Thus the advertisement for "ham" should be sent to customers younger than 40.

This problem can be formalized by the maximum subarray problem, which is to maximize the sum of a consecutive portion in an array. Suppose we have a one-dimensional array a of size six such that a[i] is the number of customers who are between 10*i and 10*(i+1)-1years of age and bought "ham". Then we have a = (0, 1, 2, 0, 0, 0). The most promising age range of customers for "ham" can be identified by finding a subarray that maximizes the sum. If array elements are non-negative, the whole array becomes the solution. Thus we normally subtract the mean value from the array elements. In our example array a is converted to a = (-½, ½, 1, -½, -½, -½), and the array portion a[2..3] becomes the solution, that is, customers in their twenties or thirties are most promising for "ham".

If we use two numerical attributes for the antecedent, the problem becomes the two-dimensional maximum subarray problem. This area has another application in graphics. If the array elements represent the intensity of light in an image file, the maximum subarray can identify the brightest portion in the image. In this chapter we describe fast algorithms for the one-dimensional and two-dimensional subarray problems. The maximum subarray problem was originated by Bentley [2, 3], and later its relationship with data mining was noticed in [21] with more efficient algorithms. Further improved algorithms are reported in [19]. The graphic application can also be used for movement detection in video images. If we take the difference of two consecutive images pixel by pixel, the maximum subarray gives us a portion that includes a candidate object that moved between the two images.

In this chapter we focus on the above two areas, and discuss algorithmic issues; association rules are the most fundamental area in data mining, and the maximum subarray problem is an emerging area for which there has been no textbook description. The latter area is becoming important as a tool for data mining in image data.

An overview of more areas of data mining is given in the section of Concluding Remarks. Algorithms in this chapter are given in pseudo C code. The codes for mining association rules are rather informal, intended for giving the concepts of mining association rules, whereas those for the maximum subarray problem are more technical and detailed, ready for use on a computer. If the reader skips those codes in Section 3, he/she will still be able to grasp the design concepts of those algorithms. The full implementations are given at the end of the chapter, which can be modified and adapted to the readers' various applications.

2 Association rules

For the development of algorithms, we repeat sales records in Example 1 with symbolic names for purchased items below.

Customer	items
1	A, C, D, E
2	B, C, E
3	A, B, C, E
4	B, E
5	B, D, E
6	A, B, C, D

2.1 Formal definition of association rules and lexicographic order

An association rule in database D is given by the form

$$X_1, X_2, \ldots, X_m \rightarrow Y_1, Y_2, \ldots, Y_n,$$
$$\text{antecedent} \rightarrow \text{consequence}$$

The left-hand side is called “antecedent” and right-hand side “consequence”. The meaning of this rule is that a person who buys $X_1, \ldots, X_m$ is likely to buy $Y_1, \ldots, Y_n$. The degree of confidence “conf” is defined by

$$\text{conf} = \text{support}(X_1, \ldots, X_m, Y_1, \ldots, Y_n) / \text{support}(X_1, \ldots, X_m),$$

where support($Z_1, \ldots, Z_k$) is the number of transactions that include $Z_1, \ldots, Z_k$ in database D. We define support(empty)=N. Thus if the lefth-and side is empty, “conf” is the proportion of support($Y_1, \ldots, Y_n$) in all transactions.

A k-tuple is an ordered set of k items, eg., (A,B,C) is a 3-tuple. A set of items, or simply item set, is expressed by a k-tuple if it has k items arranged in alphabetic order. Thus item set {B,C,A} is expressed by a 3-tuple (A,B,C).

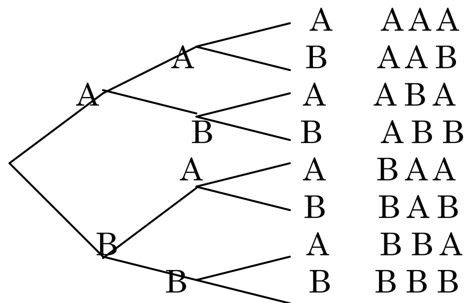
We express by C_k the set of candidate k-tuples for analyzing the database, eg., C_2 = {(A,B), (A,C), (B, D)}

We express by L_k the set of frequent k-tuples, which appear in transactions at least the number of times given by the minimum support. We call the number of times of a tuple occurring in all transactions its frequency. A frequent tuple is called a large item set in some literature. We denote the minimum support in ratio by “minsup”.

Example 2. Using Example 1, we have the following calculations.
minsup=0.5, n=6, minimum frequency=0.5*6=3, frequency in ( ).
k=1, A(3), B(5), C(4), D(3), E(5)
k=2, (A,C)(3), (B,C)(3), (B,E)(4), (C, E)(3)
k=3, (B,C,E)(2). This k-tuple is not considered for rules, because it appears 2 times.
conf(B → E) = 4/5 with support 4, conf(A → C) = 1 with support 3.

Generating words in lexicographic order

The intuitive meaning of lexicographic order is that those words appear in a dictionary in this order. More formally, if $\mathbf{a}=a_1a_2\ldots a_n$ and $\mathbf{b}=b_1b_2\ldots b_n$ are words of length n, we define $\mathbf{a} < \mathbf{b}$, if for some i ($1\leq i \leq n$), $a_1=b_1$, …, $a_{i-1}=b_{i-1}$, and $a_i < b_i$, where order "<" on symbols are defined by the alphabetic order. All words of length 3 consisting of letters "A" and "B" in lexicographic order are listed below. We can associate the words at the leaves of a tree.



Labels A and B are attached to nodes in the tree in such a way that A and B are attached to the two children of each node in this order. Thus each word attached to a leaf can be identified by the path from the root to the word. Similar tree structures are used when we generate item sets in lexicographic order in the next section.

2.2 Generating frequent item sets

We use the following obvious lemma to avoid many unnecessary item sets.

Lemma 1. Let S and T be item sets. Then we have $T \supseteq S \Rightarrow \text{support}(T) \leqq \text{support}(S)$.

If we generate all k-tuples and check their frequencies in the database, we hit what is called combinatorial explosion, which can become very time consuming. Instead, using Lemma 1, we can proceed from the k-th stage to the (k+1)-th stage by generating (k+1)-tuples that include only k-tuples generated at the k-th stage. To avoid confusion in the following, we use "minfreq" to express the minimum support in the absolute number, not a fraction. Also, "c.count" is the counter for the tuple "c" to count the number of times tuple "c" appearing in the database. We call the work by the following algorithm the tuple generation phase. The maximum size of tuples becomes M at the end. The smaller "minfreq" is, the larger the value of M. Note that we try to make access to database residing in disk space at minimum. At line 4, we make access to database D to get transaction t, and we process C_k and L_k in internal memory. Thus scanning the database occurs M times.

Algorithm Apriori(D,minfreq){

1. Insert all the single items into C_1
2. Set k=1

3. While C_k is not empty, do{
4. For all the transactions t in D, do{
5. Extract into C all k-tuples in C_k that appear in t
6. For all tuples c in C, increment the count of c, c.count
7. }
8. Extract all the tuples whose count is greater than or equal to minfreq into L_k
9. Generate the set C_{k+1} of tuples of size k+1 using Apriori_Gen(L_k)
10. Set M = k
11. Increment k
12. }
13. Return the union of L_1, ..., L_M
14. }

Function Apriori_Gen is to generate candidate (k+1)-tuples from L_k based on the left longest match as illustrated in the following example, Example 3. When we generate C_{k+1} from L_k, we pick up a k-tuple, x, from L_k, and scan the list for k-tuple y that has the same (k–1)-tuple as that of x from position 1 to k-1. We call the common (k–1)-tuple a common prefix. Then we concatenate the last element of y to the end of x, resulting a new (k+1)-tuple for C_{k+1}. Let last(x) be the last element of x.

In the following algorithm, $prefix_{k-1}(p)$ is the prefix of length k-1 of k-tuple p. In concatenation phase we use the lexicographic order in tuples. For example we have (A, B, C)<(A, B, D). In line 4, items in p and q match except for the last ones. The removal phase is understood from the fact that if (A, B, C, D) is made for C_4, and (B, C, D) is not frequent, (A, B, C, D) is not needed for future generation of frequent tuples from Lemma 1.

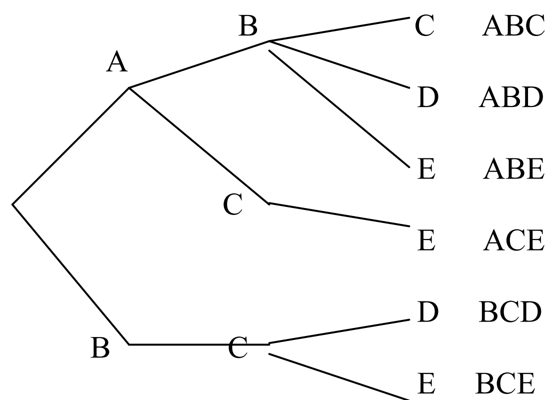
Algorithm Apriori_Gen(L_k){
1. /* concatenation phase */
2. Empty C_{k+1}
3. For each p and q in L_k such that $prefix_{k-1}(p) = prefix_{k-1}(q)$ and $p<q$, do{
4. Let c be (k+1)-tuple of (p, last(q))
5. Add c to C_{k+1}
6. }
7. /* removal phase for unnecessary tuples */
8. For each tuple p in C_{k+1},do{
9. For each k-tuple s that is a sub-tuple of p, do{
10. If s is not in L_k, delete p from C_{k+1}
11. }
12. }

Example 3. Frequencies are in parentheses following tuples. Let minsup=1/2.
C_1 = (A), (B), (C), (D), (E), L_1 = (A)(3), (B)(5), (C)(4), (D)(3), (E)(5)
C_2 = (A,B), (A,C), (A,D), (A,E), (B,C), (B,D), (B,E), (C,D), (C,E), (D,E)
L_2 = (A,C)(3), (B,C)(3), (B,E)(4), (C,E)(3)
C_3 = (B,C,E)(2), L_3 =φ (empty), M=3

From L_1 to C_2, we generate all 10 possibilities. From L_2 to C_3, (B,C) and (B,E) have common prefix of B. Thus we create (B,C,E).

When we create C_{k+1} from L_k we repeatedly scan the list L_k, which is time consuming. If we store those lists in the tree form, "Apriori_Gen" can be implemented efficiently. When we have a list of k-tuples, L_k, we store them at the leaves of a tree. Each node has branches to child nodes. Each branch corresponds to an item, and the path from the root to a leaf corresponds to the k-tuple represented by the leaf.

Example 4. Let L be given by {(A,B,C), (A,B,D), (A,B,E), (A,C,E), (B,C,D), (B,C,E)}. The list L can be implemented by the following tree.



Empty branches are not shown in this figure. When we traverse this tree from the root in depth-first manner, and reach a leaf, we can append the last items of the sibling leaves to the k-tuple at the leaf, and we can carry on for other leaves. In this example, we generate ((A,B,C,D), (A,B,C,E), (A,B,D,E), (B,C,D,E)). The first two are generated from ABC and the two siblings ABD and ABE. The third is by ABD and ABE, etc. The last two are discarded by line 10. Note that this example is different from our on-going example, and given for explanation purposes.

2.3 Generating association rules

The next step is to obtain association rules from the list of frequent tuples of items. We use the following lemma in the algorithm.

Lemma 2. For disjoint item sets S and T, and item X included in S, and not in T, we have $conf(S \rightarrow T) \geq conf(S - X \rightarrow T + X)$. X can be generalized to an item set. The meaning is that if $S \rightarrow T$ does not have enough confidence, neither does $S - X \rightarrow T + X$.
Proof. The lemma follows directly from the definition as follows:

$conf(S \rightarrow T) = support(S, T)/support(S)$
$conf(S - X \rightarrow T + X) = support(S, T)/support(S - X)$

This lemma can be used for generating association rules efficiently. Suppose an item set w of size k is frequent. We divide w into two disjoint subsets S and T, and see if conf(S

$\rightarrow$ T) is at least "minfreq". We increase the size of T one by one. Such T's are maintained in H_1, H_2, … in the following algorithm. Specifically in line 6, we can go to H_{m+1} using apriori_gen. Also we can exclude an item set in line 10, if it is not in the right-hand side of a confident rule. The correctness of those two lines is supported by Lemma 2. We call the work done by the following algorithm the rule generation phase.

```
Algorithm for association rules{
  1.  For k from 2 upto M, do{
  2.    For each frequent tuple w in Lk, do {
  3.      Let H1 be a set of items h, where conf(w-h → h) ≥ minconf
  4.      Set m=1
  5.      While m ≤ k-2, do{
  6.        Construct the set Hm+1 from Hm using Apriori_Gen(Hm)
  7.        For each tuple h in Hm+1, do{
  8.          Calculate confidence by conf=support(w)/support(w-h)
  9.          If conf ≥ minconf, then output the rule (w-h → h)
 10.          Otherwise, remove h from Hm+1
 11.        }
 12.        Increment m
 13.      }
 14.    }
 15. }
```

Example 5. Trace with minsup = 1/2 and minconf = 2/3
k=2
L_2 = (A,C), (B,C), (B,E), (C,E). For each w in L_2, we have the following.
H_1 = (A,B,C,E) given by the union of the following :

{A, C}, obtained from C→A (3/4) and A→C (1)
{B}, obtained from C→B (3/4), and
{B, E}, obtained from E→B (4/5), and B→E (4/5)
{E}, obtained, and from C→E (3/4)

k=3. C_3 = (B,C,E), L_3 = empty

Trace with minsup=1/3 and minconf = 2/3. L_2 has all 10 pairs.
L_3 = {(A,B,C) (2), (A,C,D) (2), (A,C,E) (2), (B,C,E) (2)}
Let us trace only for w=(A,B,C).
H_1={A,B,C}, obtained from B,C → A (2/3), A,C → B (2/3), and A,B → C (1)
H_2 = {(A,B), (A,C), (B,C)}, only A → B,C (2/3) is output.
C_4 = {(A,C,D,E)}, L_4 = empty

2.4 Negative rules

Suppose conf(A → C) < conf(A, –B → C). The meaning here is that a purchaser who buys A and not B is more likely to by C than the one who buys just A. We call an association rule with negative notation such as –B a negative rule. Negative notation can appear in either the left-hand side or right-hand side. We can use the previous algorithms to find association rules with negative items by simply introducing –X for each X. But

this approach will potentially produce 2^k tuples for each k-tuple, and thus not very efficient. We can use the following lemma to reduce the number of tuples, and also association rules with sufficient confidence. We explain using small examples, but the results can be extended to larger tuples.

Lemma 3.

support(A, B) + support(A, –B) = support(A).
conf(A → B) + conf(A → –B) = 1

We can use the first formula to assess support(A, –B) by support(A) – support(A, B) during the generation phase of frequent item sets; if it is smaller than “minfreq” we can discard it. We can use the second formula in the generation phase of association rules in a similar way. In real applications, the first formula does not contribute very much to the speed-up, since support(A, –B) is much greater than support(A, B). For the second formula, normally we can assume the minimum confidence is at least 1/2. Then if conf(A → B) >1/2, we can discard the generation process for conf(A → –B). This heuristic based on the second formula leads to some speed-up during the rule generation phase.

For practical applications we generate (A, –B) only when (A, B) has the minimum support. This has no mathematical ground, but has some reasoning from a management point of view; only when (A, B) has enough support, we can talk about (A, -B). Also we can generate tuples with only one negative item, such as (A, –B, C), and examine a rule, such as A, –B → C, or A, C → –B, etc. We are interested in such a question as: “if a purchaser does not buy an item, can it influence his/her purchasing pattern for other items?” “Or, if he/she buys some items, is he/she unlikely to buy something?”. Even one negative item can give us some marketing strategy.

More discussions on negative rules can be found in [24].

2.5 Hierarchical rules

In many sales databases, commodities are classified categorically. For example a “jacket” can have a hierarchical structure of (clothes, jacket), and “spinach” may have (food, vegetable, spinach). Here “jacket” and “spinach” exist as bottom items. If we can not find a useful rule regarding bottom items due to lack of enough support, we may be able to find one if we go up the ladder of hierarchy one or more steps.

We express the hierarchy by a number of trees. The roots of the trees correspond to the broadest categories, such as “clothes” and “food”. We expand the database so that we have transactions with those upper items. If we simply use the previous algorithms on the expanded database, we may have frequent item sets such as (clothes, jacket), from which we generate a redundant rule such as “jacket → clothes”. The simplest approach may be to generate all possible rules including upper items and remove those redundant rules that include ancestors and descendants. But this method may generate too many redundant rules, and is thus not very efficient. To prevent this, we remove item sets that include ancestors and descendants at the tuple generation phase, and rule generation phase. Systematic approaches to hierarchical rules are reported in [18] and [9].

We can have a mixed association rule with negative and hierarchical items. In many cases, for bottom items A and B, there is not a significant difference between conf(→ B) where the left-hand side is empty, and conf(-A → B), because support(-A) is close to the

entire set of transactions. If A is higher in the hierarchy, however, the sensitivity of –A could be higher.

2.6 Removal of redundant rules

Suppose we have a confident rule A → B. If purchases of A and B are nearly independent, this rule can not give us an effective estimation as to the behavior of purchasers of B. We call such rules redundant. We need to remove such redundant rules form the set of confident rules. As the concept of association rule is similar to conditional probability, we can use a similar strategy. Namely, if conf(→ B)=support(B) is close to conf(A → B), that is, support(A)support(B) is close to support(A, B), we discard A → B from the set of confident rules if it is there. In other word, A is not sensitive to the purchase of B, if A and B are nearly independent. We can perform stricter statistical tests on the independence of A and B as random variables.

3. Maximum Subarray Problem

The maximum subarray problem is to find the consecutive portion of an array that maximizes the sum of array elements in the portion. In most applications, one-dimensional and two-dimensional arrays are used.

Example 6. Suppose the following is the sales amount of beers for one year at a pub in some city, where units are in thousands, and the bottom numbers indicate months starting from January.

298	143	154	235	631	345	879	743	298	241	198	252
1	2	3	4	5	6	7	8	9	10	11	12

As all values are positive, the obvious solution is the whole array. If we subtract the mean value 368, we can have some meaningful trend as to which season is the most promising for the sale of beer as shown below.

–70	–225	–214	–133	263	–23	511	375	–70	–127	170	–116
1	2	3	4	5	6	7	8	9	10	11	12

The maximum subarray is given from May to August season as the most promising with the total sales amount of 631+345+879+743=2598 using the original array. By identifying the season, we can decide when to open and close a beer garden.

Example 7. We give an example for a two-dimensional array as shown below, where the mean value is subtracted from each array element. Co-ordinates are given by (x, y) where x is the row number and y is the column number.

```
0  3 -2  6  -3 -7  4  -2
3 -3 -5 -7  |3 -4  5|  2
3 -2  9 -8  |3  6 -5|  2
1 -3  5 -6  |8 -2  2| -6
```

Then the maximum subarray with the value of 16 is given by the rectangle defined by the upper left corner (2, 5) and the lower right corner (4, 7). If we regard this table as the sales amounts of some commodity, such as business suits, the rows and columns can be age groups and income levels. Then the maximum subarray given above may correspond to senior age groups and above-average income groups. If we regards the above table as pixel values of an image file, the maximum subarray correspond to a brightest part in the image.

This problem was first introduced by Bentley [2, 3] in 1984 as a programming example, and recently attracts attention from data mining point of view (see [19] and [21]).

3.1 Relationship between association rules and maximum subarrays

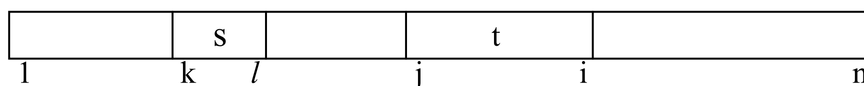
Suppose we have the attribute of purchasing time for each purchase of beer. Let us interpret the rule "A → B" as follows: if a purchase occurs at a time between month *l* and month r, the purchase is likely to include beer. Let M be the number of all items purchased during month 1 to month n. Let θ be the ratio of beer among purchased items. Then the maximum confidence for "A → B" is defined by

$$\max_{1 \le l \le r \le n} \left\{ \sum_{i=l}^{r} (a[i] - \theta (M/n)) \right\}$$

Example 6 is the problem such that purchased items are only beer, that is, $\theta = 1$. We can use the one-dimensional maximum subarray problem when the antecedent A is described by maximizing interval in a numerical attribute such as time.

If we have two numerical attributes, we can use the two-dimensional problem.

3.2 Kadane's algorithm. This algorithm scans the given one-dimensional array, accumulating a tentative sum in t, and if t>s for the current maximum s, s is updated by t. If t becomes negative, it is reset to 0. The variables k and *l* keep track of the beginning and ending positions of subarray whose sum is s. The situation is illustrated in the following figure.

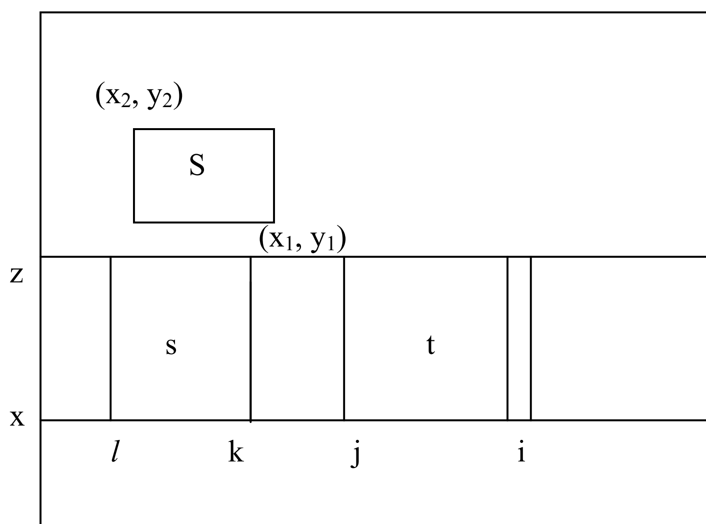


The algorithm follows. If all values are negative, we allow the empty subarray with s=0 for the solution. In this case (k,*l*)=(0,0) will not change.

```
(k,l)=(0,0); s=0; t=0; j=1;
for (i=1; i<=n; i++) {
  t=t+a[i];
  if (t>s) { (k,l)=(j,i); s=t; }
  if (t<0} then { t=0; j=i+1; }
}
```

As seen form this algorithm, the computing time is O(n), that is linear. This is optimal, because we need to look at every array element. Here if we say the time is O(f(n)), it is proportional to f(n) where f(n) is a function of input size n.

We can extend Kadane's algorithm into two dimensions. We perform the one–dimensional Kadane's algorithm for the strip defined by row x and row z as shown in Figure 2. The rectangle defined by (x_1, y_1) and (x_2, y_2) at the bottom-right and top-left corner is a tentative one for the solution, while the rectangle defined in the strip from *l* to k is a tentative one for this particular one-dimensional case defined by x and z. As we solve the one-dimensional Kadane by changing x and z, the computing time of this algorithm is $O(m^2n)$ in total. The value of column[x][i] is the sum of a[z .. x][i], that is, the sum of the i-th column of array a from row z to row x.




```
((x1,y1),(x2,y2))=((0,0),(0,0));
S=0;
for(z=1;z<=m;z++){
  /** initialize clolumn[][] **/
  for(i=1;i<=n;i++)column[z–1][i]=0;
   for(x=z;x<=m;x++){
      t=0; s=0; (k,l)=(0,0);
      j=1;
      for(i=1;i<=n;i++){
         column[x][i]=column[x][i–1]+a[x][i];
         t=t+column[x][i];
         if(t>s){s=t; (k,l)=(i,j); }
         if(t<0){t=0; j=i+1; }
      }
      if(s>S){S=s; x1=x; y1=k; x2=z; y2=l; }
   }
 }
```

3.3 Algorithm by prefix sum

The prefix sum of array a at position i, denoted by sum[i], is the sum of a[1], …, a[i]. The prefix sum array “sum” is computed in O(n) time as follows:

```
sum[0]=0;
for(i=1;i<=n;i++) sum[i]=sum[i–1]+a[i];
```

The prefix sum at position (i, j) of a two-dimensional array a is the sum of array portion a[1..i][1..j]. Using the data structure of “column” given above, we have the following algorithm for array ”sum” with O(mn) time, that is, linear time, in two dimensions..

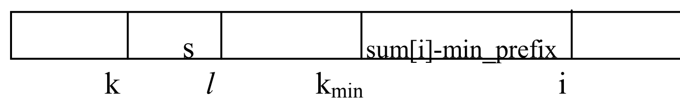
```
  /** initialize column[][] **/
  for(j=1;j<=n;i++)column[0][j]=0;
 /** main iteration **/
 for(i=1;i<=m;i++){
   for(j=1;j<=n;j++){
      column[i][j]=column[i–1][j]+a[i][j];
      sum[i][j]=sum[i][j–1]+column[i][j];
   }
 }
```

Using the concept of prefix sum, we can develop algorithms for the maximum subarray problem in the following.

Algorithm for one-dimensional case. In the following “min_prefix” is the minimum prefix sum at the end of iteration i, and “k_{min}” holds the position for it..

```
min_prefix=0; s=–999; k=0; l=0; k_min=0;
for(i=1;i<=n;i++){
   if(sum[i]–min_prefix>s) {s=sum[i]–min_prefix; l=i; k=k_min;}
   if(sum[i]<min_prefix) {min_prefx=sum[i]; k_min=i;}
}
```

The following figure illustrates the computation.



Algorithm for two-dimensional case. In the following we solve the one-dimensional problem repeatedly for the strip bounded by row z and row x similarly to the two-dimensional Kadane.

```
((x1,y1),(x2,y2))=((0,0),(0,0));
S=0;
```

```
for(z=1;z<=m;z++){
   for(x=z;x<=m;x++){
      t=0; s=0; (k,l)=(0,0); kmin=0; min_prefix=0;
      for(i=1;i<=n;i++){
         t=sum[x][i]–sum[z–1][i] –min_prefix;
         if(t>s){s=t; k=kmin; l=i; }
         if(sum[x][i]–sum[z–1][i]<min_prefix){
            min_prefix=sum[x][i]–sum[z–1][i];
            kmin=i;
         }
      }
      if(s>S){S=s; x1=x; y1=l; x2=z; y2=k+1; }
   }
 }
```

The algorithms based on prefix sums are not as efficient as Kadane's algorithms in terms of time and space, but they can be starting points for further speed-up and generalizations.

3.4 k-Maximum Subarray Problem

In many applications we need to find the maximum subarray, the second maximum, the third maximum, etc. down to the k-th maximum. For example, suppose the database is for a geographical distribution of customers, and we need to post flyers to the most loyal customers. The identified rectangle region for posting may not be very suitable due to road construction, etc. Then we need the second or third alternative. As usual, we start from the one-dimensional case.

The algorithm based on prefix sum is modified. The variable min_prefix is extended to a one-dimensional array of size k. Array portion min_prefix[1 .. k] holds k minima of sum[1], …, sum[i] at the end of iteration i in the following algorithm. Also the solution variable "s" is extended to an array of size k, s[1], …, s[k], representing the maximum, the second maximum, …, the k-th maximum. At the end of iteration i, s[1..k] represents the k-maxima for the array portion a[1..i].

In the following, "max" is to merge array s[1..k] and (sum[i]–min_prefix[1], …, sum[i]–min_prefix[k]), and take the first k elements. The function "insert" is to insert the element "sum[i]" into the sorted list of "min_prefix[1..k]". If is is smaller than "min_prefix[k]", it is abandoned. Obviously those two operations take O(k) time. Thus total time for the algorithm is O(kn). We omit the maintenance of positions for the possible k-maxima.

```
min_prefix[1..k]=(0, …, 0); s[1..k]=(0,…,0);
for(i=1;i<=n;i++){
   s[1..k]=max(s[1..k], sum[i]–min_prefix[1..k]);
   insert sum[i] into min_prefix[1..k];
}
```

We can easily extend the above algorithm into two dimensions, as we did for the ordinary maximum (1-maximum) subarray problem. This result is reported in [4] with time

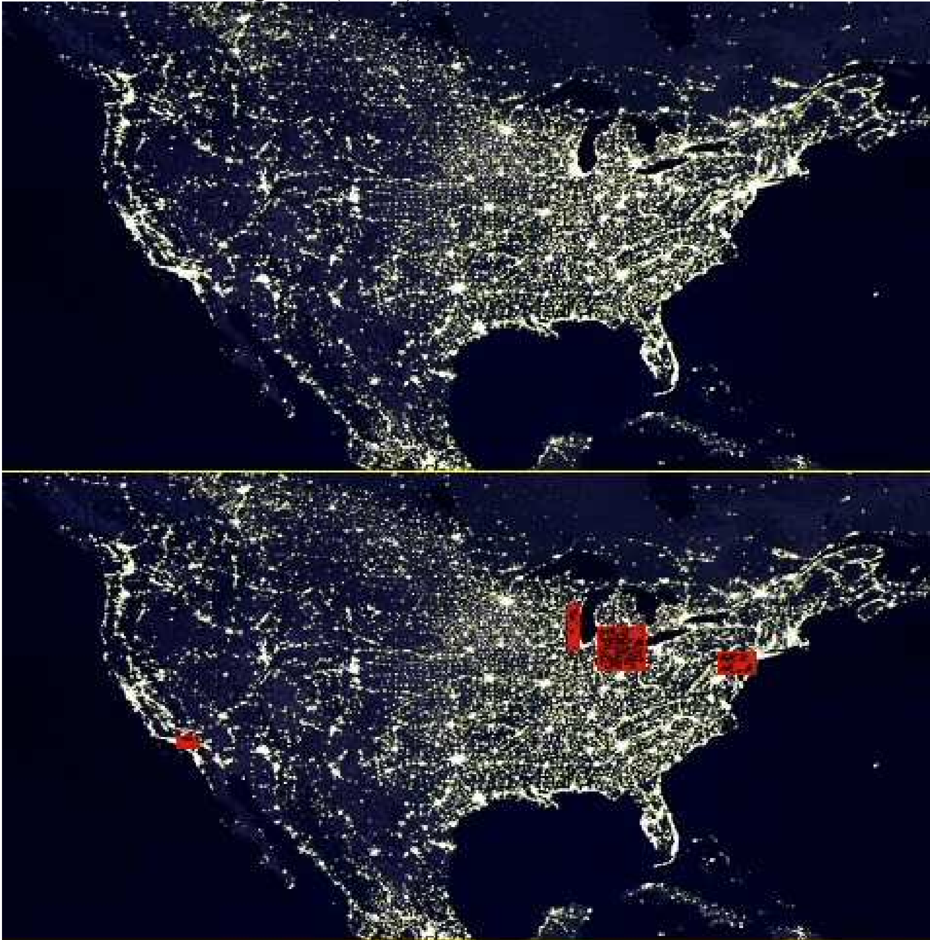
complexity of $O(km^2n)$ for an (m, n)-array. An efficient hardware implementation method is reported in the same literature.

In many applications, such as graphic images, the found k-maximum regions heavily overlap, that is, the algorithm finds k maxima by only changing co-ordinates slightly from the found portion. The following algorithm for the disjoint problem is often more useful.

3.5 k disjoint maximum subarray problem.

In this problem we find the maximum subarray, discard the found portion, find the next maximum subarray from the remaining portion, discard this portion, etc. up to the k-th maximum subarray. The algorithm can be implemented in the following way.

Use any maximum subarray algorithm. Give –999 (minus infinity) to the elements in the found portion. Apply the same algorithm for the updated array, etc. Obviously this approach takes O(kn) time for the one-dimensional case, and can be extended to two dimensions, resulting in $O(km^2n)$ time.



Night view of North America: courtesy of NASA

The above is a satellite view of the U.S.A. at night, and four brightest spots are given in the order of New York, Chicago, Michigan, and Los Angeles areas. The experiment was done by the disjoint version. If we allowed overlapping, k maximum subarrays for k

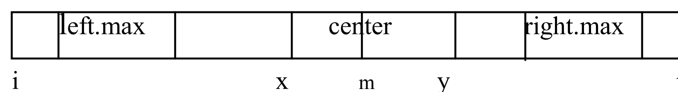
up to 3 or 4 would be centered in the New York area with slight difference in the co-ordinates.

3.6 Focused search for the maximum subarray.
In many applications, such as mono-chromatic images, the maximum subarray often corresponds to a large grey area, or a rectangle which includes several white objects, when we are looking for a brightest spot. One way to overcome this problem is to remove the constraint of the shape of rectangle, but if we impose just connectedness on the shape, the problem becomes NP-complete, meaning there is no efficient algorithm for solving the problem. If we have the restriction such that all cells vertically and horizontally are connected in the target region, there is an $O(n^3)$ time algorithm in [25].

As an alternative we introduce another idea of the repeated maximum subarray problem. Remember we subtract the mean value of the given array from each array element at the beginning. After we find the maximum subarray, we repeat the same process in the found subarray. That is, we subtract the mean of this subarray from each element of the subarray, and solve the maximum subarray on this subarray again. We repeat this process until we have some accuracy which appeals to our visual intuition. From our experiments, repetition of two or three times already gives us well focused search.

3.7 Subcubic time algorithm for two dimensions.
For a square array of size (n, n), we so far developed $O(n^3)$ time algorithms, which we call cubic. We show in this section that we can do better than cubic, that is, subcubic. The main weapon in this section is the idea of "divide-and-conquer", a major tool in algorithm design.

We start from the one-dimensional case. Divide the given array into two halves. We name the maximum subarray in the left half, right half, and the one stretching over the center by "left.max", "right.max", and "center". Then the solution must be the maximum of those three. To obtain "left" and "right", we go recursively. To obtain "center", we need the concept of the maximum prefix sum and minimum prefix sum, which we call "max_prefix" and "min_prefix". We use the struct type named "triple" which consists of three integers "max", "min_prefix", and "max_prefix". The recursive function returns those values obtained from interval [i, j]. Let us assume the prefix sum array "sum" is already available. The following picture illustrates the situation, where sum[x] is the min_prefix in the left half and sum[y] is the max_prefix in the right half



The algorithm follows.

```
1.      struct triple maxsubarray(int i, int j) {
2.        struct triple sol; int m, center;
3.        if(i==j){sol.max=sum[i]; sol.min_prefix=sum[i–1]; sol.max_prefix=sum[i];
4.          return sol;
5.        }
```

```
6.        else {
7.          m=(i+j–1)/2;
6.          left=maxysubarray(i,m);
8.          right=maxsubarray(m+1, j);
9.          center=right.max_prefix–left.min_prefix;
10.         sol.min_prefix=min(left.min_prefix, right.min_prefix);
11.         sol.max_prefix=max(left.max_prefix, right.max_prefix);
12.         sol.max=max(max(left.max, right.max), center);
13          return sol;
14.       }
15.    }
```

In the main program, we compute the prefix sum array "sum", and call "maxsubarray(1,n)". In lines 10-12, min(a, b) is the minimum of a and b, and max(a, b) is the maximum. At lines 4 and 13, the value of the solution values, minimum prefix sum, and maximum prefix sum for array portion sum[i..j] are brought back to the calling site in variable "sol". Variable "m" is to show the mid point. The key observation is that the value of "center" can be computed by right.max_prefix - left.min_prefix, and those minimum and maximum prefix sum values are brought through recursion.

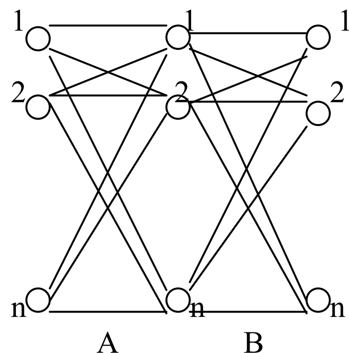
Distance matrix multiplication. Normally we multiply two (n, n) matrices over real numbers using "+" and "*". Let C=AB where A, B, and C are (n, n) matrices. Then

$$c_{ij} = \sum_{k=1}^{n} a_{ik}*b_{kj} \quad (i, j = 1, \ldots, n)$$

We can define distance matrix multiplication by corresponding the above "+" to "min" and "*" to "+" as follows:

$$c_{ij} = \min_{1 \le k \le n} \{a_{ik} + b_{kj}\} \quad (i, j = 1, \ldots, n)$$

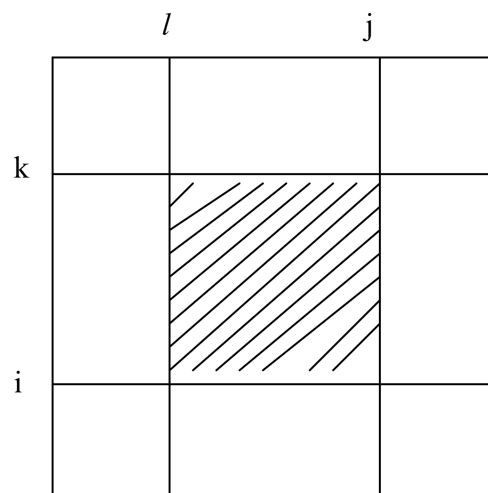
The intuitive meaning of c_{ij} is the distance of the shortest path from vertex i in the first layer to vertex j in the third layer in the following graph. The distance from i in the first layer to to j in the second is a_{ij} and that from the second layer to the third is b_{ij}.



In this figure, matrices A and B are to show the connection distance from layer to layer.

We can define the max version by changing the symbol “min” to “max” in the above formula. This corresponds to longest paths from layer 1 to layer 3. The original version is called the min version in this context.

Algorithm for two dimensions. We can solve the problem by calling the above algorithm for one dimension for the strip surrounded by the j-th row and i-th row for each i and j such that $i \leq j$. Then we hit the complexity of $O(m^2n)$ again. Here we use the idea of divide-and-conquer directly on two-dimensions. We divide the given (m, n) array into two equal rectangles at the vertical center line, and call them LEFT and RIGHT. Then the solution is the maximum of the one in LEFT, named A_{left}, the one in RIGHT, named A_{right}, and the one stretching over the center line, called A_{center}. We call the same procedure on LEFT and RIGHT recursively until we hit a bottom. For simplicity we assume the given array is a square of size (n, n). When we divide the array, we lose the shape of a square. We assume the subarray encountered in the algorithm is always horizontally rectangle, that is, m<=n. This is done by rotating the array 90 degrees when necessary. Then we hit the bottom with a single array element. The algorithm and a figure follow.



Main algorithm.
If the array becomes one element, return its value.
Otherwise, if m>n, rotate the array 90 degrees.
Thus we assume m<=n.
Let A_{left} is the solution in LEFT.
Let A_{right} is the solution in RIGHT.
Let A_{center} is the solution stretching over the center line.
Let the solution be the maximum of the three.

Now A_{center} can be computed in the following way.

$$A_{center} = \max_{k=1,l=0,i=1,j=n/2+1}^{i-1,\ n/2-1,\ m,\ n} \{s[i][j] - s[i][l] - s[k][j] + s[k][l]\}$$

In the above we first fix i and k, and maximize the above by changing l and j. Then the above problem is equivalent to maximizing the following for i=1, ..., m and k=1, ...,i–1.

$$A_{center}[i][k] = \max_{l=0,\ j=n/2+1}^{n/2-1,\ n} \{s[i][j] - s[i][l] - s[k][j] + s[k][l]\}$$

Let s*[i][j] = –s[j][i]. Then the above problem can further be converted into

$$A_{center}[i][k] = -\min_{l=0}^{n/2-1} \{s[i][l] + s^*[l][k] \quad + \max_{j=n/2+1}^{n} \{s[i][j] + s[j][k]\}$$

The first part in the above is distance matrix multiplication of the min version and the second part is of the max version. Let S_1 and S_2 be matrices whose (i,j) elements are s[i][j–1] and s[i][j+n/2]. For an arbitrary matrix T, let T* be that obtained by negating and transposing T. Then the above can be computed by multiplying S_1 and S_1* by the min version and taking the lower triangle, multiplying S_2 and S_2* by the max version and taking the lower triangle, and finally subtracting the former from the latter and taking the maximum from the resulting triangle.

Analysis. We start from an (n, n) array, and go through two levels of recursion, resulting in four recursive calls to (n/2, n/2) arrays. Let us assume that n is a power of 2. Let T(n) be the computing time for the (n, n) array. We observe the algorithm splits the array vertically and then horizontally. We can multiply (n, n/2) and (n/2, n) matrices by 4 multiplications of size (n/2, n/2). We analyze the number of comparisons. The rest is proportional to this. Let M(n) be the time for multiplying two (n/2, n/2) matrices. Then we have the following recurrence.

$$T(1) = 0$$
$$T(n) = 4T(n/2) + 12M(n).$$

If M(n) is greater than $O(n^2)$, we can prove that T(n) = O(M(n)), that is, the complexity of the algorithm by divide-and-conquer is of the same order as that of DMM. The sub-cubic complexity of $O(n^3(\log\log n)^2/\log n)$ was reported in [20], which has been improved to $O(n^3\log\log n/\log n)$ by the author. This section is mainly from theoretical interests to see how much we can improve the algorithm for the maximum subarray problem from cubic time of $O(n^3)$. For practical applications, the previous algorithms of $O(n^3)$ will work better.

4. Experimental results

Using the C program in list A, a sales database of a supermarket in Japan was analyzed. The size parameters and performance measurements are summarized in the following table.

Total number of transactions : 88303
Total number of different items : 459
Average number of items per transaction : 16.3

Experiment with minsup = 0.06, minconf = 0.7
Processing time for association rules : 1 min 28 sec
Among high confidence rule “Miso → Tofu” has conf=0.7
Time for one query given above : 0.65 sec

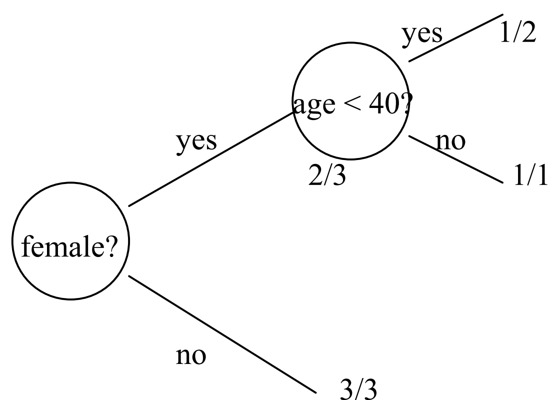
As the sales database is from a food supermarket, there were not unexpected rules mined. “miso” and “tofu” are main ingredients for making “miso soup” in the Japanese cooking.

Using the modified version of the C program in source list D, 4 disjoint brightest spots were detected as shown in the picture in the last section. The size is measured by the (241, 482) with RGB pixels. Those values were converted to grey scale values. The time was 4.9 seconds on a Linux machine on a Celeron processor with 2.2 GHz.

5. Concluding Remarks

In this chapter we discussed two representative areas in data mining; association rules and maximum subarray problem, and showed how to design algorithms to solve these problems. As data mining is a vast area, still growing fast with advent of increased capacity in hardware, more handling power in software, and increased bandwidth in the internet, it is hard to cover all areas in data mining in this chapter. The following is an overview of other areas that we did not cover. In the explanation, we use the same example of sales data given at the beginning.

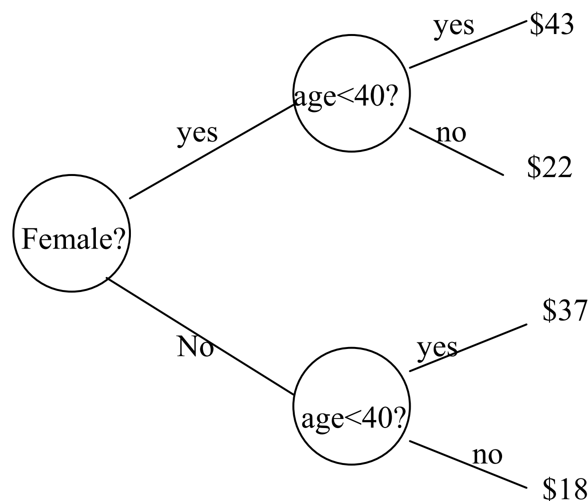
Decision tree/regression tree. Suppose we want to make a decision tree from the given sales database that can answer our question like “Is a customer who is female and age 40 or greater likely to buy milk?” A decision tree is a binary tree such that at each node we make a binary decision, and go down one of the two branches depending on the outcome of the question until we reach a leaf. In our example, we are guided to the “female branch”, and at the next node, we are guided to the branch with “no” for “age < 40”.



If the customer is female, the prediction that she buys milk is 2/3. If we ask one more question “is she under 40” and the answer in “no”, we can predict with 100%. If the

answer is “yes”, we can predict the chance she buys milk is 50%. As our example database is small, a new customer with the above profile (female, not younger than 40) may not actually buy milk. If we organize the tree properly based on a large realistic database, however, we will be able to make a more precise prediction. If the question at the second node is “Is she in suburb A?” we have 2/2 in the “yes” branch and 0/1 in the “no” branch.

In the above tree, the prediction is whether milk is purchased. If we ask the question how much in total would be spent by the customer, we can attach to each leaf the average value of the amount spent.



Using this tree we can predict a customer who is male and not younger than 40 will be spending about $18. If the predicted value at the leaves is a numerical value, the tree is called a regression tree. The database on which the tree is constructed is called the training set. The problem of how to organize the tree so that the prediction is accurate and the number of questions to ask is minimal is described in [23], [10], [6] and [15].

Clustering. Clustering is to classify data items into several categories so that items in each category are close to each other according to some criterion of closeness using some attribute values. Suppose we send advertisement brochures to customers. If we send brochures for daily items and luxurious items separately, we classify customers into two groups according to some closeness measure based on their age and annual income in our example. Then we may classify them into the junior/low income group {customer 1, customer 3} to which the former brochure is sent and the senior/high income group {customer 2, customer 4, customer 5, customer 6} for the latter brochure.

If we send brochures to customers by two delivery persons, we may classify them into two groups using their addresses plotted on a map, and actual distances as closeness measure. Then we may classify them into {customer 1, customer 2, customer 5} and {customer 3, customer 4, customer 6}.

In the above we classified data into two groups. In general, we classify them into k groups with various measures of closeness. Several methods are described in [11], [23], [10], and [13].

Text mining. The above mentioned areas are the four representative areas of data mining. As a neighboring area, we have the area called "text mining". In this area, we search for a pattern from a large text database. This area is attracting attention recently in regard to web search, where the databases from which we extract information are scattered through the internet. With a few key words, we want to extract useful information from various web sites. Another area for text mining is in bioinformatics, where a genome database is searched for a gene pattern. The technical methods used are scattered in textbooks such as information retrieval [22], [5], and pattern/text matching [7], [8], in which similarity measures on strings are described. Pattern matching with some errors allowed is called approximate pattern matching, which is particularly important in web search. Genome analysis is described in [8], [12], and similarity measures on two dimensional images are discussed in [16].

Apart from those areas, the following issues need to be addressed for practical developments.

Parallel/distributed processing. As the amount of data processed is large, we may be able to speed up the processing time using many computers that co-operate with each other. Also data may be gathered from various sites through the internet. Various techniques developed in parallel/distributed computing should be used in data mining.

Privacy/security. The database normally includes sensitive data of clients that must be protected from leakage. This is especially so in medical databases. The leakage can occur in the data acquisition phase through the network, or unauthorized access to the database. Control of pass words and data encryption and authentication need to be done. If data mining is going to be accepted by the general society, the issue of privacy must be addressed first.

User interface. Useful information extracted from the database must be shown for human consumption, using some visual method. For an example of text mining, suppose the scripts of conversations in a meeting are analyzed. We can make a directed graph that shows an arrow from member A to member B, if A quotes what B said, with label of the number of quotation times. From the directed graph, we can visualize who was influential in the meeting.

As is seen from the above observations, data mining is a very interdisciplinary area, and is still growing. We will need to investigate more systematic approach in the future.

Acknowledgement

The author is very thankful to two students Bae Sung Eun and Kiyoyuki Nakagaki, who implemented and tested the C programs given in this chapter. He is also grateful to anonymous referees for the chapter, whose constructive comments greatly improved the description of the chapter.

Appendix -- Program source lists

We provide the source lists of C programs for algorithms used in this chapter.

From the association rule section, we list two source programs. The first is the extraction of association rules with minimum support and confidence in source list A. The C program will prompt you to give several parameters. The size of the database measured by the number of transactions is stored in variable "num_transaction", and the number of all possible different items the supermarket in concern can sell in variable "num_items" . User is also prompted to give the minimum support and minimum confidence by variables "minsup" and "minconf". The program is designed to run with random numbers. For practical databases, the portion commented out in the main function should be invoked, and the portion using the random number generator should be commented out. The database file is supposed to consists of up to NUM_TRANSACTION lines. Each line consists of a number of items given by integers. Thus the database is supposed to be prepared from the original sales database in such a way that each item is converted to a unique integer. We assume there are NUM_ITEMS items that the super-market sells.

We also provide a program which can handle a specific inquiry to the database. A typical enquiry is "A → B?". This enquiry can give you the support of this rule and its confidence. This is given in source list B. The actual enquiry form will be "A > B".

For the two-dimensional maximum subarray problem, we list two C programs. The first is the two-dimensional Kadane's algorithm in source list C. The other is based on the prefix sum approach in source list D. Both are run on random numbers in the range of 0 - 99. Each value is reduced by 50, rather than the mean value. The user is prompted to give the value of "m" and "n", which define the size of the given two-dimensional array.

These source programs are available at http://cosc.canterbury.ac.nz/~tad/mining/.

Source list A

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <time.h>
#define NUM_ITEMS 228
#define NUM_TRANSACTION 88303
#define NO_NODE -1
#define ROOT -2

struct node{
    int item,count,depth;
    struct node *parent,*children;
};
struct tuple{
    int count;
    int items[NUM_ITEMS];
};

struct tuple D[NUM_TRANSACTION]; /* Transactions */
struct tuple I;      /* Itemsets */
struct node Ck;      /* Set of Candidate tuple in a tree form*/
struct node Lk;      /* St of frequent tuple in a tree form*/
struct node *C[NUM_TRANSACTION]; /* Sets appearing in a transaction */
int k,M,num_items,num_transaction,num_sets,C_size,minsup;
float minconf;
int itemset[NUM_ITEMS]; /* Space for an item set as an array */
```

```
/* traverses a tree to find an item set that appears in the given transaction*/
void traverse_tree(struct node *n, struct tuple *t, int i){
    int j;
    if(n->depth<k){  /* if not leaf node */
        for(j=i;j<t->count;j++) /* exhaust children of Ck*/
            if(n->children[t->items[j]].item!=NO_NODE)
                traverse_tree( &(n->children[t->items[j]]), t, j);
    }else     /* leaf node */
        C[C_size++]=n;  /* Put it in C */
}
void initialize_node(struct node *parent, int item){
    int i=0;
    struct node *n=&(parent->children[item]);
    n->item=item; n->count=0; n->depth=parent->depth+1;
    assert((n->children=malloc((NUM_ITEMS+1)*sizeof(struct node)))!=NULL);
    for(i=0;i<NUM_ITEMS+1;i++){
        n->children[i].item=NO_NODE;
        n->children[i].parent=n;
    }
}
void initialize_root(struct node * root){
    int i;
    root->item=ROOT; root->depth=0;
    assert((root->children=malloc((NUM_ITEMS+1)*sizeof(struct node)))!=NULL);
    for(i=0;i<NUM_ITEMS+1;i++){
        root->children[i].item=NO_NODE;
        root->children[i].parent=root;
    }
}
/* destroys a tree freeing memory */
void dispose_tree(struct node *n){
    int i=0;
    for(i=0;i<NUM_ITEMS;i++)
        if(n->children[i].item!=NO_NODE)
            dispose_tree(&(n->children[i]));
    n->item=NO_NODE;
    free(n->children);
}
/* returns an itemset the given node n represents */
void item_set(struct node *n, int *itemset){
    int temp[NUM_ITEMS],temp_size=0,i;
    while(n->item!=ROOT){
        temp[temp_size++]=n->item;
        n=n->parent;
    }
    for(i=0;i<temp_size;i++)
        itemset[i]=temp[temp_size-i-1];
}

/* extracts from Ck the item set that has larger count than the given minimum count and
puts it into Lk*/
void extract_common(struct node *c, struct node *l){
    int i;
    if(c->depth==k){ /* If leaf node */
        if(c->count>=minsup){  /* if c.count>=minsup, copy the node */
            initialize_node(l->parent,c->item);
            l->count=c->count;
        }
    }else   /* otherwise, exhaust children of Ck and Lk */
        for(i=0;i<NUM_ITEMS+1;i++)
            if(c->children[i].item!=NO_NODE)
                extract_common(&(c->children[i]),&(l->children[i]));
}

/*1) concatenation phase*/
void concatenation_phase(struct node *C, struct node *L,int depth){
    int i,j;
    if(L->depth==depth-1){ /* node for prefix_k-1(p)=prefix_k-1(q) */
        for(i=0;i<NUM_ITEMS+1;i++){           /* for each p(k) */
            if(L->children[i].item!=NO_NODE){
                for(j=i+1;j<NUM_ITEMS+1;j++){   /* for each q(k) */
```

```
                    if(L->children[j].item!=NO_NODE){
                        /* 4) Let c be (k+1)-tuple of p followed by the last item of q*/
                        /* 5) Add c to Ck+1;*/
                        initialize_node(&(C->children[i]),L->children[j].item);
                        num_sets++;
                    }
                }
            }
        }
    //Otherwise traverse further downwards
    }else
        for(i=0;i<NUM_ITEMS+1;i++)
            if(L->children[i].item!=NO_NODE)
                concatenation_phase(&(C->children[i]),&(L->children[i]),depth);

}
/* 7) removal phase of unneccessary tuple*/
void removal_phase(struct node *C, struct node *L,int depth){
    int i,j;
    struct node *n;
    if(C->depth==depth+1){  /* 8) For each p in Ck+1{ */
        struct node *n;
        int s[NUM_ITEMS];
        /* 9) For  each k-tuple s made by removing one item from p{ */
        item_set(C,s);
        for(i=0;i<depth+1;i++){
            n=L;
            for(j=0;j<depth+1;j++){
                if(i!=j){
                    /* 10) If s is not in Lk, delete p from Ck+1*/
                    if(n->children[s[j]].item==NO_NODE){
                        num_sets--;
                        C->item=NO_NODE;
                        free(C->children);
                        return;
                    }else
                        n=&(n->children[s[j]]);
                }
            }
        }
    }else /* Otherwise traverse further downwards */
        for(i=0;i<NUM_ITEMS+1;i++)
            if(C->children[i].item!=NO_NODE)
                removal_phase(&(C->children[i]),L,depth);
}
void apriori_gen(struct node *C,struct node *L,int depth){
    num_sets=0;
    concatenation_phase(C,L,depth);
    removal_phase(C,L,depth);
}

/* conf calculates a confidence level for A>B by sup(A,B)/sup(A)*/
float conf(int *ant,int ant_size,int *con, int con_size){
    int sup_ac,sup_a,i=0,j=0;
    struct node *temp = &Lk;
    while(i+j<ant_size+con_size){  /* denominator sup(A) */
        if((i<ant_size && ant[i]<con[j]) || j == con_size)
            temp=&(temp->children[ant[i++]]);
        else
            temp=&(temp->children[con[j++]]);
    }
    sup_ac=(*temp).count;
    temp = &Lk;     /* numerator sup(A,B) */
    for(i=0;i<ant_size;i++)
        temp=&(temp->children[ant[i]]);
    sup_a=(*temp).count;
    return (float)sup_ac/sup_a; /* numerator / denominator */
}

/* visit_calcConf traverses a tree calculating confidence level*/
void visit_conf(int *lk,int lk_size,struct node *hm1,int m){
```

```
    int i=0;
    if(hm1->depth==m+1){ /* 8) for each h in Hm+1 */
        int j=0,h,ant_size=0,con_size=0,ant[NUM_ITEMS],con[NUM_ITEMS],conset[NUM_ITEMS];
        float confidence;
        item_set(hm1,conset);
        while(ant_size+con_size<lk_size){
            if(lk[i]!=conset[j])
                ant[ant_size++]=lk[i];
            else{
                con[con_size++]=lk[i];
                j++;
            }
            i++;
        }
        confidence=conf(ant,ant_size,con,con_size); /* 9) conf=support(w)/support(w-h); */
        if(confidence>=minconf){    /* 10) if(conf>=minconf) output(w-h -> h); */
            printf("%f | ",confidence);
            for(i=0;i<ant_size;i++)
                printf("%d ",ant[i]);
            printf("=> ");
            for(i=0;i<con_size;i++)
                printf("%d ",con[i]);
            printf("\n");
        }else{  /* 11) Hm+1 = Hm+1 - h; */
            hm1->item=NO_NODE;
            free(hm1->children);
        }
    }else
        for(i=0;i<NUM_ITEMS+1;i++)
            if(hm1->children[i].item!=NO_NODE)
                visit_conf(lk,lk_size,&(hm1->children[i]),m);
}

void association_rules(struct node *L){
    int i,h,m;
    if(L->depth>=2){   /* 1) for(k=2,...,M){ */ /* 2) for(each frequent tuple w inLk){ */
        float confidence;
        int ant[NUM_ITEMS],con[1],ant_size,w[NUM_ITEMS]; /* ant-antecedent,con-consequence
*/
        struct node Hm;
        initialize_root(&Hm); /*3) H1 = {h in w | conf(w-h -> h) >= minconf} */
        item_set(L,w);
        for(h=0;h<L->depth;h++){
            ant_size=0;
            for(i=0;i<L->depth;i++){
                if(i!=h)
                    ant[ant_size++]=w[i];
                else
                    con[0]=w[i];
            }
            confidence=conf(ant,ant_size,con,1);
            if(confidence>=minconf){
                initialize_node(&Hm,con[0]);
                printf("%f | ",confidence);
                for(i=0;i<ant_size;i++)
                    printf("%d ",ant[i]);
                printf("=> %d\n",con[0]);
            }
        }
        m=1;  /* 5) m=1; */
        while(m<=L->depth-2){  /* 6) while m <= k-2){ */
            apriori_gen(&Hm,&Hm,m); /* 7) Hm+1 = apriori_gen(Hm); */
            visit_conf(w,L->depth,&Hm,m); /* 8-12 */
            m=m+1;  /* 13) m=m+1; */
        }
        dispose_tree(&Hm);
    }
    for(i=0;i<NUM_ITEMS+1;i++)
        if((*L).children[i].item!=NO_NODE)
            association_rules(&(L->children[i]));
}
```

```
void tuple(struct node *C,struct tuple *t){
    C_size=0;
    traverse_tree(C,t,0);
}

/* apriori function generates a set of frequent item tuple */
void apriori(){
    int i,j;
    initialize_root(&Ck); /*initializing the root of Ck & Lk*/
    initialize_root(&Lk);

    for(i=0;i<NUM_ITEMS;i++)   /* 1) C1:={(i) | i in I};*/
       initialize_node(&Ck,I.items[i]);
    k=1;   /*2) k=1;*/
    num_sets=1;
    while(num_sets>0){  /*3) while(Ck not empty){ */
       struct tuple *t;               //Temporary transaction
       struct node *n;
       int c;
       for(i=0;i<num_transaction;i++){  /*4) for (all transactions t in D){ */
           t=&(D[i]);
           tuple(&Ck,t);   /*5)C=tuple(Ck,t);*/
           for(c=0;c<C_size;c++) /*)6 for (all c<-C) c.count++;*/
              C[c]->count++;
       }
       extract_common(&Ck,&Lk); /*8) Lk:={c in Ck|c.count>=minsup}; */
       fprintf(stderr,"L[%d] generated \n",k);
       apriori_gen(&Ck,&Lk,k); /*9) Apriori-Gen(Lk);*/
       M=k;k=k+1    ;/*10) k++; */
    }
}

int main(int argc,char *argv[]){
    int i,j,num;
    float sup=0.5,probability[NUM_ITEMS];
    char line[10001],*buffer;
    FILE *fp = fopen("transaction.txt","r");
    for(i=0;i<NUM_ITEMS;i++) /* Initializing items */
       I.items[i]=i+1;
    I.count=NUM_ITEMS;
    fprintf(stderr,"Input (Number of items) (Number of transactions) :\n");
    scanf("%d %d",&num_items,&num_transaction);
    srand((unsigned int)time(NULL));
    for(i=0;i<num_items;i++)
       probability[i]=0.8*rand()/RAND_MAX;
    for(i=0;i<num_transaction;i++){
       D[i].count=0;
       num=rand();
       for(j=0;j<num_items;j++){
           if((float)rand()/RAND_MAX<probability[j] && num>rand()){
              D[i].items[D[i].count++]=j+1;
              fprintf(stderr,"%d ",j+1);
           }
       }
       if(D[i].count==0){
              D[i].items[D[i].count++]=rand()%num_items+1;
              fprintf(stderr,"%d ",j+1);
       }
       fprintf(stderr,"\n");
    }
//    fprintf(stderr,"Transactions\n");/* loading transactions */
//    for(i=0;(fgets(line,10000,fp)!=NULL) && (i<NUM_TRANSACTION);i++){
//      buffer=strtok(line," ");
//      while(buffer!=NULL){
//          D[i].items[D[i].count++]=atoi(buffer);
//          buffer=strtok(NULL," ");
//      }
//   }
//    close(fp);
//    num_transaction=i;
```

```
    fprintf(stderr,"Input minsup : ");
    scanf("%f",&sup);
    minsup=(int)(i*sup);
    fprintf(stderr,"absolute minsup = %d\n",minsup);
    apriori();
    fprintf(stderr,"Input minconf : ");
    scanf("%f",&minconf);
    association_rules(&Lk);
}
```

Source list B

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define NUM_TRANSACTION 88303

struct tuple{
    int count,items[500];
};
struct tuple D[NUM_TRANSACTION];       /*Transactions*/
int num_transaction,num_ant,num_con,num_antUcon,sup_antUcon,sup_ant;
int ant[10],con[10],antUcon[20];

void load_transactions(void){
    int i;
    char line[10001],*buffer;
    FILE *fp = fopen("transaction.txt","r");
    printf("Loading transactions...\n");
    for(i=0;(fgets(line,10000,fp)!=NULL) && (i<NUM_TRANSACTION);i++){
       buffer=strtok(line," ");
       while(buffer!=NULL){
           D[i].items[D[i].count++]=atoi(buffer);
           buffer=strtok(NULL," ");
       }
    }
    num_transaction=i;
}
int included(struct tuple *t, int *set,int size){
    int i,j,flag=1,sub_flag;
    for(i=0;i<size;i++){ /*Checks if the set is in transaction*/
       sub_flag=set[i]<0;
       for(j=0;j<(*t).count;j++)
           if(abs(set[i])==(*t).items[j])
              sub_flag=set[i]>0;
       if(!sub_flag) /* item does not exist */
           flag=0;
    }
    return flag; /* returns whether all the items exit or not*/
}
int calc_support(int *set, int size){
    int sup=0,i; /* Counts the support of the given set */
    for(i=0;i<num_transaction;i++)
       if(included(&(D[i]),set,size))
           sup++;
    return sup;
}
void show(int antUcon_sup,int ant_sup){
    printf("Support is : %d/%d\n",antUcon_sup,ant_sup);
    printf("Confidence level is : %f\n",(float)antUcon_sup/ant_sup);
}
void process_input(void){
    char line[201],*letter;
    int isAnt;
    num_ant=0; num_con=0; num_antUcon=0; isAnt=1;
    printf("Please input the rule : ");
    fgets(line,200,stdin);
    letter = strtok(line," ");
    if(*letter=='\n') /* no input exit */
       exit(0);
```

```
    while(letter!=NULL){ /* reading in input */
        if(*letter=='>')
            isAnt=0;
        else if(isAnt){ /* into antecedent */
            ant[num_ant++]=atoi(letter);
            antUcon[num_antUcon++]=atoi(letter);
        }else           /* into consequence */
            antUcon[num_antUcon++]=atoi(letter);
        letter = strtok(NULL," ");
    }
    show(calc_support(antUcon,num_antUcon),calc_support(ant,num_ant));
}
int main(void){
    load_transactions();
    while(1)
        process_input();
    return 0;
}
```

Source list C

```
int i,j,k,l,m,n,x,z,x,x1,x2,y1,y2,s,t,S;
int a[100][100];
int column[100][100];
main(){
scanf("%d %d",&m,&n); getchar();
for(i=1;i<=m;i++) for(j=1;j<=n;j++) a[i][j]=random()%100-50;
for(i=1;i<=m;i++){
  for(j=1;j<=n;j++) printf("%4d ",a[i][j]);
  printf("\n");
}
x1=0; x2=0; y1=0; y2=0;
S=-9999;
for(z=1;z<=m;z++){
  for(i=1;i<=n;i++)column[z-1][i]=0;
  for(x=z;x<=m;x++){
    t=0; s=-9999; k=0; l=0;
    j=1;
    for(i=1;i<=n;i++){
      column[x][i]=column[x-1][i]+a[x][i];
      t=t+column[x][i];
      if(t>s){s=t; k=i; l=j;}
      if(t<0){t=0; j=i+1;}
    } /* i */
    if(s>S){S=s; x1=x; y1=k; x2=z; y2=l;}
//    printf("x1,y1,x2,y2 S %d %d %d %d %d\n", x1,y1,x2,y2,S);
  } /* x */
    printf("\n");
} /* z */
printf("x1,y1,x2,y2 S %d %d %d %d %d\n", x1,y1,x2,y2,S);
}
```

Source list D

```
int i,j,k,l,m,n,kmin,x,z,x,x1,x2,y1,y2,s,t,S;
int min_prefix;
int a[100][100];
int sum[100][100];
int column[100][100];
main(){
scanf("%d %d",&m,&n); getchar();
for(i=1;i<=m;i++) for(j=1;j<=n;j++) a[i][j]=random()%100-50;
printf("original data\n");
for(i=1;i<=m;i++){
  for(j=1;j<=n;j++) printf("%4d ",a[i][j]);
  printf("\n");
}
  for(j=1;j<=n;j++)column[0][j]=0;
```

```
  for(i=1;i<=m;i++){
    for(j=1;j<=n;j++){
      column[i][j]=column[i-1][j]+a[i][j];
      sum[i][j]=sum[i][j-1]+column[i][j];
    } /* j */
  } /* i */
printf("prefix sum\n");
for(i=1;i<=m;i++){
  for(j=1;j<=n;j++) printf("%4d ",sum[i][j]);
  printf("\n");
}

x1=0; x2=0; y1=0; y2=0;
S=-9999;
for(z=1;z<=m;z++){
  for(i=1;i<=n;i++)column[z-1][i]=0;
  for(x=z;x<=m;x++){
    t=0; s=-9999; k=0; l=0; kmin=0; min_prefix=0;
    for(i=1;i<=n;i++){
      t=sum[x][i]-sum[z-1][i]-min_prefix;
      if(t>s){s=t; k=kmin; l=i;}
      if(sum[x][i]-sum[z-1][i]<min_prefix){min_prefix=sum[x][i]-sum[z-1][i];
                                           kmin=i;}
    } /* i */
    if(s>S){S=s; x1=x; y1=l; x2=z; y2=k+1;}
  } /* x */
} /* z */
printf("x1,y1,x2,y2 S = %d %d %d %d %d\n", x1,y1,x2,y2,S);
}
```

References

[1] Agrawal, et. al. Mining Association Rules between Sets of Items in Large Databases, Proceedings of the 1993 ACM SIGMOD Conference.

[2] Bentley J., Programming Pearls – Algorithm Design Techniques, Communications of the ACM, 27, pp 865−871, 1984

[3] Bentley J., Programming Pearls – Perspective on Performance, Communications of the ACM, 27, pp 1087−1092, 1984

[4] Bae, S and T. Takaoka, Algorithms for the problem of K maximum sums and a VLSI algorithm for the K maximum subarrays problem, in proceedings of the 7th International Symposium on parallel architectures, algorithms & networks(ISPAN 2004), IEEE, pp.247—253, 2004

[5] Baeza-Yates, R. and B. Ribeiro-Neto, Modern Information Retrieval, Addison-Wesley, 1999

[6] Berry, M. J. A., and Gordon S. Linoff, Data Mining Techniques, John Wiley Sons, 2004

[7] Crochemore, Maxime, Text Algorithms, Oxford University Press, 1994

[8] Gusfield, D., Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology, Cambridge University Press, 1997

[9] J. Han and Y. Fu, Discovery of multiple-level association rules from large databases, Proceedings of the International Conference on Very Large Databases, pp 420-431, 1995

[10] Han, J. and Kamber, M., Data Mining : Concepts and Techniques, Academic Press, 2000

[11] Hand, D., H. Mannila, and P. Smyth, Principles of Data Mining,
MIT Press, 2001

[12] Jones, N. C. and P. A. Pevzner, An Introduction to Bioinformatics Algorithms (Computational Molecular Biology), Bradford Books, 2004

[13] Kaufman, L and Rousseeuw, P. J., Finding Groups in Data : An Introduction to Cluster Analysis, John Wiley and Sons, 1990

[14] Mannila, H., Toivonen, and A. I. Verkamo, Efficient Algorithms for Discovering Association Rules, Knowlwdgw Discovery in Databases: Papers from the AAAI -94 Workshop (KDD 94), AAAI Press, pp 181-192, 1994

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

[16] Shapiro, L. and G. Stockman, Conputer Vision, Prentice-Hall, 2001

[17] Srikant, R. and R. Agrawal, Mining Quantitative Association Rules in Large Relational Tables, Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data

[18] R, Srikant and R Agarwal, Mining generalized association rules, Proceedings of the International Conference on Very Large Databases, pp 407-419, 1995

[19] Takaoka, T., Efficient Algorithms for the Maximum Subarray Problem, in Electronic Notes in Theoretical Computer Science, vol (2002)

[20] Takaoka, T., A Faster Algorithm for the All–Pairs Shortest Path Problem and its Application, Proceeding of the 10th International Conference on Computing and Combinatorics, COCOON 2004, Lecture Notes in Computer Science 3106, pp 278–289 (2004)

[21] Tamaki, H. and T. Tokuyama, Algorithms for the Maximum Subarray Problem Based on Matrix Multiplication, Proceedings of the ninth annual ACM–SIAM Symposium on Discrete Algorithms, Society of Industrial and Applied Mathematics, pp 446–452, 1998.

[22] Witten, Ian, Alistair Moffat, and Tim Bell, Managing Gigabytes, Morgan Kaufmann Publishing, 1999

[23] Witten, I. and Frank, E., Data Mining : Practical Machine Learning Tools and Techniques with Java Implementations, Morgan Kaufman, 1999

[24] Wu, X. D., C, Zhang, and S. Zhang, Efficient mining of both positive and negative association rules, ACM Transactions on Information Systems, vol. 22, no. 381-405, July 2004

[25] Yoda, K., Fukuda, T., Morimoto, Y., Morishita, S., and Tokuyama, T., Computing optimized rectilinear regions for association rules, Proceedings of International Conference on Knowledge Discovery and Data Mining, 1997