5th Pacific Rim International Conference on Artificial Intelligence (PRICAI-98),
Nov 1998, pp. 182-193.

Using Decision Tree Induction for Discovering Holes in Data

Bing Liu, Ke Wang, Lai-Fun Mun and Xin-Zhi Qi

School of Computing
National University of Singapore
Lower Kent Ridge Road, Singapore 119260

{liub, wangk}@comp.nus.edu.sg
http://www.comp.nus.edu.sg/{~liub, ~wangk}

Abstract. Existing research in machine learning and data mining has been focused on finding rules or regularities among the data cases. Recently, it was shown that those associations that are missing in data may also be interesting. These missing associations are the holes or empty regions. The existing algorithm for discovering holes has a number of shortcomings. It requires each hole to contain no data point, which is too restrictive for many real-life applications. It also has a very high complexity, and produces a huge number of holes. Additionally, the algorithm only works in a continuous space, and does not allow any discrete/nominal attribute. These drawbacks limit its applications. In this paper, we propose a novel approach to overcome these shortcomings. This approach transforms the holes-discovery problem into a supervised learning task, and then uses the decision tree induction technique for discovering holes in data.

1. Introduction

A company wants to profile its customers who use its two services, S_1 and S_2. It used a rule induction system for the purpose. One of the generated rules is:

If *mthly_salary* > 3000 **Then** S_1.

The rule says that if a person's monthly salary is more than $3,000, he/she uses service S_1. The information contained in this rule is, however, rather incomplete and misleading. A close inspection of the database shows that very few professionals whose monthly salary is more than $3,000 use the service, and there is no customer with a monthly income of over $6,000 using S_1. These are "holes" in the data. Realizing these holes, the company can probe into the possibilities of improving its service or of doing more promotion in order to attract these potential customers. Clearly, only giving the above rule to the user does not provide him/her the complete knowledge.

A similar situation is that a company wants to launch a marketing campaign to enlarge its customer base. In order to be effective, it needs to have a good profile of those people who are not currently their customers. However, the company only has a database about its existing customers, but none on those non-customers (this is often the case in businesses). Then, the company cannot use an induction tool to profile non-customers because there is no data on them. This paper tries to solve this problem. It is essentially the problem of looking for holes in the database.

In general, each case in the database can be seen as a point in a multi-dimensional space. A hole is a region in the space that contains no or few data points. In a continuous space, there always exist a large number of holes. The existence of large holes is mainly due to the following two reasons: (1) insufficient data in certain areas; (2) certain attribute-value combinations are not possible or seldom occur. Clearly, not all holes are interesting. For example, holes that represent common knowledge in a domain are not interesting. However, those holes that we do not know before can be very important. For example, in a disease database we may unexpectedly find that certain symptoms and/or test values do not occur together, or when a certain medicine is used, some test values never go beyond certain range. Discovery of such information can be of great importance in medical domains because it could mean the discovery of a cure to a disease or some biological laws.

[7] first studied the problem of discovering holes. It proposed an algorithm for finding holes in a continuous space. The algorithm has a number of shortcomings:

1. A hole must be absolutely empty, i.e., containing no data point in it. This is too restrictive for most real-life applications because there are usually outliers.
2. The complexity of the algorithm is very high, which is $O(n^{2k-1}k^3(\log n)^2)$, where n is the number of data points and k is the number of dimensions. The number of holes produced is also very large.
3. It only works in a continuous space. It does not handle discrete attributes.

This paper proposes a novel approach to overcome these problems. The approach first transforms the holes-discovery problem into a supervised learning task, and then uses the decision tree technique to carve the space into filled and empty regions. The existing algorithm in [7] is also generalized so that it can take filled hyper-rectangles as input in addition to data points. It is used as a post-processing method in the proposed approach to produce maximal holes (which cannot be expanded further) based on the filled regions generated by the decision tree engine.

2. Problem Statement

Let D be a database of N cases, and let $A_1, ..., A_i, ..., A_m$ be the attributes. Some attributes take continuous values, and other attributes take discrete values. For a continuous attribute A_i, we use min_i and max_i to denote the bounding (minimum and maximum) values. For a discrete attribute A_i, we use a_i to denote its domain. min_i, max_i and a_i are provided by the user. The syntax of a hole has the following form:

$$<p_1, \ldots, p_i, \ldots, p_q>$$

where $1 \leq q \leq m$, and p_i is either

1. $(A_{i_j} = v_{i_j})$ with $v_{i_j} \in a_{i_j}$ if A_{i_j} is a discrete attribute, or
2. $(A_{i_j}, l_{i_j}, u_{i_j})$ with $min_{i_j} \leq l_{i_j} \leq u_{i_j} \leq max_{i_j}$ if A_{i_j} is a continuous attribute.

$A_{i_j} \in \{A_1, ..., A_i, ..., A_m\}$, and an attribute can only appear once in the expression.

Below we give the semantic definitions of a *hole* in three types of space.

Discrete attributes only: In this case, we call $<p_1, \ldots, p_i, \ldots, p_q>$, where p_i is only $(A_{i_j} = v_{i_j})$, a combination of attribute-value pairs. A *hole* is a combination of attribute-value pairs that has a *support* in D less than the user-specified *threshold*. A combination has a support s in D if s% of the data cases in D contain the combination. In this work we are interested in those holes that are maximal. A hole is *maximal* if any removal of attribute-value pair(s) will not result in a hole.

In the discrete case, the problem of discovering all the maximal holes can be solved by adapting an association rule discovery algorithm, e.g., the Apriori algorithm [1]. This algorithm generates all the itemsets (a set of items or attribute-value pairs) that satisfy the user-specified support threshold. In the process, it drops off those itemsets that do not meet the support requirement. The dropped-off itemsets are actually the maximal holes. Although useful in the discrete case, this method cannot be applied to continuous or mixed attribute cases.

Continuous attributes only: In this case, we define a *hole* as follows:

1. We first partition the whole continuous space into small hyper-rectangles, called *cell*s. A cell is considered *filled* if its density is above a density *threshold*, otherwise it is considered *empty*.
2. A *hole* (or an *empty region*) is a sub-space in the space that consists of only connected empty cells, i.e., everywhere empty. Likewise, a *filled region* is a sub-space that consists of only connected filled cells.

The size of the cell and the choice of density threshold value depend on applications.

In this paper, we restrict the shape of a hole to hyper-rectangles. A hole is *maximal* if it is bounded on each side by some *filled* regions or a boundary of the original space.

In the continuous case, too small holes are not useful. We define a *sufficiently large* hole to be a hole that is larger than the user-specified *minimum size*. A simple way of measuring the size of a hole is by its length along each dimension.

A mixture of continuous and discrete attributes: Without loss of generality, we assume that the first k attributes are discrete attributes, A_1, ..., A_k, and the rest are continuous attributes, A_{k+1}, ..., A_m. A *cell* description is as follows:

(*discr-cell, cont-cell*)

where *discr-cell* is a cell in the discrete sub-space, which is $((A_1 = v_1), \ldots, (A_i = v_i), \ldots, (A_k = v_k))$ with $v_i \in a_i$, and *cont-cell* is a cell in the continuous sub-space for A_{k+1}, .., A_m. The meanings of *empty* and *filled* cells follow those in the continuous space respectively. A *hole* is represented with

(*discr-region, cont-region*)

which is a region consisting of a set of empty cells, where *discr-region* is a combination of attribute-value pairs, i.e.,

$((A_{i_1} = v_{i_1}), \ldots, (A_{i_j} = v_{i_j}), \ldots, (A_{i_f} = v_{i_f}))$,

where $1 \leq f \leq k$, $A_{i_j} \in \{A_1, ..., A_i, ..., A_k\}$, and $v_{i_j} \in a_{i_j}$. When *discr-region* is fixed, *cont-region* is a set of connected empty cells in the continuous sub-space.

The definitions of a *maximal* hole and a *sufficiently large* hole are more complex in this case. We will not discuss them here. This paper focuses on the generation of *filled* and *empty* regions in a continuous space and a mixed space, and how to produce all the maximal holes in a continuous space.

3. The Proposed Approach

3.1 Overview of the Approach

Following our definition of a hole in a continuous space, our generalized hole discovery algorithm to be presented in Section 4 can already produce the complete set of sufficiently large maximal holes based on the filled cells. However, this solution may not be desirable for the same reasons as the existing algorithm in [7], i.e.,

1. It does not allow any filled cell to appear in a hole. This is too restrictive.
2. The running speed will be slow because of the high complexity of the algorithm and the large number of input cells. It also produces a huge number of holes.
3. It does not handle discrete attributes.

To address these issues, we transform the problem into a supervised learning task and use a decision tree engine to carve up the space into *filled* and *empty* regions. After obtaining the filled and empty regions (which are already holes, but may not be maximal) from the decision tree, we then use our generalized holes-discovery algorithm to produce all the maximal holes based on the filled regions.

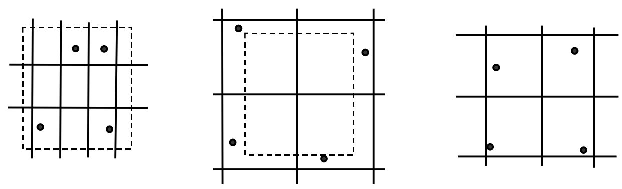
This approach addresses the above problems as follows. Problem 1 is addressed because a decision tree engine allows errors, i.e., an empty region can contain some outliers (i.e., filled cells) and vice versa. Problem 2 is addressed because we go through two stages in producing maximal holes. First, the decision tree engine produces the filled and empty regions, which can be done very efficiently. Then, based on the filled regions we can use our generalized holes-discovery algorithm to produce all the maximal holes. Since the number of filled regions can be much smaller than the number of original filled cells, the algorithm will run much faster, and also produce fewer holes. The third problem is also addressed because a decision tree engine can work in a mixed space (with both continuous and discrete attributes).

3.2 The Details

Our approach consists of the following three steps:

Partition the space into cells: We first partition each continuous attribute into a number of equal-length intervals (or units). The lengths along different dimensions may be different. The discrete attributes do not need to be partitioned. After partitioning, we can regard the space as consisting of hyper-rectangular cells. Those cells that have a density greater than the user-specified *threshold* (which is normally 0) are considered *filled*, otherwise, they are considered *empty*.

To partition the space, we need to know the cell size (for continuous attributes). This is computed using the user-specified minimum size of a *sufficiently large* hole. For continuous attributes, the user needs to provide the minimum length of the hole for each attribute. This is reasonable because in real applications, too small a hole is not meaningful. Assume our space has k continuous dimensions, and $l_1, \ldots, l_i, \ldots l_k$ are the user-specified minimum length of the hole along the dimensions. Then, the cell size along each dimension i will be $l_i/2$. The reason for choosing $l_i/2$ to partition the space is that when the cell is too small, we will find too many small holes, and this wastes computation. If the cell is too large, a filled region may contain sufficiently large holes. See Figure 1 below (the dashed box represents a sufficiently large hole, and the solid lines form the cells). In Figure 1(a), we see that each side of the cell is less than $l_i/2$. The area surrounded by the 4 points has some empty cells in it, which may form some leaf nodes (labeled *empty*) in the decision tree. But none of them is large enough. The cells in Figure 1(b) are too large. Although the 4 cells are all filled, there is an empty region among them that is sufficiently large. Figure 1(c) is the suitable size (= $l_i/2$) (i.e., the minimum sufficiently large hole in the figure is the size of the 4 cells combined). The 4 filled cells cannot contain a sufficiently large hole in it.



(a) (b) (c)

Figure 1. Selecting the cell size

Classifying the empty and filled regions: We use the decision tree engine in C4.5

[11] to carve the space into filled and empty regions. To apply C4.5 directly to the problem is not possible because those empty cells are not represented in the database. One solution is to add the points representing the empty cells into the original database. However, this may drastically increase the size of the database. We have modified C4.5 so that it can compute the required information regarding the empty cells when needed (see below). After the decision tree is constructed, the tree leaves labeled *empty* represent the empty regions, and the rest the filled regions.

The main modification to C4.5 is as follows. C4.5 uses information gain ratio in its computation to decide where to split the space. For this computation it only needs the frequency or the number of data points (in our case, cells) of each individual class on each side of a possible split. Since we do not have the points representing the empty cells, we have to compute it. This computation is quite simple. For a split in a subspace, the total number of cells on each side of the split is just the product of the number of units in each dimension. A unit along each continuous dimension i is $l_i/2$, and a unit along each discrete dimension is an individual value. Since we know the number of filled cells, we can easily compute the number of empty ones. Let us see an example. Figure 2 shows a continuous space.

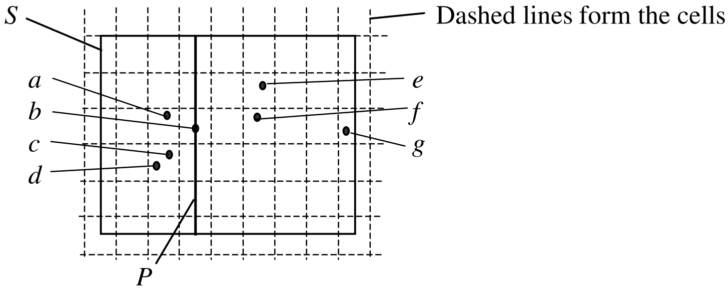


Figure 2. Computing the number of filled or empty cells

Assume the system is computing the information gain ratio at a possible split P. Note that splits are at data points as in C4.5 rather than along the borders of the cells. Let us compute the number of filled and empty cells. Assume a cell with one or more point is considered *filled*. On the left-hand-side of P, there are 16.5 cells altogether (partial cells are counted proportionally). 2.5 of them are filled cells. Note that point c and d are in the same cell and thus counted as representing one filled cell. Point b is in a partial cell on the left-hand-side. Assume it is half the size of a cell. Then the number of empty cells on the left-hand-side is 16.5 - 2.5 = 14. Likewise, the total number of cells on the right-hand-side of P is 27.5, and the number of filled cells is 2.5. Then the number of empty cells is 27.5 - 2.5 = 25.

With these numbers, C4.5 will be able to compute the information gain ratio to decide the best split. Finally, it builds a decision tree, where the leaf nodes are labeled either *filled* or *empty*. Next step produces maximal holes.

Produce all the maximal holes: The empty regions produced by C4.5 can be quite fragmented (see the example in the “illustration and experiments” section). Based on the filled regions in the decision tree, this post-processing step produces all the maximal holes. Currently, this is only performed in a continuous space. The new algorithm (Section 4) is a generalization of that in [7]. The algorithm in [7] uses data points as input. The new algorithm uses *filled regions* (FR) as input to produce the maximal holes. The algorithm in [7] is only a special case as a point is a special FR.

4. Producing All the Maximal Holes

After the space has been carved up, we use the *filled* regions (FR) to produce all the maximal holes. Since we only want those holes that are *maximal* and *sufficiently large*, our objective is to find all the holes in the k-dimensional space that satisfy the user-specified sufficiently large criterion and to rank them according to their sizes. Note that all the regions generated in the decision tree are hyper-rectangles [11].

Definition 1: Given a k-dimensional continuous space S bounded in each dimension i ($1 \leq i \leq k$) by a minimum and a maximum value (denoted by min_i and max_i), a hyper-rectangle (or HR in short), H, in S is defined as the region that is bounded on each dimension i by a minimum and a maximum value, denoted by $L_i(H)$ and $U_i(H)$ respectively, where $L_i(H) \geq min_i$ and $U_i(H) \leq max_i$. A HR has $2k$ bounding surfaces, 2 on each dimension. The two bounding surfaces on dimension i are parallel to axis i and orthogonal to all others.

In this paper, we are interested in the so-called *maximal hyper-rectangles* (*MHR*), which are our maximal holes.

Definition 2: Given a k-dimensional continuous space S and n FRs in S, a *maximal hyper-rectangle* (MHR) in S is an empty HR that does not intersect (in a normal sense) with any FR, and has at least one FR lying on each of its $2k$ bounding surfaces. We call these FRs the *bounding* FRs of the MHR.

Definition 3: A HR, H_1, *lies* on the lower (or upper) bounding surface in the ith dimension of another HR, H_2, if H_1 touches that bounding surface of H_2 in the ith dimension, and H_1 does not intersect with H_2.

4.1 Basic Idea of the Algorithm

The basic idea of the algorithm is as follows. Given a k-dimensional continuous space S, and n FRs (which may intersect with one another) in S, we first start with one MHR, which occupies the entire space S. Then each FR is incrementally added to S. At each insertion, we update the set of MHRs that have been found this far. The update is done as follows. When a new FR is added, we identify all the existing MHRs that intersect with this FR. These hyper-rectangles are no longer MHRs since they now contain part of the FR within their interiors. They are then removed from the set of existing MHRs. Using the newly added FR as reference, a new lower and upper bound for each dimension are formed to result in 2 new hyper-rectangles along that dimension. If these new hyper-rectangles are found to be MHRs and sufficiently large, they are inserted into the list of existing MHRs, otherwise they are discarded. The main difference between this algorithm and that in [7] is that we tests for intersection of hyper-rectangles, while the algorithm in [7] tests for containment of a point in a hyper-rectangle.

4.2 The Detailed Algorithm

Before presenting the algorithm, let us define the main representations and functions used. A *filled HR* (FR), H, is denoted by:

$$H = ((L_1(H), U_1(H)), \ldots, (L_i(H), U_i(H)), \ldots, (L_k(H), U_k(H)))$$

where $L_i(H)$ and $U_i(H)$ are respectively the lower and upper *bounding values* of H along the i^{th} dimension. A MHR, M, is denoted by:

$$M = ((Sl_1, Su_1), \ldots, (Sl_i, Su_i), \ldots, (Sl_k, Su_k))$$

where Sl_i and Su_i are respectively the sets of lower and upper bounding FRs of M along the i^{th} dimension.

```
Algorithm FindMHR
  Insert(T, (({bl_1},{bu_1}), …, ({bl_i},{bu_i}), …, ({bl_k},{bu_k})));
  for each FR H in the decision tree do
    RL = IntersectionSearch(T, H);
    for each M = ((Sl_1,Su_1), …, (Sl_i,Su_i), …, (Sl_k,Su_k)) in RL do
      if H is on a surface of M then   insert H into the set of bounding FRs in that surface
      else   Delete(T, M);
             for each dimension i do
                if L_i(H) > U_i(l) for l ∈ Sl_i then
                   for each j ≠ i do
                      Sl′_j = {l ∈ Sl_j | L_i(l) < L_i(H)};
                      Su′_j = {u ∈ Su_j | L_i(u) < L_i(H)};
                   endfor;
                   if Sl′_j and Su′_j not empty for all j ≠ i and
                      BigEnough(M′ = ((Sl′_1, Su′_1), …, (Sl_i, {H}), …, (Sl′_k, Su′_k)) then
                     Insert(T, M′)
                   endif;
                endif;
                if U_i(H) < L_i(u) for u ∈ Su_i then
                   for each j ≠ i do
                      Sl″_j = {l ∈ Sl_j | U_i(l) > U_i(H)};
                      Su″_j = {u ∈ Su_j | U_i(u) > U_i(H)}
                   endfor;
                   if Sl″_j and Su″_j not empty for all j ≠ i and
                      BigEnough(M″ = ((Sl″_1, Su″_1), …, ({H},Su_i), …, (Sl″_k, Su″_k)) then
                     Insert(T, M″)
                   endif
                endif
             endfor
      endif
    endfor
  endfor
  Sort and report all the MHRs in T according to their sizes;
```

Figure 3. The Holes Discovery Algorithm

Note that the lower (or the upper) bound of a MHR is bounded by a set of lower (or upper) bounding FRs, rather than a single bounding value as used in a FR. The upper bounds of all the FRs in Sl_i along the i^{th} dimension are the same, and the lower bounds of all the FRs in Su_i along the i^{th} dimension are also the same. That is,

$U_i(H_a) = U_i(H_b)$ for all $H_a, H_b \in Sl_I$, and

$L_i(H_a) = L_i(H_b)$ for all $H_a, H_b \in Su_i$.

For simplicity of notation, we let bl_i and bu_i to be the lower and upper bounding FRs (in each dimension i) of the initial space, S, where $L_i(bl_i) = U_i(bl_i) = min_i$ and $L_i(bu_i) = U_i(bu_i) = max_I$, and $L_j(bl_i) = U_j(bl_i)$ = undefined for $i \neq j$. Any comparison with $L_j(bl_i)$ or $U_j(bl_i)$ (where $i \neq j$) is always true. Hence, bl_i and bu_i serve as the lower and upper *bounding planes* along the i^{th} dimension respectively.

Next, we define the functions used in the algorithm. Let T be a data structure that stores a collection of sufficiently large MHRs. T supports the following functions:

(1) *Insert*(T, M): it inserts the MHR M into T.

(2) *Delete*(T, M): it deletes the MHR M from T.
(3) *IntersectionSearch*(T, H): it returns a list of MHRs (RL) from T, where each $M \in RL$ either intersects with the FR H or H lies on a surface of M.

Lastly, we define a function *BigEnough*(M) which returns TRUE if the MHR M is considered to be sufficiently large. Note that if *BigEnough*(M') is true, then *BigEnough*(M) must be true for all M that contain M'. The algorithm is shown in Figure 3.

The worse case time complexity of the algorithm is the same as the algorithm in [7]. However, the new algorithm is more practical because it takes FRs rather than data points as input. The number of FRs produced in a decision tree is normally small. The correctness of the algorithm can be similarly proved as the algorithm in [7]. See [9] for the detailed proof and complexity analysis.

4.3 An Example

We now use an example to illustrate the working of the algorithm. Suppose we have a 2-dimensional space S (ABCD) as shown in Figure 4. The first MHR is the whole space, ABCD, which is described with:

$M = ((\{bl_1\}, \{bu_1\}), (\{bl_2\}, \{bu_2\})))$

Line 3. We add the first FR H_1 to the inside of ABCD (see Figure 5).

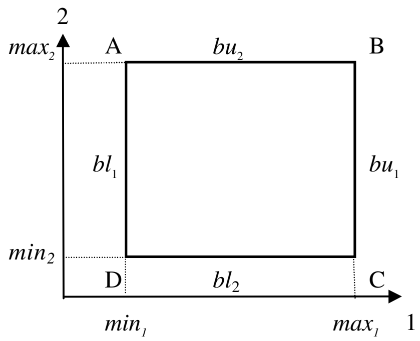


Figure 4: The initial space S or the initial MHR

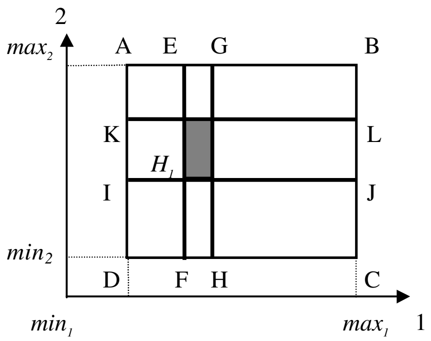


Figure 5. Splitting ABCD

Line 4. Since in this case there is only one MHR (the original space ABCD) intersecting H_1 in T, thus

$RL = \{((\{bl_1\}, \{bu_1\}), (\{bl_2\}, \{bu_2\}))\}$.

Line 5. The first MHR to be considered is

$M = ((\{bl_1\}, \{bu_1\}), (\{bl_2\}, \{bu_2\}))$, i.e., ABCD.

Line 6. Since H_1 (Figure 5) is not on a surface of M, we proceed to Line 7.

Line 7. M is deleted since it is no longer a MHR.

Line 8. We split M along each dimension. D-1 is considered first.

Line 9-17. This part forms the new MHR to the left side of H_1. Sl'_2 and Su'_2 are the sets of lower and upper bounding FRs for the new MHR. We obtain:

$Sl'_2 = \{bl_2\}$ and $Su'_2 = \{bu_2\}$.

The new MHR is thus: $M' = ((\{bl_1\}, \{H_1\}), (\{bl_2\}, \{bu_2\}))$, i.e., AEFD,

Line 18-26. This part forms the new MHR to the right side of H_1. Sl''_2 and Su''_2 are the sets of lower and upper bounding FRs for the new MHR along dimension 2. We obtain: $Sl''_2 = \{bl_2\}$ and $Su''_2 = \{bu_2\}$.

The new MHR is thus: $M'' = ((\{H_1\}, \{bu_1\}), (\{bl_2\}, \{bu_2\}))$, i.e., GBCH.

After considering D-2, we form 2 more MHRs:

$M' = ((\{bl_1\}, \{bu_1\}), (\{bl_2\}, \{H_1\}))$, i.e., IJCD.

$M'' = ((\{bl_1\}, \{bu_1\}), (\{H_1\}, \{bu_2\}))$, i.e., ABLK.

Assume the 4 new MHRs are big enough and are inserted into T. Adding the first FR is completed. Let us now add another FR, H_2 (see its location in Figure 6).

Line 4. The set of MHRs in T intersecting H_2 is:

$RL = \{((\{bl_1\}, \{bu_1\}), (\{bl_2\}, \{H_1\})),$
$((\{H_1\}, \{bu_1\}), (\{bl_2\}, \{bu_2\}))$

They are IJCD and GBCH respectively in Figure 5.

Line 5-7. We consider the first MHR, $M = ((\{bl_1\}, \{bu_1\}), (\{bl_2\}, \{H_1\}))$, which is IJCD in Figure 5 and 6. Since H_2 is not on a surface of M, M is deleted.

Line 8-27. After considering all the dimensions, we obtain the following two new MHRs (see Figure 6):

D-1: $M' = ((\{bl_1\}, \{H_2\}), (\{bl_2\}, \{H_1\}))$, i.e., IMND.

D-2: $M' = ((\{bl_1\}, \{bu_1\}), (\{bl_2\}, \{H_2\}))$, i.e., OPCD.

There is no M'' in D-1 because $Su'_2 = \{\}$. M'' should be (if it exists) VJCS in Figure 6, but VJCS is not a MHR because there is no bounding FR on VJ. Its space will be occupied later by another MHR. There is also no M'' in D-2 because the condition in line 18 is not satisfied. Assume the above two MHRs are big enough. They are inserted into T.

Line 5-29. We now go back to work on the other MHR,

$M = \{((\{H_1\}, \{bu_1\}), (\{bl_2\}, \{bu_2\})),$

which is GBCH in Figure 5 and 6. After going though the similar process as above, we obtain 3 new MHRs (see Figure 6),

D-1: $M' = ((\{H_1\}, \{H_2\}), (\{bl_2\}, \{bu_2\}))$, i.e., GQNH.

$M'' = ((\{H_2\}, \{bu_1\}), (\{bl_2\}, \{bu_2\}))$, i.e., RBCS.

D-2: $M'' = ((\{H_1\}, \{bu_1\}), (\{H_2\}, \{bu_2\}))$, i.e., GBUT.

Assume they are big enough. They are inserted into T.

The final of set of MHRs in T are GQNH, RBCS, GBUT, IMND, OPCD, AEFD, and ABLK respectively (Figure 6).

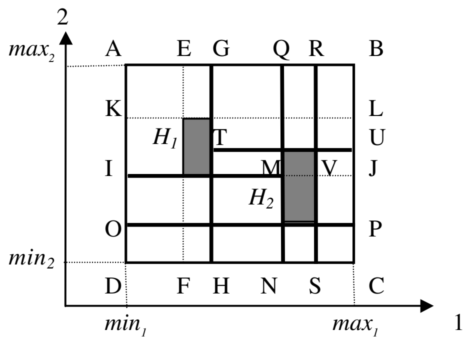


Figure 6. Forming MHRs after H_2 is inserted

5. Illustration and Experiments

A system has been implemented in C, which consists of two parts. The first part, which builds the decision tree, is modified from C4.5 [11]. The second part implements our generalized MHR discovering algorithm. So far we have carried out many experiments. Since there is no existing system to compare with, this section first gives an example to show the working of the system, and then presents some experiment results of the generalized MHR discovering algorithm. It is not appropriate to compare the holes discovered by our system with the holes produced by the algorithm in [7] because the algorithm in [7] does not allow any point in its MHRs.

Our example uses the Iris data from UCI machine learning repository [8], which has 150 cases and 4 continuous attributes. Only two attributes are used here. The class information in the data is not needed. Figure 7 shows the splits produced by our modified C4.5, which form many filled and empty rectangles. The minimal size of the hole we use is 0.6×0.6. The shaded rectangles are the filled regions, and the rest are the empty regions (which also contain some data points). From the figure, we see that the empty regions are fragmented. Figure 8 shows the final results after running our generalized MHR discovering algorithm. There are 7 MHRs discovered (numbered in the two opposite corners of each MHR). Clearly, these MHRs are better than the empty regions in Figure 7. The algorithm in [7] actually finds 22 MHRs, which are different from ours as they contain no data points.

Next, we show the performances of our generalized MHR discovery algorithm. All the data sets are randomly generated. We present the results from our experiments in 2-D, 3-D, 4-D, 5-D and 6-D spaces. Holes with more than 4-D are quite hard to understand. For each number of dimensions, we generated 6 data sets with, 50, 100, …, and 500 FRs respectively. We did not use more FRs in our experiments because a decision tree does not normally produce a large number of FRs. In data generation, as the number of FRs increases, we also increase the size of the space proportionally. We used two minimum hole sizes in our *BigEnough* function. The first one (min. size 1) is 40.00 in each dimension, and the second (min. size 2) is 50.00 in each dimension. Table 1 shows the number of MHRs found in each experiment. Table 2 summarizes the running times (512MB Sparc Ultra-2).

From the tables, we see that when the minimum size of the MHR decreases, the time taken to find all the sufficiently large MHRs increases. As the number of FRs increases, the time taken to find all the large MHRs also increases. However, in general, the relationship between the number of FRs and the time taken to find all the large MHRs is complex because there are other factors that also need to be considered, e.g., the distribution of the FRs, and the size of the initial space.

In all experiments, the running times are reasonable. Although there can be many MHRs produced, the user does not have to see all of them at one time. A query system can be implemented to allow the user to specify what he/she is interested in seeing. It is also possible that the user simply takes those large empty regions produced in the decision tree as the final holes without generating all the maximal ones.

6. Related Work

Current research in machine learning and data mining focuses on finding rules that exist in data. Our work is different from rule induction (e.g., [11]) and conceptual clustering (e.g., [5]) because they are only concern with what is in the data, while we are concerned with both what is in the data and what is missing from the data.

Scientific discovery systems such as Abacus [3], and Bacon [6] typically discover mathematical formulas in data. With the formulas, we know exactly where the data points are and where there should be no point. However, in most business databases such formulas rarely exist.

[7] first studied the problem of discovering holes in data. An algorithm was pro-

posed to discover holes by taking data points as input. Our generalized algorithm can take either FRs or data points (as input). We have discussed the shortcomings of that algorithm throughout the paper. This research helps to overcome its major problems.

In geometric algorithms research, there are algorithms [2, 10] that can find empty rectangles in the 2-D space, these algorithms cannot be extended to the multi-dimensional space. They also do not allow any point in each empty rectangle.

Our work is also related to density estimation in statistics research [12]. Density estimation is the construction of an estimate of a density function from the given data. It does not provide descriptions of empty regions (for which further processing is needed), and it also does not handle discrete attributes. The proposed technique deals with both. Our technique for finding holes is also quite different from the density estimation method as we use information theory and a geometric algorithm.

7. Conclusion

Although discovering the associative relations that exist in data is important, in many situations, discovering of large holes in the database is also significant and interesting. This paper presented a new approach for discovering holes. It first transforms the problem into a supervised learning task and uses a decision tree induction engine to divide the space into filled and empty regions. It then produces maximal holes using the filled regions generated in the decision tree. This new approach overcomes three major shortcomings of the existing method in [7].

References

1. Agrawal, R., and Srikant R. 1994. Fast algorithms for mining association rules. *VLDB-94*, 1994.
2. Chazelle, B., Drysdale, R. L., and Lee, D. T. 1986. Computing the largest empty rectangle. *SIAM Journal of Computing*, 15(1) 300-315.
3. Falkenhainer, F., and Michalski, R. 1986. Integrating quantitative and qualitative discovery: the ABACUS system. *Machine Learning*, 1(4):367-401.
4. Fayyad, U., Piatesky-Shapiro, G., and Smyth, P. 1996. From data mining to knowledge discovery in databases. *AI Magazine* 37-54.
5. Fisher, D. 1987. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2:139-172.
6. Langley, P., Simon, H., Bradshaw, G., and Zytkow, Jan. 1987. *Scientific discovery: computational explorations of the creative process*, The MIT press.
7. Liu, B., Ku, L. P., and Hsu, W. 1997. Discovering Interesting Holes in Data. *IJCAI-97*, 930-935.
8. Merz, C. J, and Murphy, P. 1996. UCI repository of machine learning database [http://www.cs.uci.edu/~mlearn/MLRepository.html].
9. Mun, L. F. 1998. *Discovering missing and understandable patterns in databases*. MSc thesis, National University of Singapore.
10. Orlowski, M. 1990. A new algorithm for the largest empty rectangle problem. *Algorithmica*, 5:65-73.
11. Quinlan, J. R. 1992. *C4.5*: *program for machine learning*. Morgan Kaufmann.
12. Silverman, B. W. 1986. Density Estimation for Statistics and Data Analysis. Chapman and Hall.