All Dimension Trees

To close out this week's discussion, we will consider an advanced data structure that is being used to support analyses for data mining. As we develop this data structure (and a couple supporting algorithms), we note that much of data mining involves counting instances within a large data set that satisfy some query. Because data mining data sets tend to be extremely large, generating these counts can be very time consuming. Thus one of the major goals of data mining research is efficiently finding patterns and associations in large data sets. Of particular interest is generating these counts with minimum time **and** space.

Contingency Tables

The principal conceptual structure that we want to work with is the **contingency table**. To define what a contingency table is, suppose we have R records in a data set, and each record has M attributes (or fields).

Let's denote these attributes as $a_1, \dots, a_M$ respectively. Let n_i be the arity (i.e., the number of values) of attribute a_i. Then let $a_{i(j)}$ indicate the jth value of attribute a_i.

From here, we will use examples to support our definition. Specifically, suppose we have the following data set:

Attribute	a_1	a_2	a_3
Arity (n)	2	4	2
Record 1	1	1	1
Record 2	2	3	1
Record 3	2	4	2
Record 4	1	3	1
Record 5	2	3	1
Record 6	1	3	1

For this data set, we note that $M = 3$ and $R = 6$.

Def: A **query** is a set of $\langle \mathit{attribute}, \mathit{value} \rangle$ pairs in which the query attributes are a subset of $\{a_1, \dots, a_M\}$ arranged in increasing order of index.

For example, we can specify the following queries on the above data set:

$$()$$
$$(a_1 = 1)$$
$$(a_2 = 3, a_3 = 1)$$
$$(a_1 = 2, a_2 = 3, a_3 = 1)$$

Def: A **count** of a query, denoted $C(\mathit{Query})$, is the number of records in the data set matching all of the pairs in *Query*.

For example, for the sample queries above, we get the following counts:

$$C() = 6$$
$$C(a_1 = 1) = 3$$
$$C(a_2 = 3, a_3 = 1) = 4$$
$$C(a_1 = 2, a_2 = 3, a_3 = 1) = 2$$

Def: Each subset of attributes, $a_{i(1)}, \dots, a_{i(n)}$, has an associated contingency table, denoted $\mathbf{ct}(a_{i(1)}, \dots, a_{i(n)})$.

This yields a table with a row for each of the possible sets of values for $a_{i(1)}, \dots, a_{i(n)}$.

So the row corresponding to $a_{i(1)} = v_1, \dots, a_{i(n)} = v_n$ records the count $C(a_{i(1)} = v_1, \dots, a_{i(n)} = v_n)$.

Def: A **conditional contingency table**, denoted $\mathbf{ct}(a_{i(1)}, \dots, a_{i(n)} | a_{j(1)} = u_1, \dots, a_{j(p)} = u_p)$, is the contingency table for the subset of records in the dataset that matches the query to the right of the "|" symbol.

The following correspond to the contingency tables for the data set given above. In these tables, the column denoted "#" provides the corresponding count of records matching the attribute values in the other columns.

$\mathbf{ct}()$

#
6

$\mathbf{ct}(a_1)$

a_1	#
1	3
2	3

$\mathbf{ct}(a_3)$

a_3	#
1	5
2	1

$\mathbf{ct}(a_1,a_2,a_3)$

a_1	a_2	a_3	#
1	1	1	1
1	1	2	0
1	2	1	0
1	2	2	0
1	3	1	2
1	3	2	0
1	4	1	0
1	4	2	0
2	1	1	0
2	1	2	0
2	2	1	0
2	2	2	0
2	3	1	2
2	3	2	0
2	4	1	0
2	4	2	1

$\mathbf{ct}(a_2)$

a_2	#
1	1
2	0
3	4
4	1

$\mathbf{ct}(a_1,a_2)$

a_1	a_2	#
1	1	1
1	2	0
1	3	2
1	4	0
2	1	0
2	2	0
2	3	2
2	4	1

$\mathbf{ct}(a_2,a_3)$

a_2	a_3	#
1	1	1
1	2	0
2	1	0
2	2	0
3	1	4
3	2	0
4	1	0
4	2	1

$\mathbf{ct}(a_1,a_3)$

a_1	a_3	#
1	1	3
1	2	0
2	1	2
2	2	1

and the following correspond to some sample conditional contingency tables for the same data set:

$\mathbf{ct}(a_1,a_3 \mid a_2=3)$

a_1	a_3	#
1	1	2
1	2	0
2	1	2
2	2	0

$\mathbf{ct}(a_1,a_2 \mid a_3=1)$

a_1	a_2	#
1	1	1
1	2	0
1	3	2
1	4	0
2	1	0
2	2	0
2	3	2
2	4	0

$\mathbf{ct}(a_2,a_3 \mid a_1=2)$

a_2	a_3	#
1	1	0
1	2	0
2	1	0
2	2	0
3	1	2
3	2	0
4	1	0
4	2	1

In these tables, the shaded boxes represent those counts that change over the unconditional contingency tables as a result of conditioning on the query.

Theoretically, it is possible to precompute all of the contingency tables for a particular data set. Unfortunately, the memory required to hold these contingency tables grows as

$$\prod_{i=1}^{M}(n_i+1) = O((1+\max_i n_i)^M) = \Omega((1+\min_i n_i)^M)$$

Note that the "+1" in this expression arises from the fact that a specific attribute value may or may not occur in a specific query. This means we must include a "don't care" value for that attribute as well. The purpose of the ADtree is to provide an approach to reduce this bound to a more manageable level.

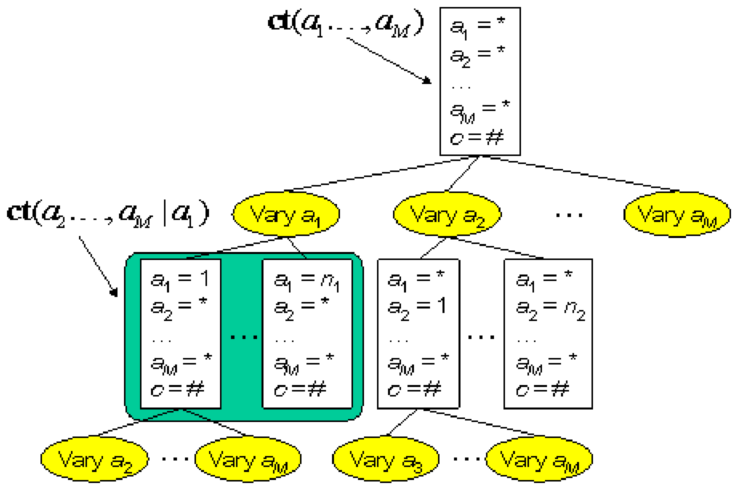
Dense *AD*Trees

Rather than store the explicit contingency tables, we will begin by storing the equivalent information in a tree data structure. This does not yet yield any memory savings, but the data structure will provide opportunities of improving the memory bound later. For this data structure, we need to include two different kinds of nodes:

1. *AD* nodes store the queries and associated "conditional" contingency tables. They also contain one child for each uninstantiated variable.

2. Vary nodes identify specific values of a single attribute for conditioning, and it contains one child for each value of that attribute.

The general form of the *AD*tree looks something like the following:



From this, we see that the contents of a particular AD node are simply counts (corresponding to specific queries) and sets of pointers to the child Vary nodes aj.

The contents of the Vary nodes, on the other hand, are the sets of pointers to the next level of *AD* nodes. With this structure, we see that the cost of looking up a specific count is proportional to the number of instantiated variables in the particular query.

If an *AD* node has Vary aj as a parent, the *AD* node's children are Vary $a_{j+1}, \ldots,$ Vary a_M.

Notice that indices below $j+1$ are found in other paths of the tree, so they do not need to be specified explicitly.

Sparse *AD*Trees

As mentioned above, the dense *AD*tree structure does nothing to reduce either memory or processing requirements; however, this data structure opens the door to finding ways to do this. For example, one simple observation we can make is that there is no need to expand a node's subtree once it has a count of zero. This is because all further specializations from that point must also have a count of zero. Therefore, we can prune the tree at that point and replace the count in the node with NIL value. This simple modification often leads to substantial reduction in the memory requirements for the *AD*tree; however, it does nothing to change the theoretical bounds.

We do not need to stop there, however. To continue, notice that the *AD*tree actually stores a lot of redundant information. The trick is to identify the conditions under which redundancy arises and then modify the tree structure to remove this redundancy. For example, we can drop one child of every *AD* node simply by observing that its count can be derived by subtracting the sum of counts in sibling nodes from the count of the parent *AD* node. If we choose an appropriate node to remove, this can also lead to substantial savings. That node, in fact, is the one with the highest count since such a node is the one most likely to have a large subtree.

So for the set of children of some Vary node, find the *AD* node that would have the highest count and designate it *MCV* (for most common value) for the attribute aj designated by its parent Vary a_j node.

Replace the count in this node with NIL and do not expand the associated subtree. We would represent all other nodes in the tree as we have before. This actually leads to an order of magnitude reduction of the exponential in the size of the tree.

Using this idea, we can then construct an *AD*tree using the following algorithm:

Algorithm 1 Constructing an ADtree

```
MakeADTree($a_j$, $RecordNums$)
// $a_j$ denotes starting attribute in $a_j \ldots a_M$
// $RecordNuma$ is a subset of records $\{1, \ldots, R\}$
  Make a new $AD$ node, called $ADN$
  $ADN.count \leftarrow |RecordNums|$
  for $j \leftarrow i$ to $M$ do
    Vary $a_j \leftarrow$ MakeVaryNode($a_j$, $RecordNums$)
    $ADN.j \leftarrow$ Vary $a_j$
end
```

This algorithm would be called with MakeADTree($a_1, \{1, \ldots, R\}$). The algorithm proceeds by creating an initial AD node at the root and setting the count for that node equal to the number of records in the data set. Then each of the M attributes is considered in turn.

At this point, a Vary ai node is created for each of theai attributes, and a pointer is created from the parent AD node to each of that node's vary nodes.

Thus, much of the work in building the ADtree occurs in the subroutine, MakeVaryNode. The corresponding algorithm for constructing a Vary node is as follows:

Algorithm 2 Constructing a Vary Node

```
MakeVaryNode($a_j$, $RecordNums$)
  Make a new Vary node called $VN$
  for $k \leftarrow 1$ to $n$ do
    $Childnums[k] \leftarrow \{\}$
  for each $j \in RecordNums$ do
    $v_{ij} \leftarrow$ Value of attribute $a_i$ in record $j$
    Add $j$ to set $Childnums[v_{ij}]$
  $VN.MCV \leftarrow \arg\max_k |Childnums[k]|$
  for $k \leftarrow 1$ to $n$ do
    if $|Childnums[k]| = 0$ or $k = MCV$ then
      Set $a_j = k$ subtree of $VN$ to NIL
    else
      Set $a_j = k$ subtree of $VN$ to MakeADTree($a_{j+1}$, $Childnums[k]$)
end
```

This algorithm is called with the specific attribute identified as well as the set of records from the parent AD node. Then n sets are initialized so we can partition the records that match a particular value for the attribute associated with the Vary node. Then the records are scanned and placed in the appropriate *Childnums* set where the attribute value matches. Once all of the records have been scanned and partitioned, the MCV is determined by finding the largest *Childnums* set. Then we run through each of the *Childnums* sets and either create a new ADtree subtree rooted at that Vary node, or cut off the subtrees when the *Childnums* set is either empty or equal to the MCV.

Extracting Contingency Tables

Once the ADtree is constructed, we are in a position to retrieve the associated contingency tables. To do this, we construct a desired contingency table recursively upon demand. For example, suppose we want to find $\mathbf{ct}(a_{i(1)}, \ldots, a_{i(n)} | Query)$. To do this, we first build the following:

$$\begin{gathered}
\mathbf{ct}(a_{i(2)}, \ldots, a_{i(n)} | a_{i(1)} = 1, Query) \\
\mathbf{ct}(a_{i(2)}, \ldots, a_{i(n)} | a_{i(1)} = 2, Query) \\
\vdots \\
\mathbf{ct}(a_{i(2)}, \ldots, a_{i(n)} | a_{i(1)} = n_{i(1)}, Query)
\end{gathered}$$

The base case of the recursion occurs when the left hand side is empty. At that point, we simply return the count associated with the *AD* node containing the empty query. The specific algorithm to construct the contingency table is as follows:

Algorithm 3 Extracting a Contingency Table

```
MakeContab({$a_{i(1)}, \ldots, a_{i(n)}$}, $ADN$)
  $VN \leftarrow$ Var $a_{i(1)}$ subnode of $ADN$
  $MCV \leftarrow VN.MCV$
  for $k \leftarrow 1$ to $n_{i(1)}$ do
    if $k \neq MCV$ then
      $ADN_k \leftarrow$ the $a_{i(1)} = k$ subnode of $VN$
      $CT_k \leftarrow$ MakeContab({$a_{i(2)}, \ldots, a_{i(n)}$}, $ADN$) $- \sum_{j \neq MCV} CT_j$
end
```

This algorithm returns a particular contingency table as follows. First, it begins with the root *AD* node (*ADN*) and identifies the first Vary node, Vary $a_{i(1)}$. It also identifies the *MCV* for that parent *AD* node given the Vary node. From there, it considers each of the attribute values for the Vary node. As it examines each attribute value, is determines whether that corresponds to the *MCV*. If it does not, then the child *AD* node is identified corresponding to the kth value of the Vary node.

The contingency table for that next node is then determined by calling MakeContab recursively. Notice, however, that we do not return CT_{MCV} because the corresponding subtree for the *MCV* has been pruned from our *AD*tree.

From here, we note that we can calculate the missing conditional contingency table as

$$\mathbf{ct}(a_{i(2)}, \ldots, a_{i(n)} | Query) = \sum_{k=1}^{n_{i(1)}} \mathbf{ct}(a_{i(2)}, \ldots, a_{i(n)} | a_{i(1)} = k, Query)$$

or

$$CT_{MCV} = \textsc{MakeContab}(\{a_{i(2)}, \ldots, a_{i(n)}\}, ADN) - \sum_{k \neq MCV} CT_k$$

Computing the Counts

Now that we have a way to build an *AD*tree and a way to derive contingency tables, we are ready to answer the question that motivates the data structure. For a specific query, how many records in the data set satisfy that query? Contingency tables help (and if fully enumerated can answer the question), but deriving the relevant table is actually more complex than necessary. Consider the following algorithm:

Algorithm 4 Counting Records

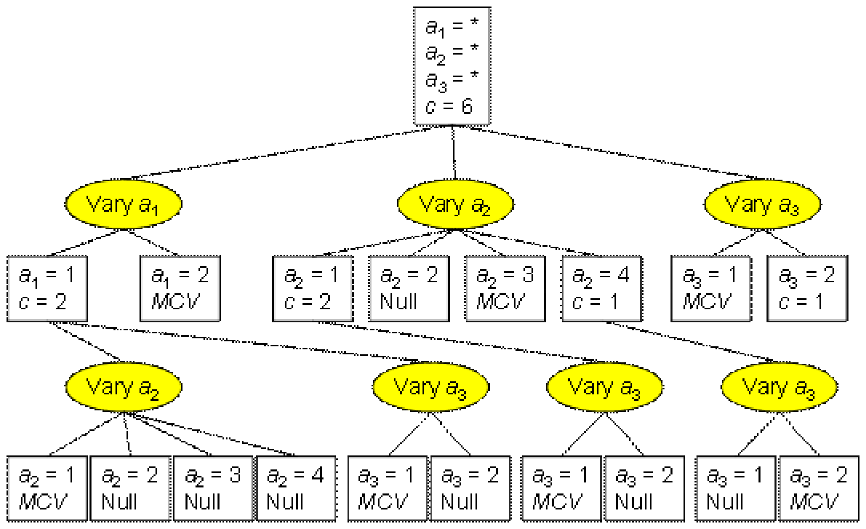
```
ADCount($ADN$, $QueryList$, $index$)
  if $index = |QueryList|$ then
    return $C(ADN)$
  $Vary \leftarrow$ Vary $a_{index}$
  $NextADN \leftarrow$ Vary $a_{index}.QueryList[index]$
  if $C(NextADN) = 0$ then
    return 0
  if $NextADN$ is $MCV$ then
    $Count \leftarrow$ ADCount($ADN$, $QueryList$, $index + 1$)
    for each sibling $s$ of $ADN$ do
      $Count \leftarrow Count-$ ADCount($s$, $QueryList$, $index + 1$)
    return $Count$
  return ADCount($NextADN$, $QueryList$, $index + 1$)
end
```

This algorithm is designed to return the count for a specific query. It is called with a pointer to the root of the *AD*tree, the query list, and an index into the query list (initially 1). Upon first entering the algorithm, a test is performed to see if the index is pointing to the end of the query list. If so, the algorithm is finished, and the count can be returned. Otherwise, we consider the Vary node corresponding to the indexed attribute value, and we traverse into the child *AD* node based on the query value. If the count associated with that *AD* node is zero, then that is the count returned. If the next *AD* node happens to be the *MCV*, then we determine the count for the next *AD* node in the query. After that, we consider the sibling of the current *AD* node. This is because we need to know the sibling counts so we can derive the *MCV* count. That *MCV* count is derived as the count from the rest of the query minus the counts of the siblings. If the *AD* nodes was not the *MCV*, then we simply return its count.

To illustrate, consider the following data set (which is slightly different from the data set first introduced):

Attribute	a_1	a_2	a_3
Arity (n)	2	4	2
Record 1	1	1	1
Record 2	2	3	1
Record 3	2	4	2
Record 4	1	1	1
Record 5	2	3	1
Record 6	2	3	1

From this data set we derive the following ADtree:



Now suppose we wish to find $C(2, 3, 1)$. In other words, we want the number of records in the data set such that $a_1 = 2$, $a_2 = 3$, and $a_3 = 1$. A quick look at the table above will show we should find three of them. We start by calling ADCOUNT(root, {2, 3, 1}, 1}. From the **root**, we move to the Vary node indicated by the index "1" and consider the value of the associated attribute. That attribute is a_1, and we are interested in the second value (i.e., $a_1 = 2$). Examining the AD node below Vary a_1, we see that $a_1 = 2$ is an MCV. The algorithm tells us to increment the index and consider the next AD node over, so we call ADCOUNT(root, {2, 3, 1}, 2}. Again, examining the Vary a_2 nodes children reveals that $a_2 = 3$ is an MCV. This leads us to call ADCOUNT(root, {2, 3, 1}, 3). Now the termination test hits because $index = 3$. From here, we simply return the count for $a_3 = 1$. But we see that this is another MCV, so this is going to be the size of the data set (because we are at the end of the list) minus the count of the siblings $= 6 - 1 = 5$.

Once we find $C(a_3 = 1)$, we are ready to pop the recursion stack and return to $C(a_2 = 3)$. This requires us to consider the siblings, s. So for every sibling, we calculate $Count = Count - \text{ADCOUNT}(s, \{2, 3, 1\}, 2)$. Since we are starting with $Count = 5$ (returned from the call for a_3), we get $Count = 5 - (a_2 = 1) - (a_2 = 2) - (a_2 = 4)$. Let's consider each of these. First, we can get rid of $(a_2 = 2)$ right away since it has a NIL pointer. This means the associated count is zero. Consider $(a_2 = 1)$. This leads to a recursive call to ADCOUNT, but from the figure we can see that there are only two children.

Here we apply the rest of the query and must consider $(a_3 = 1)$, which is an MCV. But notice that the other child has a zero count $(a_3 = 2)$, because of the NIL pointer), so the count for the MCV is 2. This means we must consider the entire count for $a_2 = 2$, and that is 2. So now we have $Count = 5 - 2 - 0 - (a_2 = 4)$. For $(a_2 = 4)$, we also must call ADCOUNT recursively, and we see two children again. This time, because of the query we have, we again must consider the rest of the query $(a_3 = 1)$, and this time, we see that its count is zero. Therefore, the count for $a_2 = 3$ is $Count = 5 - 2 - 0 - 0 = 3$.

Now that we have finished with $C(a_2 = 3)$, we pop the recursion stack and return to $C(a_1 = 2)$. Again, we must consider the siblings to $(a_1 = 2)$, and as we call ADCOUNT recursively, we consider the queries. This amounts to traversing the tree along $(a_2 = 3)$ and $(a_3 = 1)$ again. But notice that when we consider $(a_2 = 3)$, we find a NIL pointer. This means the count is zero, as is the count for a_3 below. So our count is now calculated as $Count = 3 - 0 = 3$, and this is the correct answer.

Our goal in this part of the discussion was to consider a new data structure to see if we could improve performance in counting over large data sets. How did we do? To answer this question, assume all attributes have the same arity k. Then a contingency table with n attributes will have kn entries. Clearly, this is too large to manage in a brute force fashion. Let $C(n)$ denote the cost of computing such a contingency table. At the top level of MAKECONTAB, there are k calls to build contingency tables from $n - 1$ attributes. Then there are $k - 1$ subtractions of contingency tables, each requiring $kn - 1$ subtractions. This leads to the recurrence

$$C(n) = \begin{cases} 1 & n = 0 \\ kC(n-1) + (k-1)k^{n-1} & n > 0. \end{cases}$$

If we solve the recurrence, we find $C(n) = (1 + n(k-1))k^{n-1}$. With this, a brute force count (without ADtrees) would require $O(nR + k^n)$ operations. ADtrees will provide a cost savings whenever $k^n \ll R$. Then using ADtrees yields a complexity of $O(n \times \min(R, k^n))$.