Fast Counting with AV-Space for Efficient Rule Induction

Linyan Wang   and   Aijun An[*]

Abstract

We present AV-space, a new data structure for caching data set statistics for efficiently learning classification rules from large data sets. The AV-space is designed to work with sequential-covering rule induction algorithms. It is used to accelerate queries about the count of the examples in a data set that satisfy a conjunction of attribute-value pairs. With an AV-space, the learning algorithm does not have to access the training data to obtain the statistics about the data. We present the structure of an AV-space, algorithms for building and querying an AV-space, and procedures for dynamically updating the AV-space during the rule induction process. We present an experimental evaluation that compares the AV-space with a commonly-used data structure that simply loads the (encoded) training examples into memory. We show that the use of AV-space significantly improves the speed of rule induction and that it consumes less memory on large data sets.

1 Introduction

Rule induction is a machine learning technique that discovers a set of *if-then classification rules* from data. An example of a classification rule is "*if Temperature ≠ normal and Headache = yes then the patient has Flu.*" The most widespread approach to inducing a set of classification rules is called *sequential-covering*. Examples of sequential-covering algorithms include CN2 [4], ELEM2 [1] and PRISM [3]. A sequential-covering algorithm learns a set of rules for every class in turn. For each class, it sequentially learns a set of rules that together cover the set of training examples belonging to the class. During this process, the most frequent operation is to count the examples that satisfy a conjunction of attribute-value pairs. Thus, the speed of the rule induction process greatly depends on the time spent on counting examples. Fast counting can accelerate the induction process.

Most sequential-covering programs load the training data or an encoded version of the training data into the memory. The most commonly used data structure for holding the training data in memory is a dynamic array, where each element holds an example in the training data. To obtain the count of examples that satisfy a conjunction of attribute-value pairs, a scan of the array is needed,

during which the examples in the array are matched with the attribute-value pairs. Since such counting requests are frequently issued during rule induction, a large number of scans of training data are needed, which greatly impede the learning process, especially when the training set is large.

In this paper, we propose a new data structure, called AV-space, for holding the counts that may be required by a rule induction process. The AV-space is built with one scan of the training data. After it is built, the learning algorithm does not have to access the training data to obtain the counts of examples. A count can be obtained by traversing part of the AV-space, which is much faster than scanning the training data. Our experiments confirm that the use of the AV-space leads to much faster rule induction.

2 Related work

As large data sets become easily available in the real world, computational efficiency has become an important issue in data mining. Much effort has been spent on speeding up data mining algorithms. In the area of association rule mining, a number of algorithms, such as FP-growth [6] and Partition [12], have been proposed to accelerate the mining process. For example, FP-growth compresses a large transaction data set into a compact tree structure that is complete for frequent pattern mining. In the area of regression, Moore *et al.* investigated how kd-trees with multiresolution cached regression matrix statistics can enable very fast locally weighted and instance based regression [10]. In decision tree learning, a few algorithms, such as RainForest [5], were proposed that used efficient data structures to speed up decision tree learning. In the area of classification rule learning, little work has been done on using efficient data structures to speed up the learning process. An exception is ADtree (All-Dimensions Tree) [2][7][9]. An ADtree is a tree structure that caches sufficient statistics for quick counting of training examples. It can be used to accelerate Bayes net structure finding algorithms, rule learning algorithms and feature selection algorithms. An example of an ADtree is shown in Figure 1, which represents the data set shown in the bottom right hand corner of the figure. There are two types of nodes in the tree, *ADtree* node and *Vary* node. The root of the tree is an *ADtree* node containing the total number of the training examples representing all the attributes (* representing any

---

[*] Department of Computer Science and Engineering
York University, Toronto, Ontario, Canada, M3J 1P3
{lwang, aan}@cse.yorku.ca

values). The root is partitioned by a bunch of *Vary* nodes, each of which represents an attribute. On the second level, each *Vary* node contains a set of *ADtree* nodes. Each represents a conjunction of attribute-value pairs, such as $\langle a_1=1 \rangle AND \ \langle a_2=* \rangle AND \langle a_3=* \rangle$. To save memory, the ADtree removes the nodes with count zero, and for each *Vary* node it replaces the *ADtree* child node with the most common value (*MCV*) with a node that stores a *NULL* value. An ADtree can be built by one scan over the data set.
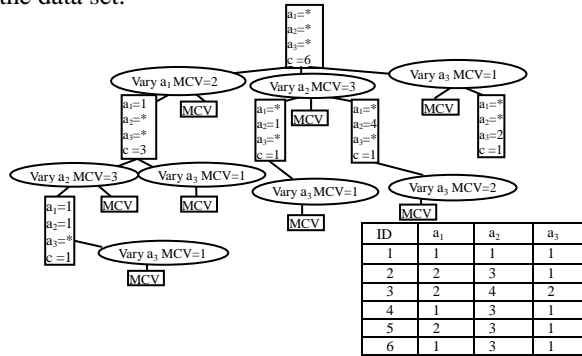

Figure 1. The ADtree of an example dataset

The ADtree can be used to answer counting queries about the data set. For example, we can obtain the number of examples satisfying $\langle a_1=2 \rangle \ AND \ \langle a_3=1 \rangle$ with a specially-designed recursive algorithm [2]. The advantage of using an ADtree is that there is no need to match the training data with the query. However, the ADtree has the following two problems when it is used with a sequential-covering algorithm. First, since the ADtree always returns a count from the entire data set, a problem arises when a rule is made more specific during the rule generation process. Rules with more attribute-value pairs take more time to evaluate. Second and more importantly, the ADtree cannot handle the *rule tiling* problem. In a sequential-covering algorithm, once a rule is generated, the examples that match the rule should be removed from the training set and then the mining of additional rules for a class is based on the remaining examples. The ADtree lacks a mechanism for subtracting the count that represents the number of the examples matched by the generated rule. An ADtree cannot update these counts during the rule induction process, since the sub-trees rooted by MCV nodes cannot be reconstructed. A new ADtree would have to be built from the remaining training set, ignoring the examples covered by the generated rules, in order to learn more rules for a class. To overcome these problems, we propose AV-space.

3 Basic Concepts and Background

In classification rule learning, the training examples are described by a set of attributes. One of the attributes, called *class attribute*, describes the class membership of the examples. The other attributes are called *condition attributes*. A condition attribute can be symbolic or continuous. In most classification rule learning algorithms,

continuous attributes are discretized into symbolic ones before rule induction begins. In this paper, we assume that the continuous attributes have been discretized.

3.1 Conjunctive rule

A rule induction algorithm learns a set of conjunctive rules for each class. A *conjunctive rule* is an if-then rule whose antecedent contains a conjunction of condition attribute-value pairs and whose consequent indicates a class. An example of conjunctive rule is *if $\langle Temperature \neq normal \rangle$ AND $\langle Headache = yes \rangle$ then $\langle Flu = yes \rangle$.*

3.2 Conjunctive counting query

A *conjunctive counting query* is a request for obtaining the number of the examples in a data set that satisfy a conjunction of attribute-value pairs. In this paper, a conjunctive counting query is represented as

$$\langle a_i=v_{i1} \text{ or } v_{i2} \text{ or } ...or \ v_{ik_i} \rangle \ AND \ \langle a_j=v_{j1} \text{ or } v_{j2} \text{ or } ...or \ v_{jk_j} \rangle$$
$$AND \ ... \ AND \ \langle a_m=v_{m1} \text{ or } v_{m2} \text{ or } ...or \ v_{mk_m} \rangle,$$

where a_i, a_j, ... a_m are attributes and v_{xy}'s are attribute values. Here, an attribute-value pair may contain more than one value, i.e., a disjunction of multiple values[1]. An example of a conjunctive counting query is $\langle color = blue$ or *red* or *green*$\rangle$ AND $\langle height=medium \rangle$.

3.3 Sequential-covering algorithms

Most of the rule induction algorithms employ the *sequential covering* technique. In general, a sequential-covering algorithm works as follows:

1. To learn a set of rules for class C_i, divide the training data into examples that belong to C_i (positive examples) and examples that do not belong to C_i (negative examples);
2. Learn a conjunctive rule that covers as many positive examples as possible and as few negative examples.
3. After a conjunctive rule is learned, remove the examples that are covered by this rule.
4. Repeat steps 2 and 3 as long as there are positive examples that are not covered by the generated rules or some other criterion is met.
5. Repeat the above process for all other classes and learn a rule set for each of them.

To learn a single conjunctive rule in step 2, a general-to-specific search can be conducted. That is, a rule can be learned by initializing its antecedent to be empty and then iteratively adding an attribute-value pair that optimizes an objective function until it covers only positive examples or another criterion is met. After a rule is generated, post-pruning is usually conducted to prevent the rule from overfitting the data. In post-pruning, some of the attribute-value pairs in the rule are removed if their removal can result in better performance according to a criterion.

---

[1] Actually, we also allow a query to contain an attribute-value pair with an $\neq$ operator, such as $\langle a_i \neq v_2 \rangle$. Such an attribute-value pair can be transformed into an attribute-value pair with an "=" operator and multiple "or" values. For example, assuming the value set of a_i is $\{v_1, v_2, v_3\}$, attribute-value pair $\langle a_i \neq v_2 \rangle$ can be transformed into $\langle a_i = v_1 \ OR \ v_3 \rangle$.

3.4 ELEM2 heuristics

ELEM2 is a sequential-covering algorithm [1]. We use it as a test bed to evaluate the AV-space. Below we describe three major heuristics used in ELEM2.

Firstly, ELEM2 uses a hill-climbing general-to-specific search method when generating a conjunctive rule. It uses a significance function to select an attribute-value pair during the search. Let t be a candidate attribute-value pair and S be the set of examples covered by the already-selected attribute-value pairs for the current rule. The significance value (also referred to as weighted relative accuracy [8]) of t with respect to a class C and the set S is:

$$SIG_{C,S}(t) = P(t)\,(P(C/t) - P(C)),$$

where the probabilities in the formula are estimated from the set S. During an attribute-value pair selection, dynamic grouping of attribute-values can be performed. For example, if $\langle color=red \rangle$ is selected, other values of $color$, such as $\langle color=green \rangle$, can be selected as well if its significance value on the subset of S not covered by $\langle color=red \rangle$ is no less than $SIG_{C,S}(\langle color=red \rangle)$. This results in the generation of pair $\langle color=red\ or\ green \rangle$.

Secondly, ELEM2 defines an unlearnable region (*ULR*) for each class to handle inconsistent examples. The *ULR* is used in one of the stopping criteria in rule induction. Let's partition the training set into subsets: X_1, X_2, ..., X_m, so that each X_i ($1 \le i \le m$) contains examples that are identical in terms of condition attribute values. We define the *negative region* of class C as:

$$NEG(C) = \bigcup_{P(C|X_i) \le P(C)} X_i$$

The *unlearnable region* (*ULR*) of class C is defined as the set of positive examples in $NEG(C)$. During ELEM2's rule induction for class C, if the positive examples not covered by the already-generated rules for C belong to the *ULR* of C, the induction process for class C is stopped. This prevents ELEM2 from learning from inconsistent examples that do not provide positive classification gain.

Thirdly, after a conjunctive rule is generated, ELEM2 may post-prune the rule by removing some of the attribute-value pairs in the rule. For this purpose, ELEM2 uses a rule quality measure, evaluated on the training set, to determine which attribute-value pairs are removed from the rule. Assume R is a rule for class C (i.e., R predicts C). The quality of R is a log odds ratio, defined as

$$Q(R) = \log \frac{P(R \mid C)(1 - P(R \mid \overline{C}))}{P(R \mid \overline{C})(1 - P(R \mid C))},$$

where $P(R/C)$ is the probability that an example in the training set satisfies R given that the example belongs to C, and $P(R \mid \overline{C})$ is the probability that an example satisfies R given that it does not belong to C. In ELEM2's post-pruning process, an attribute-value pair is pruned if its removal does not decrease the rule quality.

3.5 Major computation in rule induction

From the above description, we can see that the major computation in the rule induction process is the estimation of probabilities from training data. To estimate a probability, we need to count the examples satisfying certain conditions. For example, assuming that rule R is "*if* $\langle a_1 = 2 \rangle$ AND $\langle a_3=1 \rangle$ *then* $\langle class = C \rangle$", $P(R|C)$ in $Q(R)$ can be estimated as the number of examples satisfying $\langle a_1 = 2 \rangle$ AND $\langle a_3=1 \rangle$ AND $\langle class = C \rangle$ divided by the number of examples satisfying $\langle class = C \rangle$. The requests for obtaining such numbers are *conjunctive counting queries*. Another observation is that the probabilities may need to be estimated from different subsets of the training data depending on the stage of sequential-covering rule induction. For example, when learning the first conjunctive rule for a class C, to select the first attribute-value pair, $P(t)$ in $SIG_{C,S}(t)$ is estimated from the whole training set. However, when learning subsequent rules for C, $P(t)$ should be estimated from the training examples not covered by the already learned rules for C. This is referred to as the *rule tiling* problem.

4 AV-Space

The objective of building an AV-space is to store the counts of examples so that conjunctive counting queries can be quickly answered without accessing the training data. To accommodate the *rule tiling* nature of sequential-covering algorithms, the counts stored in an AV-space should be dynamically updated as the induction process progresses.

4.1 The structure of AV-space

An AV-space consists of two components: an AV-tree and an AVC-group. Figure 2 shows the structure of an AV-space, where the table on the left is the AVC-group and the rest is the AV-tree.

4.1.1 AV-tree

Let $\{a_1, a_2, ..., a_M\}$ be the set of M attributes in the data set, where a_M is the class attribute. An AV-tree contains $M+1$ levels, where the root is at level 0 and each of the other levels corresponds to an attribute. For example, level 1 corresponds to attribute a_1, level 2 to a_2 and level M to the class attribute a_M. Each node of the AV-tree represents a conjunction of attribute-value pairs. The root represents an empty conjunction that contains no attribute-value pair. Below the root, a node at level i represents a conjunction of i attribute-value pair(s). For example, a node at level 1 represents one attribute-value pair from attribute a_1, and a node at level 2 represents a conjunction of two attribute-value pairs. We use C_i to represent a node on level 1, where C_i corresponds to the ith value of a_1. On level 2, a node $C_{i,j}$ represents a conjunction of two attribute-value pairs from a_1 and a_2: $\langle a_1 = i$th value of $a_1 \rangle$ AND $\langle a_2 = j$th value of $a_2 \rangle$, where $\langle a_1 = i$th value of $a_1 \rangle$ is from the node's parent on level 1. Similarly, a node $C_{i,j,k}$ on level 3 represents a conjunction of three attribute-values pairs from a_1, a_2 and a_3: $\langle a_1 = i$th value of $a_1 \rangle$ AND $\langle a_2 = j$th value of $a_2 \rangle$ AND $\langle a_3 = k$th value of $a_3 \rangle$. In general, a node $C_{i1, i2, ..., in}$ on level n

Figure 2. AV-space

represents a conjunction of n attribute-value pairs from n attributes: $\langle a_1 = i_1$th value of $a_1 \rangle$ AND ......AND $\langle a_n = i_n$th value of $a_n \rangle$. Each node of the tree stores three counts:

- *Family count.* The family count of a node is the total number of examples that satisfy the attribute-value pair conjunction represented by the node in the *original* data set. Since the root contains an empty conjunction, its family count is the total number of examples in the original data set.

- *Remaining count.* The remaining count of a node is the number of examples not covered by the rules generated so far for the current class, but satisfying the attribute-value pair conjunction represented by the node. The remaining count of the root is the total number of examples not covered by the already-generated rules.

- *AVP count.* The AVP count of a node is the number of examples not covered by the already-generated rules for the current class but satisfying both the attribute-value pair conjunction represented by the node and the conjunction of attribute-value pairs selected so far for the rule being learned at the moment. The AVP count of the root is the total number of examples that are not covered by the already-generated rules for the current class but covered by the attribute-value pairs selected so far for the rule being learned.

To save space, the nodes with the zero family count are not created in the AV-tree. That is, only the combinations of attribute-value pairs that appear in the training data are represented in the tree. An example of an AV-space is depicted in Figure 3, which is built from the data set shown in Figure 1. The family count of a node never changes once the tree is built, since it represents the counts of examples in the original training set. The remaining count of a node remains static during the induction of a single conjunctive rule, but its value may change after a conjunctive rule is generated to reflect the removal of examples covered by the newly generated rule. The AVP count of a node changes dynamically whenever a new attribute-value pair is selected during the generation of a single conjunctive rule to reflect the count of the examples covered by the rule being learned.

Figure 3. An example AV-space

4.1.2 AVC-group

The second component of an AV-space is an AVC-group (standing for Attribute Value Count group). Each element of the AVC-group corresponds to an attribute-value pair (AVP) that is present in the training data. The number of elements in the AVC-group is $\sum_{i=1}^{M} k_i$, where M is the number of attributes and k_i is the number of unique values of the ith attribute. An element of the AVC-group contains a *counting set* and a *pointer*. The *counting set* consists of a *family count*, a *remaining count* and an *AVP count*. Their definitions are similar to those for the tree nodes. The family count for pair $\langle a_i = v_{ij} \rangle$ is the number of the examples in the original training data satisfying $\langle a_i = v_{ij} \rangle$. The remaining count is the number of the examples satisfying $\langle a_i = v_{ij} \rangle$, but not covered by the already-generated rules for the current class. The AVP count is the number of the examples that are not covered by the already-generated rules for the current class, but satisfy both $\langle a_i = v_{ij} \rangle$ and the conjunction of attribute-value pairs selected so far during the generation of the current conjunctive rule. These counts are updated exactly in the same way as the counts stored in a tree node are updated.

The *pointer* in the element corresponding to $\langle a_i = v_{ij} \rangle$ serves as a header to a linked list that connects all the nodes on the ith level of the AV-tree that contain $\langle a_i = v_{ij} \rangle$ in their corresponding conjunctions of attribute-value pairs. The purpose of this list is to be able to easily locate the nodes that contain a specific attribute value pair. An-

other advantage of the AVC-group can be seen during rule induction. A rule is learned sequentially by selecting one attribute-value pair at a time. An evaluation function chooses a pair based on a criterion that needs the number of the examples not covered by the already-generated rules but covered by the pairs selected so far for the rule being learned. This is exactly the AVP count of a single attribute-value pair which can be obtained by a query to the AVC-group (without accessing the tree).

Please note that in the implementation of the AV-space, there is no need to put the attribute-value pairs in the AVC-group or AV-tree. Given an attribute-value pair, the index to its corresponding AVP-group element can be easily computed if we arrange the attributes and their values in an order. This allows direct access to an element of the AVC-group.

4.2 Building an AV-space

With M attributes and N examples in a dataset, we can pay one time cost, O(MN), to build an AV-tree. Below is the algorithm for building an AV-space.

Algorithm: Build_AV-space
Input: A data set (D) with M nominal attributes, a_1, a_2, …a_M, where a_M is the class attribute; the number of unique values, k_i, for attribute a_i ($i=1, …, M$).
Output: An AV-space for D.
Method: Create an AVC-group with $k_1 + k_2 +…+ k_M$ elements and initialize all of its counts to 0;
Create the root node pointed by *root*, initialize its three counts to 0;
Create k_1 child node pointers in the root, and initialize the pointers to NULL;
For each example e in D do
  Increment the three counts in *root* by 1;
  node←root;
  For each attribute-value $<a_i=v_{ij}>$ (in the order from a_1 to a_M) in e do
    currentNode←node;
    *node←*addNode(a_i, v_{ij}, k_i, *currentNode*);
  End
End

Algorithm: addNode
Input: Attribute a_i, the jth value v_{ij} of a_i, the number k_i of unique values of a_i, the current node *currentNode*.
Output: The tree node associated with $\langle a_i=v_{ij}\rangle$
Method: Increment by 1 the three counts in the element of the AVC-group corresponding to $\langle a_i=v_{ij}\rangle$;
If the jth child node pointer of *currentNode* is NULL, then
  Create a new node, *newNode*;
  Set the three counts in *newNode* to 1;
  Set the jth child node pointer of *currentNode* to *newNode*;
  Link this node to the list headed by $\langle a_i=v_{ij}\rangle$ in the AVC-group;
Else
  Increment the three counts of the jth child node of *currentNode* by 1;
Return the jth child node pointer of *currentNode*;

The algorithm first creates the AVC-group and the root of the AV-tree. It then scans the data set. For each attribute-value pair $\langle a_i=v_{ij}\rangle$ in the first example, it creates a node at level i by calling the addNode procedure. These M nodes form a branch of the tree. Then, for each attribute-value pair $\langle a_i=v_{ij}\rangle$ of the next example, it either increases the counts of a tree node or adds a new node at level i by calling the addNode procedure, depending on whether the corresponding node existed or not. When a new node is added, it is added to the corresponding linked list of the AVC-group. The counts in the AVC-group are also incremented for each attribute-value in an example. Note that all the three counts in a tree node or in an element of the AVC-group have the same value after the AV-space is initially built with this algorithm.

4.3 Query Answering with an AV-space

With an AV-space, we can answer a conjunctive counting query efficiently without accessing the original training examples. Given a query, the AV-space needs to know which count (family, remaining or AVP count) should be used to answer the query. Below are three situations where different counts are used:

- During the attribute-value pair selection for generating a conjunctive rule, AVP counts are used to answer a query about how many examples satisfy a conjunction of attribute-value pairs.
- In the post-pruning procedure of some rule induction algorithms (such as ELEM2), a rule is evaluated over the entire training data. In this case, the family counts are used to answer a query.
- After a conjunctive rule is generated in a sequential-covering algorithm, some of the remaining counts in an AV-space should be updated. In order to update the remaining counts, we transform the newly-generated rule into a conjunctive counting query, which will be described in Section 4.4. To answer such a query, the remaining counts need to be used.

The algorithm for answering a query with an AV-space is shown below. One of its input parameters is *CountType*, which specifies which count should be used to answer the query.

Algorithm: Get_Count
Input: *AVS*: an AV-space of a data set;
Query: a conjunction of attribute-value pairs, sorted according to their attributes;
CountType: the *type* of the counts (such as AVP counts) used to answer the query.
Output: The number of examples satisfying *Query*, stored in a global variable *Count*.
Method: Initialize *Count* to 0;
Let *ElementSet* be the set of elements in the AVC-group of AVS whose attribute-value pair satisfies the first pair of *Query*;

```
For each element in ElementSet
    If there is only one attribute-value pair in Query then
        Increment Count by the count of CountType in
            the element;
    Else
        If the count of CountType in the element is not 0,
        then
            Get the pointer to the node list of this element;
            Set index to 2; //index pointing to Query
            For each node in the list
                For each child node of this node
                    If this child is not NULL , then
                        GetNodeCount(child, CountType,
                            Query, index);
                End
            End
```

```
                    Get_Node_Count
                    node: a tree node;
Algorithm:          CountType: the type of the counts (such as AVP counts)
Input:                  used to answer the query;
                    Query: a conjunction of attribute-value pairs, sorted
                        according to their attributes;
                    index: the index of the current attribute-value pair in
                        Query
                    None. The found count is added to the global variable
                    Count.
Output:             If index exceeds the number of attribute-value pairs in
                        Query, then
Method:                 Return;
                    If the count of CountType in node is 0, then
                        Return;
                    If the attribute corresponding to node is not the attribute
                        in the indexth pair of Query, then
                        For each child node of this node
                            If this child is not NULL, then
                                Get_Node_Count(child, CountType, Query, in-
                                    dex);
                        End
                    Else
                        Let ValueSet be the set of values in the indexth pair of
                            Query;
                        If the value corresponding to node² is contained in
                            ValueSet, then
                            If the indexth pair is the last pair in Query, then
                                Increment the Count by the count of
                                    CountType of node;
                                Return;
                            Else
                                For each child node of this node
                                    If this child is not NULL, then
                                        Get_Node_Count(child, CountType,
                                            Query, index+1);
                            End
```

The AV-space answers a query in a top-down fashion. Given a query, the *Get_Count* procedure first finds the AVC-group elements whose attribute-value pair satisfies the first attribute-value pair in the query[3]. Then, for each of these elements, if the query contains only one attribute-value pair, the count specified by *CountType* in the element of AVC-group is added to the global variable *Count*; otherwise, if the specified count of the element is not zero, the procedure goes through the nodes in the linked list headed by the element as follows. For each child node of each node on the list, if the child node is not null, it calls the *Get_Node_Count* procedure to obtain the counts of the examples that satisfy the query and are covered by the subtree rooted at this child node. The child node is passed as the first parameter (named *node*) to the Get_Node_Count procedure. The last parameter of the procedure is the *index* that points to the next attribute-value pair to be processed in the query[4]. The Get_Node_Count procedure first checks whether the attribute corresponding to *node* matches the attribute in the *index*th pair of the query. If not, it recursively calls the Get_Node_Count procedure with each of *node*'s non-null child nodes. If yes, it checks whether the value corresponding to *node* matches the value(s) in the *index*th pair of the query. If yes, it then checks whether the *index*th pair of the query is the last attribute-value pair of the query. If yes, the count of the node specified by *CountType* is added to the global variable *Count*. If the *index*th pair is not the last pair in the query, it recursively calls the Get_Node_Count procedure with each of *node*'s non-null children and an incremented *index* value. The final result is stored in the global variable *Count*.

For example, suppose that we would like to answer the query "⟨*a1=2*⟩AND⟨*a3=1*⟩" with the AV-space shown in Figure 3. Assume that the Family counts are required to answer this query. The *Count* variable is initiated to 0. We start from the AVC-group element corresponding to ⟨*a1=2*⟩ since ⟨*a1=2*⟩ matches the first attribute-value pair in the query. Since ⟨*a1=2*⟩ is not the last attribute-value pair of the query, we follow the list headed by this element to node C_2 and exam C_2's two child nodes, $C_{2,3}$ and $C_{2,4}$, in turn by calling the Get_Node_Count procedure. Since the corresponding attribute (i.e., a_2) in $C_{2,3}$ does not match the attribute (i.e., a_3) in the second attribute-value pair in the query, the Get_Node_Count procedure is called recursively with the child node of $C_{2,3}$, which is $C_{2,3,1}$. Since the attribute and value corresponding to $C_{2,3,1}$ both match the second attribute-value pair in the query, the Family count in $C_{2,3,1}$ is added to variable *Count*, resulting in 2. For the other child, $C_{2,4}$, of C_2, since none of its descendents matches with the second attribute-value pair, no count is added to variable *Count*. Therefore, the result for this query is 2.

Please note that if the query contains only one attrib-

---

[2] The value corresponding to a node at level *i* is the value of the *i*th attribute in the conjunction of attribute-value pairs represented by the node.
[3] Note that an attribute-value pair in the query may contain a disjunction of values, as defined in Section 3.2. Therefore, there may be more than one element of the AVC-group match the query. Also, finding such elements can be done quickly through direct access to the array that

stores the AVC-group. The attribute and values in an attribute-value pair of the query can serve as the indexes to the elements of the array.
[4] The index value for the first attribute-value pair in the query is 1, for the second it is 2, and so on.

ute-value pair, we only need to visit one or more elements in the AVC-group to obtain the answer to the query, without accessing the AV-tree. Thus, answering such a query with an AV-space is very fast. In sequential-covering algorithms, a conjunctive rule is generated by sequentially selecting attribute-value pairs. We evaluate each of the possible attribute-value pairs according to a selection criterion. To evaluate a pair, we need to obtain the number of examples that are not covered by the previously generated rules but are covered by the pair and the already-selected pairs. To obtain this number, we only need to issue a query with a single attribute-value pair to the AV-space to obtain the AVP-count from an element in the AVC-group because the AVP-counts in the AVC-group represent such numbers. That is, we do not need to form a query that concatenates the already-selected pairs with the pair being evaluated. This greatly speeds up the process for attribute-value pair selection, and is an improvement over the ADtree algorithm.

With respect to answering a general query, in the worst case, the data set contains all the possible combinations of the attribute values and the query contains all the attributes in the data set. To answer such a query from an AV-space built from such a data set, we need to traverse the entire tree. If there are M attributes in a data set and each attribute has k values, the time complexity for answering a query in the worst case is $O(k^M)$.

4.4 Updating the AV-space

An important feature of an AV-space is that the counts stored in the AV-space are updated during the rule induction process. There are three situations where the counts need to be updated, which are described below.

4.4.1 Updating AV-space after selection of an attribute-value pair

During a single conjunctive rule generation, whenever an attribute-value pair is selected and added to the rule, some of the AVP counts in the AV-space need to be updated so that the AVP counts will not include the examples not covered by the newly-selected pair. The updating process starts with the AVC-group. Suppose that the newly-selected attribute-value pair is $\langle a_i=v_{ij} \rangle$. For each element E in the AVC-group that corresponds to $\langle a_i=v_{ik} \rangle$ (where $k \neq j$), we do the following:

- Change the AVP-count in E to zero;
- For each AV-tree node, *avnode*, in the linked list headed by E, if the AVP-count of *avnode* is not zero,
    - For each ancestor, *ancenode*, of *avnode*,
        - Reduce the AVP count of *ancenode* by the AVP count of *avnode*;
        - Reduce the AVP count in the AVC-group element that links to *ancenode* by the AVP count of *avnode*;
    - Change the AVP count in *avnode* to zero;

- For each descendent, *descnode*, of *avnode*, if the AVP count of *descnode* is not zero,
    - Reduce the AVP count in the AVC-group element that links to *descnode* by the AVP count of *descnode*;
    - Reduce the AVP count of *descnode* to zero.

An example of updating an AV-space using this procedure is given in Section 4.5. In the best case, only one branch of the tree (from the root to a leaf) is visited during this process. Thus, the best case time complexity is $O(M)$, where M is the number of attributes. In the worst case, almost all the tree nodes are visited, meaning an $O(k^M)$ time complexity if all the possible combinations of attribute values occur in the data set, assuming k is the number of unique values for each attribute. However, this worst case, if it occurs, usually occurs when the first attribute-value pair is selected for a conjuctive rule. Updating the AVP counts becomes faster and faster when more pairs are selected.

4.4.2 Updating AV-space after generation of a single conjunctive rule

After a conjunctive rule is generated, some remaining and AVP counts in the AV-space need to be updated to reflect the removal of the examples covered by the newly-generated rule. The procedure for updating the remaining counts is as follows. Suppose that the newly-generated rule is "*IF* $\langle a_{k1}=v_{k1j1} \rangle$ *AND* $\langle a_{k2}=v_{k2j2} \rangle$ *AND…AND* $\langle a_{kn}=v_{knjn} \rangle$ *THEN* $\langle a_{class}=k \rangle$". We first transform this rule into a conjunctive counting query: $\langle a_{k1} = v_{k1j1} \rangle$ *AND* $\langle a_{k2}=v_{k2j2} \rangle$ *AND…AND* $\langle a_{kn}=v_{knjn} \rangle$ *AND* $\langle a_{class}=k \rangle$, and then call a procedure that is similar to the Get_Count procedure (described in Section 4.3) to obtain the leaf nodes of the AV-tree that satisfy the query. The difference between this procedure and Get_Count is that this procedure returns the nodes that satisfy the query but Get_Count returns the sum of the counts (e.g., AVP counts) of the nodes that satisfy the query. Since the query transformed from a rule contains an attribute-value pair of the class attribute, the nodes that satisfy the query must be leaf nodes. Then, for each leaf node, *leafnode*, returned by the procedure, we do the following:

- Reduce the remaining count in the AVC-group element that links to *leafnode* by the remaining count of *leafnode*;
- For each ancestor, *anode*, of *leafnode*, reduce the remaining count in *anode* by the remaining count of *leafnode*, and also reduce the remaining count in the AVC-group element that links to *anode* by the Remaining count of *leafnode*;
- Set the remaining count of *leafnode* to zero.

After remaining counts are updated, each AVP count in the AV-space is refreshed to be the value of the remaining count in the same node or element. After that,

the AV-space is ready for learning another rule for the current class. The time complexity of this procedure is the same as the one for updating the AVP counts described in Section 4.4.1.

4.4.3 Updating AV-space after learning a set of rules for a class

The third case where an AV-space needs to be updated is when the learning of a set of rules for a class is finished. In order to learn a set of rules for the next class, we need to refresh all the remaining and AVP counts in the AV-space. This updating process is simply to copy the family count in each tree node or AVP-group element to its corresponding remaining and AVP counts. Note that this process only occurs once per class.

4.5 Mining Rules with ELEM2 and AV-Space

In this section, we show how to mine rules with ELEM2 and an AV-space. Let's consider the Flu data set shown in Table 1, where *Headache*, *Pains* and *Temperature* are condition attributes and *Flu* is the class attribute. Below are the steps for generating rules for this data set.

Table 1. Flu dataset

Patient	Headache	Pain	Temperature	Flu
x1	Yes	yes	Normal	No
x2	Yes	yes	High	Yes
x3	Yes	yes	very high	Yes
x4	No	no	high	No
x5	No	yes	normal	No
x6	No	no	high	No
x7	No	yes	very high	Yes
x8	No	no	High	Yes
x9	No	yes	very high	No
x10	No	no	High	No

1. Build an initial AV-space from the data set. The result is shown in Figure 4.
2. Compute the unlearnable region (*ULR*) for class ⟨*Flu=yes*⟩.
2.1. Partition the training data set into subsets so that each subset contains examples with an identical set of condition attribute values. For the Flu data set, there are six such subsets, each corresponding to a tree node at level 3, represented by $C_{i,j,k}$. The partitioning of the training data set takes place when the AV-space is built in Step 1, and thus there is no extra computation time for this step.
2.2. Compute the *ULR* of class ⟨*Flu=yes*⟩ by collecting all the positive examples in the subsets satisfying $P(⟨Flu=yes⟩|subset) \leq P(⟨Flu=yes⟩)$. In this data set, there is only one such example, which corresponds to tree node $C_{1,1,2,2}$. Thus, *ULR* (⟨Flu=yes⟩) = {$C_{1,1,2,2}$}.
3. Generate the first conjunctive rule for class ⟨*Flu=yes*⟩.
3.1. Select the first attribute-value pair by calculating the significance value (*SIG*) of each attribute-value pair and selecting the one with the highest *SIG* value. To compute the *SIG* value of a pair, ⟨$a_i=v_j$⟩, with respect

to class ⟨*Flu=yes*⟩, we need to issue the following three queries on the AVP count to the AV-space:

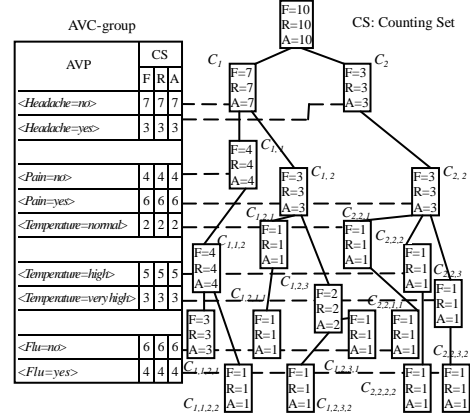- ⟨$a_i=v_j$⟩
- ⟨$a_i=v_j$⟩ AND ⟨*Flu=yes*⟩
- ⟨*Flu=yes*⟩

Figure 4. Initial AV-space for the Flu data set

The first and third queries can be easily answered by looking up the AVC-group in the AV-space. The second one can be answered by traversing the AV-tree starting from the nodes on the linked list headed by the AVC-group element for ⟨$a_i=v_j$⟩ down to the leaf nodes. After computing *SIG*s for all the possible pairs, pair ⟨*Temperature≠normal*⟩ is selected because it has the highest *SIG* value and covers more positive examples than the other pair with the highest *SIG* value.

3.2. After the selection of ⟨*Temperature≠normal*⟩, the AVP counts in the AV-space are updated. For each AVC-group element that does not satisfy ⟨*Temperature≠normal*⟩ (i.e., the element corresponding to ⟨*Temperature=normal*⟩), change its AVP count to zero. Then, for each tree node linked by element ⟨*Temperature=normal*⟩ (such as, $C_{1,2,1}$), reduce the AVP counts in each of its ancestors (such as $C_{1,2}$, C_1, and the root) and in the AVC-group element linking to the ancestor by the amount of the AVP count of the tree node. The resulting AV-space is shown in Figure 5.

3.3. Since the updated AVP count in the AVC-group element corresponding to ⟨*Flu= no*⟩ is not zero, another attribute-value pair needs to be selected to specialize the rule. To select the second pair, the *SIG* value of each candidate pair is computed with respect to the updated AV-space. In this time, ⟨*Pain=yes*⟩ is selected.

3.4. After ⟨*Pain=yes*⟩ is selected, the AV-space is updated. The rule generated so far is "*IF* ⟨*Temperature≠normal*⟩ *AND* ⟨*Pain=yes*⟩ *THEN* ⟨*Flu=yes*⟩". Since it still covers negative examples, a third pair is selected based on the updated AV-space. In this time, the pair ⟨*Head-*

ache=yes⟩ is selected and the AV-space is updated again (see Figure 6). Since the updated AVP count in the AVC-group element for ⟨*Flu=no*⟩ becomes 0 (i.e., the rule does not cover any negative examples), the process for generating a single conjunctive rule stops. The generated rule is "*IF* ⟨*Temperature≠normal*⟩ *AND* ⟨*Pain=yes*⟩ *AND* ⟨*Headache=yes*⟩ *THEN* ⟨*Flu=yes*⟩".
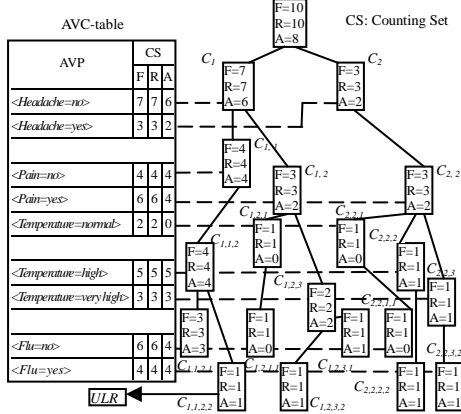
Figure 5. The AV-space after the first AV-pair selection

3.5. Post-prune the generated rule. To calculate the rule quality values in post-pruning, the family counts of the AV-space are used to answer the queries. The first row in Table 2 shows the rule quality value of the un-pruned rule; and the second and third rows show the rule quality values of two rules with one pair removal. The rule in the third row is selected because its rule quality value is no less than that of the unpruned rule. That is, pair ⟨*Pain=yes*⟩ is removed from the generated rule. The rule becomes: *IF* ⟨*Temperature≠normal*⟩ *AND* ⟨*Headache=yes*⟩ *THEN* ⟨*Flu=yes*⟩.

4. After the first rule is generated and pruned, update the AV-space according to the procedure described in Section 4.4.2. The resulting AV-space is shown in Figure 7.

5. With the updated AV-space, we find that there are still some leaf nodes corresponding to ⟨*Flu=yes*⟩ whose remaining counts are not zero and these nodes do not belong to the *ULR* of class ⟨*Flu=yes*⟩. That is, there are still some positive examples that are not covered by the already generated rule(s) and do not belong to the unlearnable region of the current class. Thus, the learning of the second rule for class ⟨*Flu=yes*⟩ starts. The learning process for class ⟨*Flu=yes*⟩ stops until all the positive examples are covered or all the remaining positive examples belong to the *ULR* of the class.

6. Update the AV-space by refreshing the remaining and AVP counts of each node or element using the family count of the node or element. Repeat Steps 2-5 to learn a set of rules for class ⟨*Flu=no*⟩.

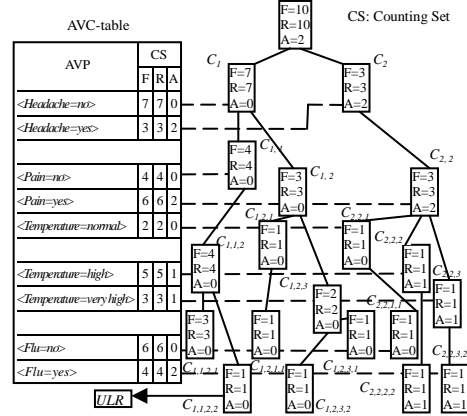**Figure 6.** The AV-space after the third attribute-pair selection

Table 2. Results of post-pruning the generated rule

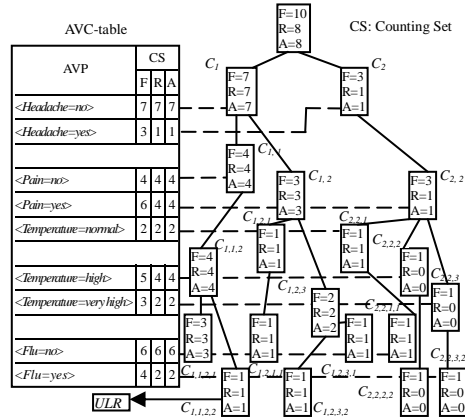| AVP1	AVP2	AVP3	Quality of Rule
⟨Temperature≠normal⟩	⟨Pain=yes⟩	⟨Headache=yes⟩	1.114
⟨Temperature≠normal⟩	⟨Pain=yes⟩		0.932
⟨Temperature≠normal⟩		⟨Headache=yes⟩	1.114

Figure 7. The updated AV-space for the second rule learning

5 Experimental Evaluation

To evaluate the performance of the AV-space, we implemented in Java two versions of ELEM2 that use AV-space and Example-set (to be described below) as the main data structure, respectively. We compare the AV-space-based ELEM2 with the Example-set-based ELEM2 in terms of execution time and memory cost. The experiments are conducted on a 3.0-GHz Pentium PC with a 512M main memory.

5.1 The Data Sets

The data sets used in our experiments were obtained from the UCI Repository [11]. Table 3 describes the data sets. Most of the data sets contain continuous attributes. We discretized continuous attributes so that after discretization, all the attributes are treated as symbolic attributes. Table 3 also shows the number of attribute values and the number of nodes in the AV-tree for each data set.

Table 3. Description of the data sets

Data set	# of examples	# of attributes	# of classes	# of nodes in AV-space	# of attribute values
Flu	10	4	2	20	8
Lenses	24	5	3	63	24
Monks	122	7	2	379	122
Flag	194	27	8	3024	193
Glass	214	10	6	702	129
Balance Scale	625	5	3	1406	625
Credit Screening	653	16	2	3618	604
Breast Cancer	683	10	2	3099	449
Mushroom	5,644	23	2	17,304	5,644
Nursery	12,598	9	5	29,403	12,960
Adult	45,222	15	2	111,259	24,206

5.2 Performance Comparison of AV-space with the Example Set

We compare the performance of the AV-space with that of a dynamic array, the most common data structure used in rule induction systems. A dynamic array stores all the (encoded) training examples in an array structure, where each element of the array stores an (encoded) training example. The size of the array is the number of examples in the data set. We refer to this dynamic array structure as the *Example-set*. The time complexity of loading an Example-set is O(MN), where M is the number of attributes and N is the number of examples in the data set. With an Example-set, it takes O(N) to compute the significance value of an attribute-value pair during attribute-value pair selection.

5.2.1 Comparison in Running Time

We split the running time into the data set loading time and rule induction time. Table 4 shows both the loading time and the rule induction time[5]. Figure 8 depicts the percentage improvement of the AV-space over the Example-set in terms of the rule induction time. The result shows that the AV-space significantly improves the Example-set in terms of both the data set loading time and the rule induction time. The speed acceleration varies among the data sets. The improvement is more significant on large data sets. For example, on Mushroom and Nursery data sets, the percentage improvement in rule induction time is more than 85% and on the Adult data set it is over 65%.

When building an AV-space, only M comparisons are made to load an example with M attributes. To load the data into an Example-set, we need to create a new object (i.e., a new array) when loading an example[6]. This

is the reason why the AV-space has a shorter data loading time. The reason for the AV-space to have a shorter rule induction time is that it does not need to scan the training data during rule induction, while with the Example-set we need to do multiple scans to collect the necessary statistics in the data.

Table 4. Data set loading time and rule induction time (in seconds) in AV-space and Examples-set

Data set	Time Use	AV-space	Example-set	Percentage improvement[7]
Flu	Loading	0.140	0.172	18.6%
	Rule induction	0.031	0.031	0%
Lenses	Loading	0.141	0.141	0%
	Rule induction	0.031	0.032	3.1%
Monks	Loading time	0.204	0.219	6.8%
	Rule induction	0.078	0.109	28.4%
Flag	Loading	0.172	0.203	15.3%
	Rule induction	0.360	0.484	25.6%
Glass	Loading	0.016	0.172	90.7%
	Rule induction	0.219	0.406	46.1%
Balance Scale	Loading	0.031	0.031	0%
	Rule induction	0.516	1.484	65.2%
Credit Screening	Loading	0.187	0.203	7.9%
	Rule induction	0.516	1.156	55.4%
Breast Cancer	Loading	0.172	0.187	8.0%
	Rule induction	0.156	0.422	63.0%
Mushroom	Loading	0.438	1.125	61.1%
	Rule induction	0.297	2.859	89.6%
Nursery	Loading	0.328	0.594	44.8%
	Rule induction	10.110	67.469	85.0%
Adult	Loading	1.313	3.641	63.9%
	Rule induction	76.110	219.109	65.3%

Figure 8. Improvement of AV-space over Example-set on rule induction time

5.2.2 Comparison in Memory Cost

Table 5 shows that on small data sets the AV-space uses more memory than the Example-set, but on large data sets it consumes smaller space.
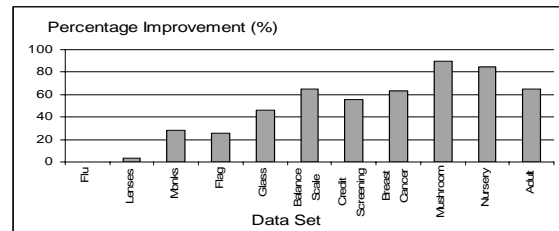
---

[5] The rule induction time on a data set is the time taken for generating all the rules for all the classes except for the adult data set for which we only allow a maximum of 20 rules to be generated per class.
[6] The loading time depends on the programming language used to write the program. We use a vector structure in Java to hold an Example-set in memory. If another language, such as C, is used, we may not need to spend time on creating new objects during data loading, but we

need to scan the data set twice to create a dynamic array. In the first time, we find the number of examples in the data set. In the second time, we load the data into an array whose size equals to the number of examples in the data set. Thus, the loading time with the Example-set may still be much longer than the time with an AV-space.
[7] The percentage improvement is the relative improvement rate expressed as a percentage. The relative improvement rate is the difference between the time the Examples-set based ELEM2 takes and that the AV-space based ELEM2 takes divided by the time the Examples-set based ELEM2 takes.

Table 5. Memory consumption of AV-space and Example-set

Data set	AV-space (bytes)	Example-set (bytes)	AV-space/ Example-set
Flu	484	384	1.26
Lenses	1,308	908	1.44
Monks	7,108	5,104	1.39
Flag	56,428	24,788	2.28
Glass	13,576	11,844	1.15
Balance Scale	25,656	20,272	1.27
Credit Screening	66,352	50,560	1.31
Breast Cancer	57,236	36,616	1.56
Mushroom	313,260	588,328	0.53
Nursery	529,748	622,460	0.85
Adult	2,004,836	3,257,624	0.62

5.2.3 Comparison on Scalability

To see how the performance of the AV-space changes with the size of training data, we conducted another set of experiments with two data sets, Adult and Covtype. The original Adult data set from the UCI Repository contains a training data set (Adult2 with 30162 examples) and a test data set (Adult1 with 15060 examples). We created Adult3 (with 45222 examples) by concatenating the Adult1 and Adult2 data sets[8].

The speed comparisons on the three Adult data sets are shown in Figure 9 and the memory comparisons in Figure 10. The result reveals that the larger the data set, the more significant the improvement of the AV-space over the Example set on the speed of rule induction, and the more reduction on the memory cost.
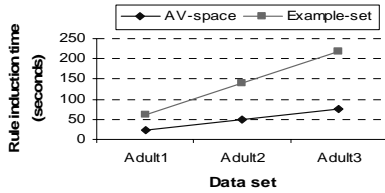

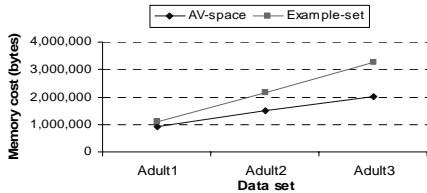**Figure 9.** Speed comparisons on Adult data sets

Figure 10. Memory comparisons on Adult data sets

The Covtype data set is another public data set donated by the US Forest Service. In this experiment, we select 40 condition attributes, Soil_type1 to Soil_type40, and discover relationships between the condition attrib-

utes and the class attribute, the cover type of forest. The data set is sparse, meaning that only a small portion of the possible combinations of attribute values appear in the data set. The original Covtype data set contains 581,011 examples. We created 4 subsets of the data set, containing the first 300,000, 50,000, 10,000, and 2,000 examples respectively. We tested the AV-space and the Example-set on the original data set and the 4 subsets. The result (see Table 6) shows that the AV-space significantly outperforms the Example-set on this large sparse data set on both memory consumption and running time. The larger the data set, the more significant the improvements are.

Table 6. Comparisons on the Covtype data set

Number of examples	Algorithm	Memory (bytes)	Loading time(sec)	Rule induction time(sec)
581,011	AV-space	19,588	12.469	0.109
	Example-set	102,258,976	330.281	187.891
300,000	AV-space	19,500	6.390	0.110
	Example-set	52,801,040	168.735	81.843
50,000	AV-space	18,276	1.156	0.109
	Example-set	8,801,040	26.281	14.750
10,000	AV-space	17,880	0.406	0.141
	Example-set	1,761,040	4.688	3.032
2,000	AV-space	13,468	0.203	0.109
	Example-set	353,040	1.203	0.484

5.3 Comparison between AV-space and ADtree

In Table 7, we compare the AV-space with the AD-tree algorithm on the three adult data sets in terms of the memory cost and the building time. The results for the ADtree algorithm are taken from [9].[9] From Table 9, we can see that both structures can control the growth of memory well, and that the AV-space consumes much less memory than the ADtree, and it takes much less time to build an AV-space than building an ADtree. Since the AD tree cannot handle the rule tiling problem in sequential-covering, it can only be used to learn a single conjunctive rule. To learn an additional rule, another AD-tree has to be built from the data. Thus, its rule induction time is not comparable to that of the AV-space.

Table 7. Comparisons between AV-space and ADtree

Data set	# of Attrib- utes	# of Exam- ples	AV-space		ADtree	
			Memory Cost (M)	Build Time (seconds)	Memory Cost (M)	Build Time (seconds)
Adult1	15	15060	0.9	0.515	7.0	6
Adult2	15	30162	1.5	0.937	10.9	10
Adult3	15	45222	2.0	1.313	15.5	15

6 Analysis of Space Complexity of AV-space

In this section, we analyze the space complexity of

---

[8] The Adult data set used in the experiments described in the previous subsections is the Adult3 data set.

[9] Since we do not have the programs for the ADtree algorithm, we can only make limited comparisons based on the results reported in the ADtree paper. Since a slower machine was used to evaluate the ADtree structure (roughly 15 times slower), we adjust the building times for the ADtree to be 15 times faster than what was reported in [9].

the AV-space in the worst case scenario and compare it with those of the ADtree and the Example-set.

In the worst case, the training data set contains all possible combinations of the attribute values. We assume that N is the number of examples in the training set, M is the number of attributes and k is the number of unique values for each attribute. In this case, the number of AV-tree nodes at level i is k^i. The total number of nodes in the AV-tree is $\sum_{i=1}^{M} k^i$. Assume that the size of a node is four[10], then the size of the AV-tree is

$$4 * \sum_{i=1}^{M} k^i = \frac{4}{k-1} k^{M+1} - \frac{4}{k-1} \qquad (1)$$

We do not take the AVC-group into account because the other structures that we compare the AV-space with also use a structure similar to the AVC-group during rule induction. Equation (1) shows that the size of the AV-space is independent of N, which is usually much larger than M and k. In addition, it is usual that a real-world data set does not contain all the combinations of the attribute values. Thus, the actual size of the AV-tree is usually much smaller than what is shown in Equation (1). Also, when the numbers of unique values vary among attributes, placing the attributes with fewer values at higher levels of the AV-tree consumes less space.

Given N training examples and M attributes, the size of the Example-set is MN. We can prove that in the worst case scenario, when $N > \frac{4}{M(k-1)} k^{M+1}$, the size of the AV-space is smaller than that of an Example-set.

To compare the AV-space with the ADtree structure, we restrict our attention to the case of binary attributes. Given a dataset with M attributes, in the worst case, all the 2^M different examples appear in the training set. The number of AV-tree nodes is $2^{M+1}-1$. In the worst case, the ADtree has no ADnodes with counts of zero. Due to the use of the Most Common Values (MCVs), there is only one ADnode under each Vary node. Thus, in the worst case, the number of ADtree nodes is 2^M, the number of Vary nodes is 2^M-1, and the total number of nodes is $2^{M+1}-1$. Thus, in terms of the number of nodes, the AV-tree is comparable to the ADtree. However, an ADnode contains a set of attribute values (one for each attribute, including the * value), while an AV-tree node stores only three counts (not considering the pointers in both trees). Therefore, the size of an AV-tree can be much smaller than that of an ADtree.

7 Conclusions

Sequential-covering rule induction is one of the ma-

---

[10] Logically, a tree node contains three counts, a set of points to its child nodes and a pointer to its parents. However, in the implementation we only need to store the three counts and one pointer in a node, and the other pointers can be derived. Hence, the size of the node is 4.

jor classification techniques in machine learning. Although many sequential-covering systems are successful in generating accurate classification rules, most of them suffer from the problem of slow induction when the data set is very large. To solve this problem, we proposed the AV-space data structure for caching sufficient statistics of a data set. The AV-space can be built efficiently with one scan of the data set. The AV-space can answer a conjunctive counting query efficiently. The process for updating the AV-space is also efficient. The experimental results showed that the AV-space leads to a significant improvement in the rule induction time and data set loading time. In terms of memory usage, the AV-space consumes less space than the Example-set when the data set is large, especially when the data set is sparse. We also showed that the AV-space consumes less memory than the AD-tree, and it is faster to build an AV-space than building an ADtree. We are currently incorporating the AV-space into other sequential-covering algorithms.

References

[1]. An, A. and Cercone, N. ELEM2: A Learning System for More Accurate Classifications, Proc. of the 12th Canadian Conf. on Artificial Intelligence, Vancouver, Canada, 1998.

[2]. Anderson, B., Moore, A.: ADtrees for Fast Counting and for Fast Learning of Association Rules. Proc. of the 4th Intl. Conf. on Knowledge Discovery in Data Mining, 1998.

[3]. Cendrowska, J.: PRISM: An algorithm for inducing modular rules. International Journal of Man-Machines Studies, 27:349-370, 1987.

[4]. Clark, P., Niblett, R: The CN2 Induction Algorithm. Machine Learning3:261-284, 1989.

[5]. Gehrke, J., Ramakrishnan, R., Ganti, V.: RainForest – A Framework for Fast Decision Tree Construction of Large Datasets. Proc. of the 24th Intl. Conf. on Very Large Databases, 1998.

[6]. Han, J., Pei, J. and Yin, Y. 2000. Mining Frequent Patterns without Candidate Generation, Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, 2000.

[7]. Komarek, P., Moore A.: A Dynamic Adaptation of AD-trees for Efficient Machine Learning on Large Data Sets. Proceedings of the Seventeenth International Conference on Machine Learning, 2000.

[8]. Lavra, N., Flach, P. and Zupan, B. Rule Evaluation Measures: A Unifying View, Proc. of the 9th Intl. Workshop on Inductive Logic Programming, 1999.

[9]. Moore, A., Lee, M. S.: Cashed Sufficient Statistics for Efficient Machine Learning with Large Datasets. Journal of Artificial Intelligence Research, 8, 1998.

[10].Moore, A.W., Schneider, J. And Deng, K. Efficient Locally Weighted Polynomial Regression Prediction, Proc. of the International Conference on Machine Learning, 1997.

[11].Murphy, P. M., Aha, D. W. UCI Repository of Machine Learning Databases, 1994.

[12].Savasere, A., Omiecinski, E. and Navathe, S. An Efficient Algorithm for Mining Association Rules in Large Databases, Proc. of the 21st Intl. Conf. on Very Large Databases, 1995.