ADtrees for Fast Counting and for Fast Learning of Association Rules

Brigham Anderson
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
brigham@ri.cmu.edu

Andrew Moore
Carnegie Mellon University
5000 Forbes Ave, Pittsburgh, PA 15213
awm@cs.cmu.edu

Abstract

The problem of discovering association rules in large databases has received considerable research attention. Much research has examined the exhaustive discovery of all association rules involving positive binary literals (e.g. Agrawal et al. 1996). Other research has concerned finding complex association rules for high-arity attributes such as CN2 (Clark and Niblett 1989). Complex association rules are capable of representing concepts such as "PurchasedChips=True and PurchasedSoda=False and Area=NorthEast and CustomerType=Occasional ⇒ AgeRange=Young", but their generality comes with severe computational penalties (intractable numbers of preconditions can have large support). Here, we introduce new algorithms by which a sparse data structure called the ADtree, introduced in (Moore and Lee 1997), can accelerate the finding of complex association rules from large datasets. The ADtree uses the algebra of probability tables to cache a dataset's sufficient statistics within a tractable amount of memory. We first introduce a new ADtree algorithm for quickly counting the number of records that match a precondition. We then show how this can be used in accelerating exhaustive search for rules, and for accelerating CN2-type algorithms. Results are presented on a variety of datasets involving many records and attributes. Even taking the costs of initially building the ADtree into account, the computational speedups can be dramatic.

Problem Definition

If-then rules are expressive and human readable representations of learned hypotheses, so finding association rules in databases is a useful undertaking. The rules one might search for could be of the form "if workclass=private and education=12+ and maritalstatus=married and capitalloss=1600+, then income ⇒ 50K+ with 96% confidence." Association rules can be quite useful in industry. For instance, the above example could help target income brackets.

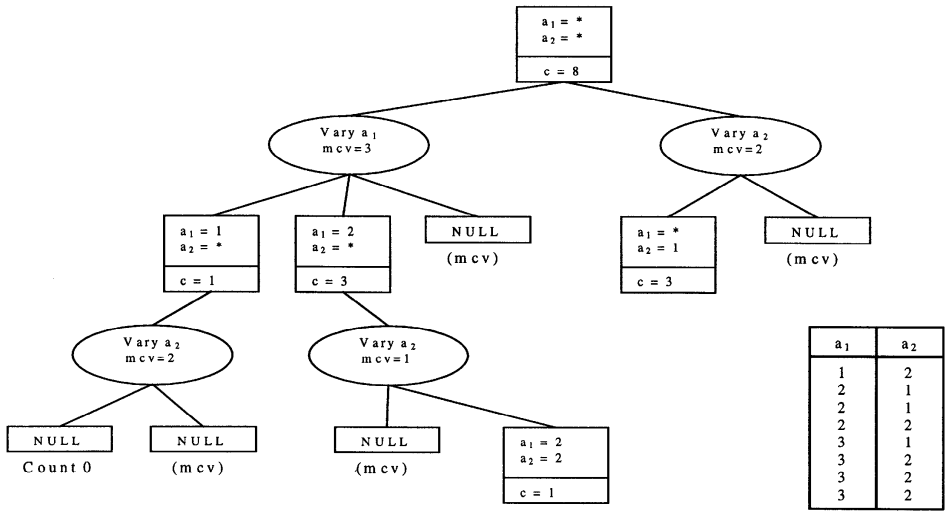
Consider a database of R records with symbolic attributes. A database could be, for example, a list of loan applicants where each entry has a list of attributes such as type of loan, marital status, education level, and income range, and each attribute has a value. A record thus has M attributes, and is represented as a single vector of size M, each element of which is symbolic. The attributes are called $a_1, a_2, \ldots a_M$. The value of attribute a_i in a record is represented as a small integer lying in the range $\{1,2,\ldots n_i\}$ where n_i is called the *arity* of attribute *i*.

In our definition of association we followed (Agrawal, et al. 1996). Their definition of an association rule is a conjunction of attributes implies a conjunction of other attributes. Here, the terminology is slightly different; we define a *literal* as an attribute-value pair such as "education = masters". Let L be the set of all possible literals for a database. An *association rule* is an implication of the form $S1 \Rightarrow S2$, where $S1, S2 \subset L$, and $S1 \cap S2 = \emptyset$. S1 is called the *antecedent* of the rule, and S2 is called the *consequent* of the rule. We thus denote association rules as an implication of sets of literals. An example of an association rule is "gender=male and education=doctorate ⇒ maritalstatus=married and occupation=prof-specialty".

Each rule has a measure of statistical significance called *support*. For a set of literals $S \subset L$, the *support* of S is the number of records in the database that match all the attribute-value pairs in S. Denote by *supp*(S) the *support* of S. The support of the rule $S1 \Rightarrow S2$ is defined as *supp*($S1 \cup S2$). Support is a measure of the statistical significance of a rule. A measure of its strength is called *confidence*, and is defined as the percentage of records that match S1 and S2 out of all records that match S1. In other words, the support of a rule is the number of records that both the antecedent and consequent literals match. The confidence is the percentage that the supporting records represent out of all records in which the antecedent is true.

This paper considers the problem of mining association rules to predict a user-supplied target set of literals S2. The objective is to find rules of the form S1⇒S2 that maximize confidence while keeping support above some user-specified minimum (*minsupp*). One version of this procedure is to return the best *n* rules encountered, or perhaps all rules above a certain confidence.

Generation of such rules requires calculating large numbers of rule confidences and supports. Rule evaluation thus requires two calculations, *supp*(S1) and *supp*($S1 \cup S2$). These two numbers give both the support and the



a_1	a_2
1	2
2	1
2	1
2	2
3	1
3	2
3	2
3	2

Figure 1: A sparse ADtree built from the dataset in the bottom right. The most common value for a_1 is 3, and so the $a_1 = 3$ subtree of the Vary a_1 child of the root node is NULL. At each of the Vary a_2 nodes the most common child is also set to NULL (which child is most common depends on the context.)

confidence of the rule S1⇒S2. One method for calculating a *supp*(S) is to run through every relevant record and count the number of matches. Another method is to use some way of caching statistics that allows calculating these numbers directly, such as an ADtree (described below). A third possibility is to build all queries having adequate support with sequential passes through the dataset (Agrawal et al. 1996). This is very effective if only positive binary literals are being found, but if negative literals are also required the number of rule sets found will be intractably large: $O(2^M)$.

ADtree Data Structure

If we are prepared to pay a one-time cost for building a caching data structure, then it is easy to suggest a mechanism for doing counting in constant time. For each possible query, we precompute the count. For a real dataset with more than ten attributes of medium arity, or fifteen binary attributes, this is far too large to fit in main memory.

We would like to retain the speed of precomputed counts without incurring an intractable memory demand. That is the purpose of ADtrees. An "ADNODE" (shown as a rectangle in Figure 1) has child nodes called "Vary nodes" (shown as ovals).

Each ADNODE represents a query and stores the number of records that match the query. The "Vary a_j" child of an ADNODE has one child for each of the $arity_j$ values of attribute a_j. The kth child represents the same query as "Vary a_j"'s parent, with the additional constraint that $a_j = k$.

Although drawn on the diagram, the description of the query (e.g., $a_1 = *$, $a_2 = 1$) is not explicitly recorded in the ADNODE. The contents of an ADNODE are simply a count and a set of pointers to the "Vary a_j" children. The contents of a "Vary a_j" node are a set of pointers to ADNODEs. Notice that if a node ADN has "Vary a_i" as its parent, then ADN's children are *"Vary a_{i+1}", "Vary a_{i+2}", ... "Vary a_M"*. It is not necessary to store Vary nodes with indices below i+1 because that information can be obtained from another path in the tree.

As described so far, the tree is not sparse and contains every possible count. Sparseness is easily achieved by storing a NULL instead of a node for queries that match no records. All of the specializations of such a query also have a count of zero and they will not appear anywhere in the tree. This helps, but not significantly enough to be able to cope with large numbers of attributes.

To greatly reduce the tree size, we will take advantage of the observation that very many of the counts stored in the above tree are redundant. For each vary node, we will

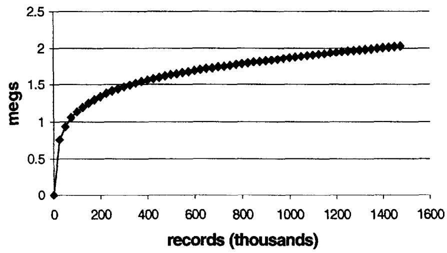


Figure 2: Memory usage in megabytes of ADtree for ASTRO database.

find the most common of the values of a_j (call it MCV) among records that match the node and we will store a NULL in place of the MCVth subtree. The remaining ($arity_j$-1) subtrees will be represented as before. An example for a simple dataset is given in Figure 1. Each "Vary a_j" node now stores which of its values is most common in a MCV field. (Moore and Lee 1997) describes the straightforward algorithm for building such an ADtree. On datasets with O(10^5) records and dozens of attributes, ADtrees typically consume 1-10 Megs of memory and require 1-10 seconds to build. The memory-cost of ADtrees increases sublinearly with the number of records. For example, Figure 2 shows memory use versus database size for the ASTRO database (described later).

Removing most-common-values has a dramatic effect on the amount of memory needed. For datasets with *M* binary attributes, the worst case number of counts that need to be stored drops from 3^M to 2^M, and the best case goes from 2^M to *M*. Furthermore, (Moore and Lee 1997) show that if the number of records is less than 2^M, or if there are correlations, or non-uniformities among the attributes then the number is provably much less than 2^M. As we shall see, this is borne out empirically.

Notice in Figure 1 that the MCV value is context dependent. Depending on constraints on parent nodes, a_2's MCV is sometimes 1 and sometimes 2. This context dependency can provide dramatic savings if (as is frequently the case) there are correlations among the attributes. This point is critical for reducing memory, and is the primary difference between the use of ADtree versus the use of Frequent Sets (Agrawal et al. 1996) for representing counts as suggested in (Mannila and Toivonen 1996).

ADtree-Assisted Counting

In (Moore & Lee 1997) an algorithm was presented and discussed that takes an ADtree and a set of attributes as input, and outputs a contingency table in constant time. Contingency tables are very closely related to probability

```
Preconditions:
Query_list ← list of attribute-value pairs sorted by attribute
Index ← 0
Current_ADnode ← root ADNODE of ADtree

AD_COUNT(ADnode, Query_list, index) {
If index equals the size of Query_list then
  Return ADnode's count

Varynode ← Vary node child of ADnode that corresponds to
                indexth attribute in Query_list
Next_ADnode ← ADNODE child of Varynode that corre-
                sponds to indexth value in Query_list

If Next_ADnode's count is 0 then
  Return 0

If Next_ADnode is a MCV then
  Count ← AD_COUNT(ADdnode, Query_list, index+1)
  For each s in siblings of Next_ADnode do
    Count ← Count - AD_COUNT(s,Query_list, index+1)
  Return Count

Return AD_COUNT(Next_ADnode, Query_list, index+1)
}
```

Figure 3: Pseudocode for AD_COUNT, an algorithm that returns the number of records matching a given list of literals.

tables in the Bayes net community or DataCubes (Harinarayan et al. 1996) in the database community.

Here we show how the ADtree can also be used to produce counts for specific queries in the form of a set of literals. For example, one can calculate the number of records having {a1=12, a4=0, a7=3, a8=22} directly from the ADtree. The algorithm in Figure 3 returns these types of counts.

Evaluation of a rule S1 ⇒ S2 only requires calculating *supp*(S1) and *supp*(S1 ∪ S2). A simple use of the ADtree is to return numbers of records matching simple queries which are conjunctions of literals, such as "in the ADULT1 dataset, how many records match {income=50K+, sex=male, education=HS}?" The answer can be returned by a simple examination of the tree, usually several orders of magnitude faster than going through the entire dataset. What that means in this particular application is that rules can be evaluated more quickly, and thus can be learned faster.

Since counting is so important to rule learning, we compare here the performance of ADtree counting against straightforward searching through the database. The comparison results are in Table 2. To generate the results, we generate many random queries, return counts for each, and time the counting process. Each randomly generated query consists of a random subset of attribute-value pairs

Dataset	Num Records	Num Attrib-utes	Tree Size (nodes)	Tree Size (MB)	Build Time (sec)
ADULT1	15060	15	58200	7.0	6
ADULT2	30162	15	94900	10.9	10
ADULT3	45222	15	162900	15.5	15
BIRTH	9672	97	87400	7.9	14
MUSHRM	8124	22	45218	6.7	8
CENSUS	142521	13	24007	1.5	17
ASTRO	1495877	7	22732	2.0	172

Table 1: Datasets used to produce experimental results. The size of the ADtrees from each dataset is included both in the number of nodes in the ADtree and in the amount of memory the tree used. The preprocessing time cost is given also. Descriptions of the datasets are in the appendix.

	Attributes in Query/Speedup Ratio				
Dataset	**2**	**4**	**6**	**10**	**20**
ADULT1	1019	208	76	25	-
ADULT2	1980	361	130	36	-
ADULT3	2782	508	166	46	-
BIRTH	1494	272	86	19	2.5
MUSHRM	881	319	179	82	28
CENSUS	10320	1139	261	61	-
ASTRO	20350	10506	6677	-	-

Table 2: Speedup ratio of average time spent counting a random query of a given size without ADtree vs. using ADtree.

taken from a randomly selected record in the database. Generating queries this way ensures that the count has at least one matching record. Why create the random queries as subsets of literals of existing records? Would it not have been simpler to generate entirely random queries? The reason is that completely random queries usually have a count of zero. The ADtree can discover this extremely quickly, giving it an even larger advantage over direct counting.

ADtree-Assisted CN2

We now look at how the fast counting method of the previous section can accelerate rule-finding algorithms. We look at CN2 (Clarke and Niblett 1989), an algorithm that finds rules involving arbitrary arity literals, as opposed to just positive binary literals. The rule-finding algorithm used here differs from CN2 in that only the most confident antecedents for S2 are sought instead of attempting to cover the entire dataset.

The learning algorithm is given S2 and begins search at S1 = {}. It then evaluates adding each possible literal one at a time, retaining the best k rules so produced. These best k rules are taken from each generation as starting points for the next generation. This process continues until the *minsupp* condition can no longer be satisfied or the length of the rule exceeds a preset maximum. Upon termination, the best rules ever encountered are returned. The search thus considers increasingly specific rules using a breadth-first beam search of beam size k.

Table 3 is a comparison of ADtree-assisted CN2 beam search rule learning and regular CN2 search. The parameters for the CN2 search were a beam size of 4 and a *minsupp* of 200. Rules learned were of the form S1⇒S2. The average time to learn a best rule for a randomly generated target literal, S2, was regarded as the "rule-learning time". The target literal S2 was restricted to being a single literal, where that literal's attribute was first selected randomly from all possible attributes for the dataset, then a value was randomly assigned from the set of values that the attribute could take on. Rule-learning times for normal CN2 and for ADtree-assisted CN2 were both recorded. Table 3 reports the ratio of these two averages for different rule size limits. Rule size is defined as the number of literals in S1 plus the number of literals in S2.

As can be seen, there is a general and large speedup achieved from using ADtree evaluation on these datasets. ADtrees provide a big win, provided the cost of building

Rule Size Limit	4			8			16		
	Regular Time (sec)	**ADtree Time (sec)**	**Speedup**	**Regular Time (sec)**	**ADtree Time (sec)**	**Speedup**	**Regular Time (sec)**	**ADtree Time (sec)**	**Speedup**
ADULT1	2.8	0.041	68.1	2.9	0.09	31.8	2.9	0.12	23.8
ADULT2	5.4	0.041	132.7	5.5	0.16	34.0	5.7	0.22	26.3
ADULT3	8.7	0.049	178.2	8.5	0.10	84.9	8.4	0.12	69.3
BIRTH	4.3	0.16	26.9	5.1	1.4	3.6	6.3	13.1	0.5
MUSHRM	1.8	0.039	46.6	1.8	0.064	28.5	1.9	0.094	19.9
CENSUS	16.6	0.058	286.8	15.8	0.22	71.1	16.1	0.26	61.3
ASTRO	198.6	0.034	5855.2	203.3	0.033	6191.5	-	-	-

Table 3: Comparison of CN2 with ADtree-assisted CN2 under different rule size limits.

the ADtree is not taken into account. If CN2 were to be run only once, the cost of building the datastructure would make the ADtree approach inferior. ADtrees become useful once again, though, if multiple rule learning tasks are performed with the same dataset, and thus the same ADtree. The cost can thus be amortized and can also allow interactive speeds once the initial cost is paid. Furthermore, without requiring rebuilding the tree, CN2 can be run on subsets of the database, defined by conjunctive restrictions on records (e.g., "find the rules predictive of high income restricted to people in the Northwest who rent their home") or on subsets of the attributes ("don't include any rules using age or sex") without rebuilding.

Our implementation of the unassisted counting version of CN2 exploited a major advantage of the original CN2 algorithm: the fact that, as candidate rules are made more specific, they are relevant to only a subset of the records relevant to their parent rule. Since it is the case that no more specific descendent of a rule can ever match records that the parent rule did not, a running list of relevant examples is kept for each rule. This list is pruned each time that the rule is made more specific, and drastically reduces the number of records that the algorithm needs to look at in order to evaluate a rule as it grows. Thus CN2 search speeds up dramatically near the end of the search. The ADtree, on the other hand, cannot use this information; it will always return a count for the entire dataset. Thus ADtree-assisted CN2 search slows down as larger rules are considered (see Table 3.)

However, ADtree's lack of reliance on general-to-specific search as an aid to rule evaluation allows for much greater flexibility in search. ADtree-assisted rule evaluation makes practical strategies such as exhaustive search, simulated annealing, and genetic algorithms for rule learning.

Discussion

The current implementation of ADtrees is restricted to only symbolic attributes. Furthermore, The current implementation assumes that the dataset can be stored in main memory. This is frequently not true. Work in progress (Davies & Moore 1998) introduces algorithms for building ADtrees from sequential passes through the data instead of by random access and does not require main memory data storage.

A disadvantage of ADtrees in rule learning is that they cannot be easily used to do "tiling" of datasets. Use of ADtree rule evaluation, one the other hand, does not restrict one to general-to-specific search. Moreover, the speed of ADtrees could make rule learning a process that takes place at interactive speeds. Relevant rules can be located quickly, perhaps enhanced by the operator's knowledge, then resubmitted to the program for further polishing of the hypothesis. The ADtree is a useful tool for creating fast and interactive rule learning programs.

Appendix

ADULT1: The small "Adult Income" dataset placed in the UCI repository by Ron Kohavi. Contains census data related to job, wealth, and nationality. Attribute arities range from 2 to 41. In the UCI repository this is called the Test Set. Rows with missing values were removed.

ADULT2: The same kinds of records as above but with different data. The Training Set.

ADULT3: ADULT1 and ADULT2 concatenated.

BIRTH: Records concerning a very wide number of readings and factors recorded at various stages during pregnancy. Most attributes are binary, and 70 of the attributes are very sparse, with over 95% of the values being FALSE.

MUSHRM: A database of wild mushroom attributes compiled from the Audubon Field Guide to Mushrooms by Jeff Schlimmer and taken from the UCI repository. Attribute arities range from 2 to 12.

CENSUS: A larger dataset than ADULT3, based on a different census. Also provided by Ron Kohavi. Arity ranges from 2 to 15.

ASTRO: Discretized features of 1.5 million sky objects detected in the Edinburgh-Durham Sky Survey.

References

Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., & Verkamo, A. I. 1996. Fast Discovery of Association Rules. In Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., & Uthurusamy, R. eds., *Advances in Knowledge Discovery and Data Mining*. AAAI Press.

Clark, P., & Niblett, R. 1989. The CN2 induction algorithm. *Machine Learning* 3:261-284.

Davies, S. and Moore, A. W.. 1998, Lazy and sequential ADtree construction. In preparation.

Harinarayan, V, Rajaraman, A. and Ullman, J. D., 1996, Implementing Data Cubes Efficiently. In Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems : (PODS 1996), Assn for Computing Machinery. Pages 205-216.

John, G. H., and Lent, B., 1997, SIPping from the data firehose. In Proceedings of the Third International Conference on Knowledge Discovery and Data Mining, AAAI Press, 1997

Mannila, H., and Toivonen, H., 1996, Multiple uses of frequent sets and condensed representations. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, edited by Simoudis, E., and Han, J., and Fayyad, U. AAAI Press.

Mitchell, T. 1997. *Machine Learning*. McGraw-Hill.

Moore, A.W., and Lee, M.S.,1997, Cached Sufficient Statistics for Efficient Machine Learning with Large Datasets. CMU Robotics Institute Tech Report TR CMU-RI-TR-97-27. *(Journal of Artificial Intelligence Research* 8. Forthcoming.)