

---

# Sequential Update of ADtrees

---

Josep Roura  
Andrew W. Moore

JROURE@CS.CMU.EDU  
AWM@CS.CMU.EDU

School of Computer Science, Carnegie Mellon University Pittsburgh, PA 15213

## Abstract

Increasingly, data-mining algorithms must deal with databases that continuously grow over time. These algorithms must avoid repeatedly scanning their databases. When database attributes are symbolic, ADtrees have already shown to be efficient structures to store sufficient statistics in main memory and to accelerate the mining process in batch environments. Here we present an efficient method to sequentially update ADtrees that is suitable for incremental environments.

## 1. Introduction

Modern data-mining problems often involve streams of data that grow continuously over time. Examples include customer click streams, telephone records, large sets of web pages, multimedia data, and sets of retail chain transactions. In these environments, sequential or incremental learning methods become particularly relevant, since they can revise existing models efficiently. It is widely accepted in the literature (Hulten & Domingos, 2002) that incremental algorithms should fulfill four constraints: (1) build a model using only one scan of data, (2) require small and constant time per record, (3) require a fixed amount of memory irrespective of the total number of records, and (4) obtain a model that is equivalent to the one that would be obtained with the corresponding batch algorithm. Note that the first three requirements indicate that an incremental algorithm should scale up well with data, while the last one requires that the scalability should not affect the quality of results.

Many data-mining algorithms deal with datasets with symbolic attributes. Such algorithms generally perform a great number of counting queries and so they spend many time doing direct counting from data. ADtrees (Moore & Lee, 1998) are sparse data structures able to answer any counting query efficiently with

respect to time. ADtrees have been seen to accelerate algorithms such as Bayes network learning (Teyssier & Koller, 2005; Goldenberg & Moore, 2004; Moore & Wong, 2003), association rules (Anderson & Moore, 1998) and feature selection (Moore & Lee, 1998).

In this paper we propose an incremental version of ADtrees so that learning algorithms use them in order to avoid multiple scan of data and increasing memory requirements.

## 2. Description of ADtrees

An ADtree is a data structure that efficiently stores the number of rows of a dataset that match any given query. Let us introduce some notation. We have a dataset with  $R$  rows and  $M$  attributes  $a_1, a_2, \dots, a_M$  and the  $i$ 'th attribute can take one of  $n_i$  different values. Usually  $n_i$  is called the arity of  $a_i$ . A query is a list of attribute-value pairs and an ADtree stores the number of rows that match each query. For example,  $(a_1 = 0, a_M = 1); (a_2 = 1); ()$  are three different queries where the attribute values that rows must match in order to be counted are specified and where the non-specified attributes mean a don't care value. In our example, all rows match the last query since it does not specify any restriction. The number of possible queries is  $\prod_{i=1}^M (n_i + 1)$  because each attribute can take the don't care value in addition to its own  $n_i$  values. This is far too large to fit in main memory a count for each query of a real-world database. We would like to retain the speed of precomputed counts without incurring an intractable memory demand.

### 2.1. ADtree Structure

First we describe an ADtree that explicitly stores a count for each possible query, and afterward, we will see how we can obtain a sparse structure. In an ADtree there are two types of nodes, ADnodes and vary nodes (Vary). The former are drawn as rectangles and the later as ovals in Figure 1. Each ADnode represents a query and stores the number of rows that match that query. The Vary  $a_i$  child of an ADnode is a specialization of its query for attribute  $a_i$ , and has one child for each of its  $n_i$  values.

---

Appearing in *Proceedings of the 23<sup>rd</sup> International Conference on Machine Learning*, Pittsburgh, PA, 2006. Copyright 2006 by the author(s)/owner(s).

The ADtree root is an ADnode that represents the empty query,  $()$ , and we say that it is in level 0. The ADnode root has one Vary child for each attribute. So an ADnode stores, in addition to the query count, a list of pointers to its children. Let  $a_i$  be a vary just under the root, then its  $j$ -th child is an ADnode that represents the query  $(a_i = j)$ , and we say that it is in level 1. An ADnode which is in level  $k$  represents a query with  $k$  attribute-value pairs, and each pair corresponds to the path chosen at each level from the root to the ADnode.

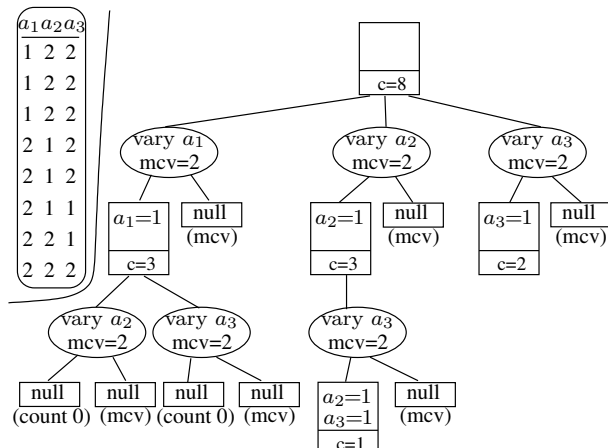


Figure 1. A sparse ADtree

Now we will introduce three modifications to ADtrees in order to very significantly reduce their size. First modification: we store a NULL instead of a node for any query that matches zero rows. All specializations of such query will also have a count of zero and will not appear anywhere in the tree. Second modification: for each Vary  $a_i$  in the tree we will find the most common of the values (call it MCV) of attribute  $a_i$  and store a NULL instead of the MCV-th sub-ADtree. The counts for the queries in these sub-trees can still be exactly calculated as we will see in the next section. See Figure 1 for an example of an sparse ADtree, with these two first modifications, built with the database also shown in the figure. Third modification: it is not worth building an ADtree for a dataset with few rows because the time we would save by asking the tree would not compensate the amount of memory being spent. For this reason we will not create a sub-ADtree for a node that matches less than  $R_{\min}$  rows. Instead that ADnode will keep a list of pointers to the rows matching it. The two major consequences of this modification is that some rows from the dataset will need to be maintained in main memory and the second is that the ADtree may require much less memory. See (Moore & Lee, 1998) for a discussion of the memory savings obtained with these modifications.

## 2.2. Computing Contingency Tables from Sparse ADtrees

Here we will explain how to compute contingency tables which in turn let us calculate those counts, associated with MCVs, not stored in sparse ADtrees. Later on, in this work, we will use a similar trick.

A contingency table  $ct(a_{i(1)}, \dots, a_{i(n)})$  is a table that has one entry for each possible instantiation of attributes  $a_{i(1)}, \dots, a_{i(n)}$  that contains the number of data rows that match that instantiation. Note that contingency table  $ct(a_{i(1)}, \dots, a_{i(n)})$  can be defined as the concatenation of the following conditional contingency tables  $ct(a_{i(2)}, \dots, a_{i(n)} | a_{i(1)} = 1)$ ,  $\dots$ ,  $ct(a_{i(2)}, \dots, a_{i(n)} | a_{i(1)} = n_i)$ . Note that we can recursively use this definition in each contingency table until we obtain tables with all attributes instantiated in the conditional part, for example  $ct(|a_{i(1)} = 1, \dots, a_{i(n)} = 1)$ . Such contingency table has one single entry for the count associated with the instantiation (or query). In other words, we can define a contingency table  $ct(a_{i(1)}, \dots, a_{i(n)})$  as a concatenation of the counts associated with each instantiation of attributes  $a_{i(1)}, \dots, a_{i(n)}$ .

---

### Algorithm 1 Contingency table from sparse ADtree

---

```

MakeContab(attList = { $a_{i(1)} \dots a_{i(n)}$ }, adn)
  if attList == {} then
    Return adn.count
  else
    vary = the Vary  $a_{i(1)}$  subnode of adn
    for ( $k = 1; k \leq adn.nvals; k++$ ) do
      if  $k \neq vary.MCV$  then
         $adn_k = vary.adn_{i(1)}$ 
         $CT_k = MakeContab(\{a_{i(1)} \dots a_{i(n)}\}, adn_k)$ 
      end if
    end for
     $CT_{vary.MCV} =$  as explained above
    Return the concatenation of  $CT_1, \dots, CT_{adn.nvals}$ 
  end if
    
```

---

Algorithm 1 builds the contingency table for a list of attributes. It follows the recursion described above and uses an ADtree in order to obtain the counts associated with queries. The algorithm omits the calculation of the contingency tables for the MCVs,  $CT_{vary.MCV}$ . In order to do so, we can take advantage of the following property,  $ct(a_{i(1)}, \dots, a_{i(n)} | Query) = \sum_{k=1}^{n_{i(1)}} ct(a_{i(2)}, \dots, a_{i(n)} | a_{i(1)}, Query)$ . The value  $ct(a_{i(2)}, \dots, a_{i(n)} | Query)$  can be computed with our algorithm by calling  $MakeContab(\{a_{i(2)}, \dots, a_{i(n)}\}, adn)$  and so the missing conditional contingency table in the algorithm can be computed by the following row-wise subtraction:  $CT_{vary.MCV} = MakeContab(\{a_{i(2)}, \dots, a_{i(n)}\}, adn) - \sum_{k \neq vary.MCV} CT_k$ . Refer to (Moore & Lee, 1998) for a discussion of the time complexity of Algorithm 1.

### 2.3. ADtree’s Memory and Time Complexity

A non-sparse ADtree will have  $\prod_{i=1}^M (n_i + 1)$  ADnodes, that is, one for each possible query. In contrast, the sparse ADtree stores an ADnode for each query that does not contain any MCV in the instantiation of its attributes, and so one symbol is not used. Thus the memory cost is  $\prod_{i=1}^M (n_i)$ . Note that this holds despite the fact that the MCV of a given attribute may be different in different parts of the ADtree.

Now, without loss of generality, we assume that all attributes are binary. In this case, the memory cost for a non-sparse ADtree is  $3^M$  while the cost for a sparse one is  $2^M$ . Note that this worst case will happen only if all possible records exist in the database, otherwise the ADtree would be smaller since it does not store counts of zero. Usually the number of records is  $R \ll 2^M$  and since at each level the records are split in two subsets (one for each attribute-value) then, at most, the total number of levels an ADtree can have is  $\lceil \log_2 R \rceil$ . Noting that the number of nodes at level  $k$  is  $\binom{M}{k}$ , we obtain that the number of ADnodes in a sparse ADtree is  $\sum_{k=0}^{\lceil \log_2 R \rceil} \binom{M}{k}$ . From this we can see that the time complexity of building a sparse ADtree is  $\sum_{k=0}^{\lceil \log_2 R \rceil} R 2^{-k} \binom{M}{k}$  since at each level we have to go through  $R 2^{-k}$  records for each node in level  $k$ . Refer to (Moore & Lee, 1998) for an extended discussion of the memory and time complexity of ADtrees.

## 3. Sequential ADtrees

In this section, we want to transform an ADtree that is built all at once from a given dataset to one that is able to update its counts as new rows of data are incorporated into the database. Strictly speaking, an incremental ADtree should be updated each time a single data row is acquired, however in many incremental environments data arrive in small batches of instances. In addition, it is usually best to wait until there is some small number of new instances since a single one would not provide enough new information to introduce a significant modification to the ultimate data model being learned. So, we will assume that data is acquired in small batches.

### 3.1. First Sequential Approach

The most straightforward way to obtain an incremental ADtree is to adapt the procedures used to build an ADtree, as in Appendix B in the original paper (Moore & Lee, 1998). The initial ADtree is built from scratch with the first batch of data using the above-mentioned algorithms. When new data arrive, the existing ADtree is updated by incrementing the counts of existing ADnodes according to the number

of rows that match their query, and by creating new sub-ADtrees for those queries that used to be zero. We want to stress that, in this approach, once MCVs are set, they will never be reconsidered and thus it may happen that they stop corresponding to the actual MCVs when counts are updated.

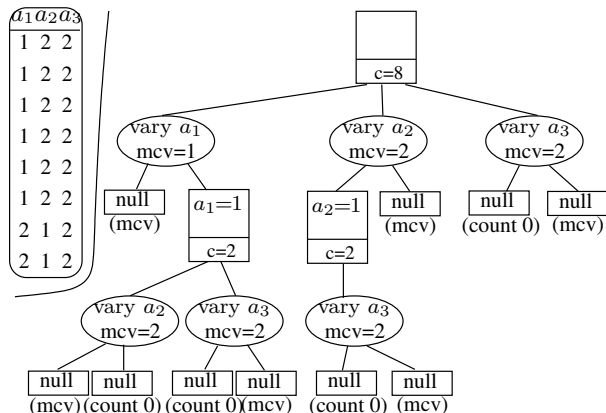


Figure 2. ADtree build with a first batch of data

In order to better illustrate the updating process let us follow the *path* of data rows. First, an ADtree is built with the first batch of data, see Figure 2. Once this first version of the ADtree is built, when a new batch of data arrives, the tree is updated. Figure 3 shows a new batch of data, the updated ADtree, and the *path* of data rows down the tree. The *path* begins at the top of the ADtree going down the levels. When, in a given level, rows match a query that does not correspond to a MCV the count is updated and the rows continue their path down the tree to the following level. Next, however, if the rows match a query corresponding to the MCV (there is not an ADnode for this query) then nothing has to be updated and the path ends. A third possibility is that the rows match a query that used to be zero, in this case, a new ADnode is created for this query and for the ones below. Recall that the new data might change the MCVs of vary nodes but this version does not update them. Figure 3 shows an ADtree where the MCV of *vary a1* is incorrect ( $mcv=1$ ). We want to remark that this procedure is the same as described in the original paper with the only difference that now ADnodes may already exist for matched queries and thus they must be updated rather than created.

The main advantage of this approach is that *ideally* its memory and time complexity is the same as the original algorithm. But, in fact, this is only true when, during the incremental process, sub-ADtrees are built with the actual MCV values of the whole dataset. For example, take the ADtree in Figure 1 and update it with the same database shown in the figure. The resulting ADtree will have exactly the same structure

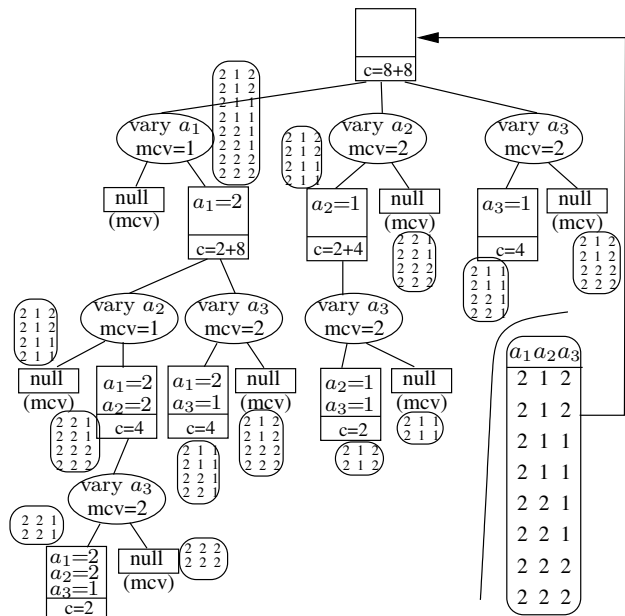


Figure 3. Updated ADtree

and all its counts will be doubled. In this case, the time spent building the final tree is the same as that spent if we had built the tree directly from scratch using a single batch with twice the database shown in the figure. In general, if we construct a database with a set of  $R/n$  rows and copy it  $n$  times to obtain a dataset of size  $R$ , then both the batch and incremental approaches would spend in time  $\sum_{k=0}^{\lceil \log_2(R/n) \rceil} R/2^k \binom{M}{k}$ . Note that the number of levels depends on the number of possibly different rows  $R/n$ .

A different case would be to build a dataset with  $n$  batches of the same size  $R/n$  ordered in a way where the rows of each batch matched the least common values of the vary nodes build with the preceding batches. So, the MCVs of one batch are not representative of the whole dataset. See, for example, Figure 3 where all rows from the new batch travel down the tree though the least common value of *vary a1*. In the worst case, all the rows from one batch would travel through the least common values to each of the existing leaves and build new sub-ADtrees from there. In such a case the complexity in time would be,

$$\sum_{i=0}^{R/n} \sum_{k=0}^{i \lceil \log_2 R/n \rceil} R/n \binom{M}{k} + \binom{M}{i \lceil \log_2 R/n \rceil} \sum_{k=0}^{\lceil \log_2 R/n \rceil} R/(2^k) \binom{M}{k}$$

Where the first term of the sum corresponds to the data traveling down the existing ADtree, and the second to build new sub-ADtrees from each of the leaves. The time spent for this dataset is much bigger than it would be if we knew the actual MCVs. Accordingly the size of the ADtree is proportional to the equation showed above.

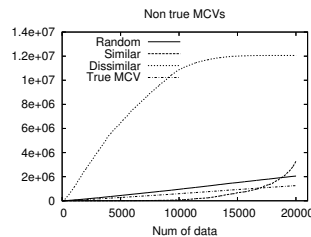


Figure 4. ADtree size. True and non true MCVs

From this second example we have learned that ADtrees updated in this fashion can rapidly degenerate, which is the main drawback of this simple approach. This is further illustrated in Figure 4 where it is shown the number of ADnodes (y-axis) when ADtrees are grown with a certain amount of data (x-axis). We used the Alarm (Cooper & Herskovits, 1992) dataset presented in three different orders of data rows. In the first order, similar rows are presented together. In the second, dissimilar rows are presented together. In the third, rows are randomly sampled. The graph shows the growing behavior using this first simple incremental approach where ADtrees may not have the actual MCVs. There are three curves that correspond to the three orders and a fourth one, drawn for comparison purposes, that corresponds to the random order when the ADtree has the actual MCV. We can see that different orders yield different sizes and also that they can be much bigger than the ADtree with the actual MCV. For example the ADtree grown using the dissimilar order is an order of magnitude bigger than the one with the actual MCVs. Note that this case is close to the second example above where most of the rows travel to each leaf. The smallest ADtree is obtained with the random order because the sampling of the dataset is less skewed and therefore the guess of the MCVs is more accurate. Note that the ADtree of Figure 3 is grown with the same database as the ADtree of Figure 1 (with twice the rows shown) and that the former is bigger than the latter.

**Algorithm 2** Obtains an ADtree with correct MCV

---

```

ADtreeCorrectMCV(adn, bn)
  for (i = bn; i ≤ M; i++) do
    VaryCorrectMCV(adn, adn.varyi, i)
  end for
VaryCorrectMCV(adn, vary, bn)
  oldMCV = adn.count -  $\sum_{i \neq \text{vary.MCV}} \sum_{i \in [1, \text{vary.nvals}]}$  vary.adni.count
  maxIdx = argmaxi(vary.adni.count)
  if oldMCV < vary.adnmaxIdx.count then
    vary.adnmaxIdx = ADtreeBuildMCV(adn, vary, bn + 1)
    vary.adnvary.MCV = NULL
    vary.MCV = maxIdx
  end if
  for (i = 1; i ≤ vary.nvals; i++) do
    if vary.adni != NULL then
      ADtreeCorrectMCV(vary.adni, bn + 1)
    end if
  end for
    
```

---

Such an algorithm needs to go through all the *vary* nodes of the ADtree and check whether its current MCV is the actual one. If any are incorrect the sub-ADtree that corresponds to the actual MCV has to be build and the sub-ADtree of the incorrect MCV dropped (see Algorithm 2). This algorithm will first be called with the root of the ADtree being processed (*adn*) and 1 for the base index (*bn*). Note that, as we will see below, **ADtreeBuildMCV** (Algorithm 3) is not guaranteed to return a sub-ADtree with the actual MCVs, so it also needs to be checked.

**Algorithm 3** **ADtreeBuildMCV**(*adn*, *vary*, *bn*)

---

```

tmpCount = adn.count
  for (i = 1; i < vary.nvals; i++) do
    if vary.adni != NULL then
      tmpCount = tmpCount - vary.adni.count
    end if
  end for
  if tmpCount != 0 then
    if bn == M then
      adnNew = new(adn) {Base case}
      adnNew.count = tmpCount
    else
      adnNew = CopyTree(adn, bn + 1)
      for (i = 1; i ≤ vary.nvals; i++) do
        if vary.adni != NULL then
          ADtreeSubs(adnNew, vary.adni, bn)
        end if
      end for
    end if
  else
    adnNew = NULL
  end if
  return adnNew
    
```

---

In order to build the sub-ADtree, see Algorithm 3, that corresponds to a MCV we will use the same trick previously used to calculate contingency tables but now subtracting ADtrees instead of tables. That is, to build the ADtree that corresponds to the MCV of Vary  $a_i$  in a certain level, we first need to copy the ADnode parent of Vary  $a_i$  and all its subtrees that correspond to a Vary  $a_j$  where  $i < j \leq M$ , (see Figure 5). Second, we need to subtract from the copied ADtree all the sub-ADtrees of Vary  $a_i$ . The base case occurs when  $i = M$ , that is, the sub-ADtree has one

level and the MCV can be calculated subtracting from the count in the ADnode parent of Vary  $a_i$ , the counts in its children ADnodes.

**Algorithm 4** Subtracts two ADtrees

---

```

ADtreeSub(Aadn, Badn, bn)
  Aadn.count = Aadn.count - Badn.count
  for (i = bn; i ≤ M; i++) do
    VarySub(Aadn.varyi, Badn.varyi, Badn, i)
  end for
VarySub(Avary, Bvary, Badn, bn)
  for (i = bn; i < Avary.nvals; i++) do
    if Avary.adni != NULL && i == Bvary.MCV then
      Bvary.adni = ADtreeBuildMCV(Badn, Bvary, bn + 1)
    end if
    if Avary.adni != NULL && Bvary.adni != NULL then
      if Avary.adni.count == Bvary.adni.count then
        Avary.adni = NULL
      else
        ADtreeSub(Avary.adni, Bvary.adni, bn + 1)
      end if
    end if
  end for
    
```

---

In order to subtract one ADtree from another  $ADtree_1 - ADtree_2$ , we need to go through all the ADnodes and subtract their counts, see Algorithm 4. Note that the result should never be negative and thus  $ADtree_1 \leq ADtree_2$  in terms of counts. Here we will notate  $vary_i^1$  as a vary node of attribute  $a_i$  in  $ADtree_1$ . When we subtract the sub-ADtrees below two vary nodes,  $vary_i^1$  and  $vary_i^2$  we need to subtract all their sub-ADtrees,  $\forall j \in [1, vary_i^1.numVals]$   $vary_i^1.adn_j - vary_i^2.adn_j$  and different cases of the recursion must be considered. Firstly, we enumerate the base cases:

- $vary_i^1.MCV = j$  (then the sub-ADtree is redundant and does not need to be calculated)
- $vary_i^1.adn_j.count = vary_i^2.adn_j.count = 0$  ( $vary_i^1.adn_j$  and  $vary_i^2.adn_j$  in fact are null)
- $vary_i^1.adn_j.count = vary_i^2.adn_j.count \neq 0$  (the resulting sub-ADtree is null)
- $vary_i^1.adn_j \neq null$  and  $vary_i^2.adn_j.count = 0$  ( $vary_i^2.adn_j$  in fact is null)

Secondly, we enumerate the cases where additional recursion is needed:

- $vary_i^1.adn_j.count > vary_i^2.adn_j.count \neq 0$ , then the subtraction of counts is performed and their sub-ADtrees also need to be subtracted
- $vary_i^1.adn_j.count \neq 0 \wedge vary_i^2.MCV = j$ , then the sub-ADtree  $vary_i^2.adn_j$  must be calculated and the subtraction performed

Thirdly, there are two cases that would result in negative counts:

- $vary_i^1.adn_j.count < vary_i^2.adn_j.count \neq 0$
- $vary_i^1.adn_j.count = 0 \wedge vary_i^2.MCV = j$  ( $vary_i^1.adn_j$  in fact is null)

See that the resulting ADtree of the subtraction is not guaranteed to have the correct MCVs and thus neither does **ADtreeBuildMCV** (Algorithm 3).

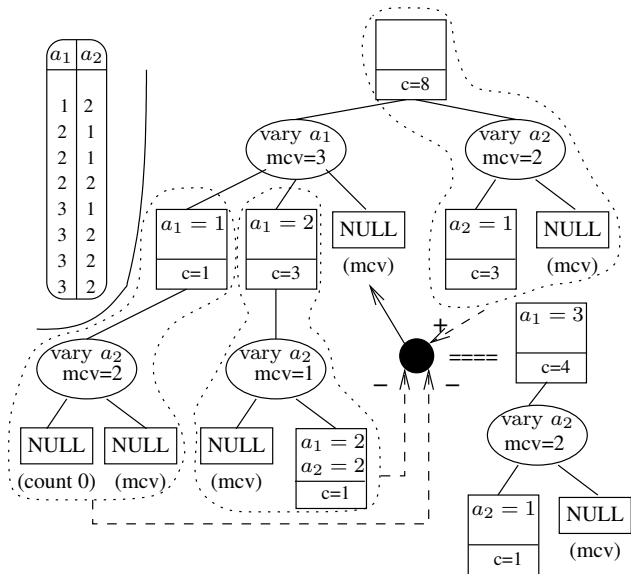


Figure 5. Calculate MCV sub-ADtree

The complexity of calculating the ADtree with the actual MCVs is proportional to the number of nodes of a non-sparse ADtree,  $\prod_{i=1}^M (n_i)$ . The worst case happens when all MCVs are incorrect and their sub-ADtrees have to be calculated. Since the algorithm that calculates those sub-ADtrees does not guarantee to return ADtrees with correct MCVs, again in the worst case all MCVs are incorrect. Following this recursion, at the end, counts for all the possible queries must be calculated. In a *normal* case an ADtree will not have all its MCV wrong and will have some zero counts but it is still computationally expensive as we will see in the experiments in Section 4.

In addition to the time complexity, another drawback for this approach is that when ADtrees use the leaf lists we also need to keep the lists of the rows that match each MCV. In this manner an ADtree will always grow proportionally to the length of a data-stream, violating the third requirement for incremental algorithms.

### 3.3. A Buffering Strategy

In the previous section we saw that to calculate an ADtree with correct MCVs is very expensive. In this section we re-examine the idea of Section 3.1 and further develop it. The main problem of the first simple approach was that a decision of what is the MCV with respect to the whole dataset was usually taken with very few rows and thereof the probability of making a wrong choice was very high. In order to alleviate this problem, we will wait to make a decision until the probability of being wrong is low.

See that a decision needs to be taken at every Vary of the ADtree and that it has to be done with a sample of the dataset. In order to have a low probability of

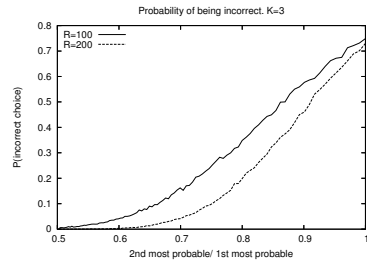


Figure 6. Probability of being incorrect

being wrong, we will postpone the decision until we have enough rows. That is, we will have a buffer at each Vary in order to store the rows *traveling* down the ADtree and the sub-ADtree will not be build until there are sufficient rows. A decision will be made only when the size of the buffer is larger than  $n$  or larger than  $m < n$  and the ratio between the MCV and the second MCV is larger than  $\alpha$ .

In order to theoretically justify that this approach will make the correct choice with high probability we use the following result (Bechhofer et al., 1959):

**Theorem 1 (Probability of a correct selection)**  
 Given a multinomial random variable with  $k$  values and  $R$  and any probability vector  $\mathbf{p}$  with  $p_{k-1} < p_k$ , the probability of a correct selection is given by  $\theta_k = \theta(p_1, \dots, p_k) = \sum \frac{1}{s} \cdot \frac{R!}{y_{(1)R}! \dots y_{(k)R}!} p_1^{y_{(1)R}} \dots p_k^{y_{(k)R}}$  where the summation is over all vectors  $\mathbf{Y}_R = (y_{(1)R} \dots y_{(k)R})$  such that  $\sum_{i=1}^k y_{(i)R} = R$  and  $y_{(k)R} \geq y_{(i)R} \forall i = (1, \dots, k-1)$ , and  $s$ , is the number of  $y_{(i)R}$ 's tied for largest.

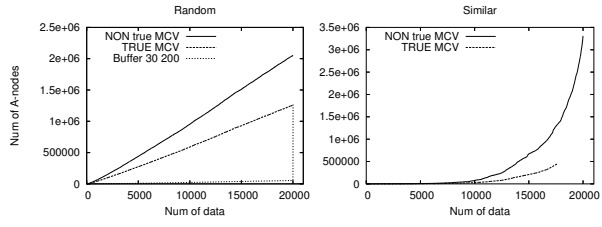
From this theorem we can see that the more data we have the higher is the probability of being correct, and that the flatter the probability distribution of the variable the more chances of being incorrect. To illustrate this fact we performed a simulation with a multinomial random variable where  $k = 3$  and  $\mathbf{p} = (\delta / (\mathbf{k} - 1 + \delta), 1 / (\mathbf{k} - 1 + \delta), 1 / (\mathbf{k} - 1 + \delta))$  and where  $\delta \in [1, 2]$ . This variable was sampled 1000 times with  $R = 100$  and  $R = 200$  data and the probability of being wrong was estimated. In Figure 6 we can see that the probability of being wrong is very low (almost zero) when the probability of MCV is twice the probability of the second MCV. We want to stress that to make a wrong choice when the distribution is very flat will not result in a much bigger ADtree since the number of rows that match the queries for the different values will be similar. Note also that the time cost of this buffering strategy is the same than the one discussed in Section 3.1 but now the worst case is much more unlikely and thus closer to the batch approach.

## 4. Experiments

We performed several experiments to see how sensitive our buffering strategy is to ordering effects. We

# Sequential Update of ADtrees

---



more than that spent calculating the ingrown ones.

## 5. Conclusions

In this work, we have proposed a sequential ADtree suitable for incremental environments where new data are available over time. We saw that having an ADtree with all the MCVs correct would be optimal in memory and query time but also that would be very expensive in computing time. Then, we proposed a buffering strategy that basically defers decisions until some amount of data rows are available and the probability of being wrong is low. We have experimentally seen that ordering effects are reduced and that the ADtrees built with this technique are close to those built with all the database at once. Buffering strategies have proven to work well in other fields such as incremental clustering (Talavera & Roure, 1998).

A drawback for this buffering strategy could be that if many data are buffered and many sub-ADtrees are not grown then queries would require more time. This could be solved by growing those sub-ADtrees being queried (like in dynamic ADtrees (Komarek & Moore, 2000)) even if the probability of being wrong is high. We could still keep the buffers, so that, when enough records were available, we could drop the ill-grown sub-ADtrees and build new ones. However, we saw in our experiments that with small buffers (size 200) this strategy works well. In cases with very strong memory restrictions, we could combine the buffering strategy with running the process to calculate the actual MCVs only when computing time is available.

We strongly believe that sequential ADtrees are a valuable tool for data-mining algorithms in incremental environments. Incremental algorithms, in addition to the gain they obtain from querying cached sufficient statistics, can use sequential ADtrees to avoid multiple scans of data and to bound the memory requirements. The simplest version of sequential ADtrees have already been used in incremental learning of Bayesian Network structures (Roure, 2004b; Roure, 2002a), TAN (Roure, 2002b) and BAN (Roure, 2004a). In order to reduce the amount of memory used by ADtrees, these methods use the fact that the number of parents of variables is usually limited to some number  $n$  that is much smaller than the total number of variables. The queries required to learn such structures have a maximum of  $n + 1$  variables, that can be answered with only the first  $n + 1$  levels of ADtrees.

## Acknowledgments

The first author was supported by the Fulbright program and the Generalitat de Catalunya

## References

- Anderson, B., & Moore, A. (1998). Ad-trees for fast counting and for fast learning of association rules. *Knowledge Discovery from Databases Conference*.
- Bechhofer, R. E., Elmaghraby, S., & Morse, N. (1959). A single-sample multiple-decision procedure for selecting the multinomial event which has the highest probability. *The Annals of Mathematical Statistics*, 30, 102–119.
- Blake, C., & Mertz, C. (1998). UCI repository of Machine Learning databases.
- Cooper, G., & Herskovits, E. (1992). A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9, 309–347.
- Goldenberg, A., & Moore, A. (2004). Tractable learning of large bayes net structures from sparse data. *International Conference on Machine Learning (ICML)*.
- Hulthen, G., & Domingos, P. (2002). Mining complex models from arbitrarily large databases in constant time. *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.
- Komarek, P., & Moore, A. (2000). A dynamic adaptation of ad-trees for efficient machine learning on large data sets. *International Conference on Machine Learning (ICML)* (pp. 495–502).
- Langley, P. (1995). Order effects in incremental learning. In P. Reimann and H. Spada (Eds.), *Learning in humans and machines: Towards an interdisciplinary learning science*. Pergamon.
- Moore, A., & Lee, M. S. (1998). Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8, 67–91.
- Moore, A., & Wong, W.-K. (2003). Optimal reinsertion: A new search operator for accelerated and more accurate bayesian network structure learning. *International Conference on Machine Learning (ICML)*.
- Roure, J. (2002a). An incremental algorithm for tree-shaped Bayesian network learning. *Proceedings of the European Conference of Artificial Intelligence (ECAI)*.
- Roure, J. (2002b). Incremental learning of tree augmented naive bayes classifiers. *Ibero-American Conference on Artificial Intelligence (IBERAMIA)*.
- Roure, J. (2004a). Incremental augmented naive bayes classifiers. *Proceedings of the European Conference of Artificial Intelligence (ECAI)*.
- Roure, J. (2004b). Incremental hill-climbing search applied to bayesian network structure learning. *Workshop of Knowledge Discovery on Data Streams, (KDD-ECML)*.
- Talavera, L., & Roure, J. (1998). A buffering strategy to avoid ordering effects in clustering. *Proceedings of the European Conference on Machine Learning, (ECML)*.
- Teyssier, M., & Koller, D. (2005). Ordering-based search: A simple and effective algorithm for learning bayesian networks. *Proceedings of the Twenty-first Conference on Uncertainty in AI (UAI)*.