Methods to Reduce I/O for Decision Tree Classifiers

Vineet Singh and Anurag Srivastava
Hitachi America, Limited
Research and Development Division
3101 Tasman Drive, M.S. 120
Santa Clara, CA 95054, USA
(vsingh,anurags)@hitachi.com
(This work was done by the authors at IBM T.J. Watson Research Center.)

Abstract

Classification is an important data mining problem. Although datasets can be quite large in data mining applications, it can be advantageous to use the entire training dataset as opposed to sampling since that can increase accuracy. I/O is a significant component of overall execution time in many decision tree classifiers. We present some new optimizations that work with many of these classifiers on both sequential and parallel processors. For ease of explanation, we describe these optimizations mostly in the context of SPRINT, a classifier developed recently for large problems where the training datasets may be disk resident.

1. Introduction

Classification is an important data mining problem. Recently, there has been significant interest in classification using training datasets that are large enough that they do not fit in main memory and need to be disk-resident. Although training data can be reduced by sampling, it has been shown that it can be advantageous to use the entire training dataset since that can increase accuracy [Cat91, CS93a, CS93b].

A classification problem has an input dataset called the training set which consists of a number of examples each having a number of attributes. The attributes are either *continuous*, when the attribute values are ordered, or *categorical*, when the attribute values are unordered. One of the categorical attributes is called the *class label* or the *classifying attribute*. The objective is to use the training dataset to build a model of the class label based on the other attributes such that the model can be used to classify new data not from the training dataset. Application domains include retail target marketing, fraud detection, and design of telecommunication service plans.

Several classification models like neural networks [Lip87], genetic algorithms [Gol89], and decision trees [Qui93] have been proposed. Decision trees are probably the most popular since they obtain reasonable accuracy [DMT94] and they are relatively inexpensive to compute.

In the data mining domain, the data to be processed tends to be very large. Hence, it is highly desirable to design computationally efficient algorithms. One way to reduce the computational complexity of building a decision tree classifier using large training datasets is to use only a small sample of the training data. Such methods do not yield the same classification accuracy as a decision tree classifier that uses the entire data set [WC88]. In order to get reasonable accuracy in a reasonable amount of time, one option is to design optimizations to reduce I/O time, a major component of the overall execution time.

Most current classification algorithms including the popular C4.5 [Qui93] are based on the ID3 algorithm [Qui93]. However, most of these algorithms are not suitable for large disk-resident datasets because they require that all the training data be memory resident and they require multiple sorting passes over the data and its subsets.

The SPRINT algorithm [SAM96] is designed to be more suitable for large datasets as well as parallel processing. Data can be arbitrarily large and may be disk-resident. We describe some new I/O optimizations that work with SPRINT as well as other decision-tree classification algorithms such as SPEC [SSHK97], SCALPARC [JKK98], and CLOUDS [ARS98].

This paper is organized as follows. Section 2 describes the basic approach of decision-tree classification. Section 3 describes the SPRINT algorithm, which was designed specifically for large disk-resident datasets. Sections 4, 5, 6 describe our three new I/O optimizations. Section 7 presents the conclusions.

2. Decision-Tree Classification

In this paper, we restrict our attention to decision tree classification algorithms based on ID3. For more details, the reader is referred to [Qui93]. The input is a training dataset such as the one shown on the left in Figure 1. **tid** is simply a unique number for each example. In this case, the classifying attribute is **Risk**, which can be either **low** or **high**, abbreviated L or H respectively. The other attributes are (1) **Car**, which is a categorical attribute and can take the values **family**, **sports**, or **truck**, (2) **Age**, which is a continuous attribute, and (3) Salary, which is also a continuous attribute. The output is a decision tree such as the one shown on the right in Figure 1. Each node is either (1) a leaf node with a class label or (2) an interior node with an associated test on a single attribute value. For continuous attributes, a test is of the form $v \leq x$, where v is the attribute value and x is some constant value. For categorical attributes, the test is a subset condition such as $v \in S$ where S is a subset of all values for v. A new example is classified by applying the tests starting from the root node and following the resulting path to a leaf node. For illustrative purposes only, we provide an additional labeling of each node for the decision tree shown on the right in Figure 1. The additional label given directly below the test or class label is the subset of values from the training dataset that would be classified to that node.

tid	*Car*	*Age*	*Salary*	*Risk*
0	family	30	65	L
1	sports	23	15	H
2	sports	40	75	L
3	family	55	40	H
4	truck	55	100	L
5	family	45	60	L

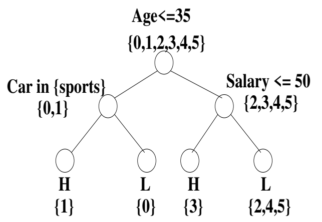


Figure 1. Decision tree classification

The two major phases of the classification process are initial tree building and pruning. Initial tree building is done by recursively partitioning the input training dataset such that the leaves contain examples from a single class (i.e., the classifying attribute values are identical). The splitting test chosen at an interior node is the one that minimizes the overall entropy [BFOS84] of the children nodes. Almost all of the decision tree algorithms use sorting of continuous attributes to consider each possible split point of data.

The second phase called the pruning phase prunes the tree to avoid the tree from over-fitting the data in order to remove statistical variations and noise. Typically, this phase takes less than 1% of the time for the whole algorithm [SAM96]. We will not discuss this further in this paper because we use the same pruning method as SPRINT.

3. SPRINT overview

3.1. Data Structures

SPRINT has two major data structures: attribute lists and histograms. There is an attribute list for each attribute for each node. Each attribute list has associated histograms. Examples of these data structures for the **Risk** data of Figure 1 are given in Figure 2. The **Age** and **Car** attribute lists are given in Figure 2 for the root node. There is a row for each **tid**. Each row in an attribute list contains the **tid**, attribute value, and class label of one example. For continuous attributes, the attribute lists are kept sorted by attribute value as shown for the **Age** attribute list. The initial lists created from the training set are associated with the root node. During tree building when nodes are split to create children, the attribute lists are partitioned. During partitioning, the ordering of the examples is preserved. Therefore, if the the list for the root is sorted, lists for other nodes remain sorted.

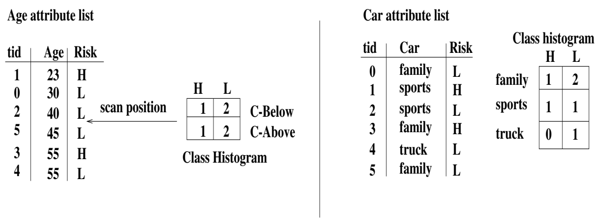


Figure 2. SPRINT data structures

Histograms are used to show the class distributions of the records at a particular node of the decision tree. Two histograms are maintained for each continuous attribute called **C-below** and **C-above**. They are associated with a particular scan position in the attribute list. In the **Age** attribute list in Figure 2, the **C-below** histogram shows the class distribution for all examples which have **Age** $\leq$ **40**. The **C-above** histogram shows the class distribution for all examples which have **Age** $>$ **40**. For categorical attributes, only one histogram is required and it contains the class distribution for each value of the attribute.

3.2. Finding split points

To find the best split point for a node, each attribute list for that node is scanned once. For a continuous attribute, the candidate split points (and associated scan positions) are mid-points between every two consecutive (distinct) values in the training data. Before the scan of an attribute list, the **C-below** histogram is initialized to all zeros, while the **C-above** histogram is initialized to the class distribution for all the examples for that node. This can be done when the node is first created from the parent node. For the root, it can be done when the attribute lists are first sorted. During the scan, the histograms are updated for each scan position. The histograms have all the information required to compute the *gini index* [BFOS84], an entropy-like metric that is used by SPRINT and other algorithms. After the scan, we have the best cut for that attribute.

For categorical attributes, the histogram is used only after the scan of the entire attribute list is done. At that point, the gini index can be computed for all subsets of attribute values (or some approximation if the number of subsets is too large). The subset test that gives the best gini index is chosen.

After scanning all the attributes lists, we can determine the best split overall based on the best splits found for each attribute.

SPRINT generates the tree in a breadth-first manner. Attribute lists for a particular attribute for all the nodes at a particular level are kept in one sequential file.[1] One scan through the attribute file for an attribute is sufficient to determine the best split for that attribute for each node at that level of the tree.

3.3. Performing the split

For a particular node, the attribute list for the attribute that gives the best split can be partitioned directly based on the split test. The other attribute lists cannot be partitioned directly since the tid order of examples in different attribute lists will be different (since they are sorted separately at the beginning). Therefore, a hash table needs to be created first from the split of the splitting attribute. The hash table indicates the splitting status for each tid – either left or right child. For splitting the other attribute lists, the tid can be used to probe the hash table and determine the splitting status for each example (or row). Again, for a particular attribute, the child partitions of all nodes at a particular level are written to a single sequential file. Note that each attribute list is read once and written once for splitting each level of the decision tree.

4. Reduction in reads from disk

To generate the children of a decision tree node, SPRINT requires one read of each attribute list to determine the best split point and another read/write to split the node. However, one of these reads can be eliminated. Essentially, some pre-computation can be done while a node is being split into its children. During the write to create the children, the best split can be computed for each of them. Therefore, there is no need to have the usual read just to compute the best split for the children. With this optimization, it is still necessary to have two reads to split the root node of the decision tree. After that, each node requires a single read and a single write of each attribute list.[2]

[1] It is actually two sequential files for an additional optimization but the rest of the SPRINT description is orthogonal to this optimization.

5. Reduction in writes to the disk

It is possible to avoid writes at every alternate level of the decision tree by using the following method. We generalize the method to avoid writes for n levels afterwards.

Figure 3 shows the write optimization, which skips writes at every alternate level. Assume that level $i-1$ of the tree is being split, leading to the generation of level i of the decision tree. During this process, first, the attribute list for the splitting attribute is read at level $i-1$ and a hash table is created. This hash table is used to write attribute values of non-splitting attributes to the left or right node at level i and then deleted. Next, when splitting the nodes at level i, attribute values are not written for the next level $i+1$. Also, while reading these values at level i, histograms for the nodes at level $i+1$ are calculated to avoid a redundant read phase at level $i+1$, as explained in the read optimization. All other processing done previously, except actual write of attribute values, is done while reading the attribute values for level i. Now for splitting level $i+1$ and generating the level $i+2$ of the tree, the attribute list is read from the corresponding attribute list of the ancestor node at level i. The hash table for level $i+1$, used to generate nodes at level $i+2$, is computed by extending the hash table for the parent node at level i by the splitting attribute at level $i+1$. For a binary tree, this requires an extra bit per hash table entry. All splits of other attribute lists (other than the splitting attribute list) at level $i+1$ into nodes at level $i+2$ can now use the extended hash table of the parent node at level i.

In summary, if write is skipped during the split of nodes at a particular level i, hash table created at level i augmented with the splitting attribute for level $i+1$ node is used to perform a four way split for separating and writing attribute lists for nodes at level $i+2$. This skip of write phase can be repeated again while splitting level $i+2$, or generating level $i+3$.

A disadvantage of this technique can be illustrated for a binary tree. Without the write reducing technique, the writes have to be in two locations for each of the two children. Now, the writes have to be in four locations for the grand children, which are generated simulataneously.

This write technique can be further generalized to skip writes at n contiguous levels. This can be done by augmenting the hash table with n bits.

[2] The extra read for the root node can be eliminated by combining the read for determining the split into the pre-processing required to generate sorted attribute lists for the root node.

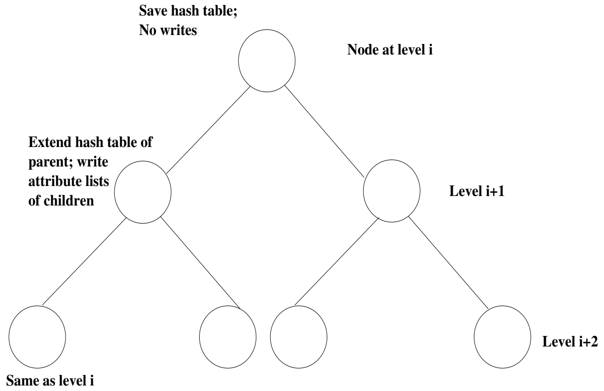


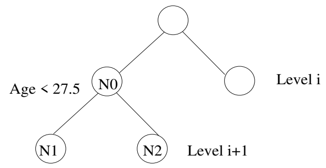
Figure 3. Write optimization for decision tree generation, using write skips every alternate level

6. Reduction in disk seeks while reading from disk

Using a specific layout of attribute records on disk, the number of seeks can be reduced. The formulation for SPRINT organizes the data so that four lists for each attribute type are used. There are two lists each for the consecutive levels of the tree. The two lists at each level are for the left and right child of the tree. When reading records for a particular node, corresponding portions of the attribute's left or right list are read. Figure 4 shows the data storage and layout in SPRINT. The attribute lists Age and CarType for node N0 are shown at the top. These lists are split using the test on age attribute. The new attribute lists for node N1 are shown at the bottom.

For any attribute, SPRINT stores the lists, corresponding to every node, contiguously in four files. We use a layout wherein all attribute lists for a particular node are stored contiguously as shown in Figure 5 for the layout of node N1. This leads to a total of four files (assuming breadth-first expansion; see note below about alternative expansion strategies), instead of four files for each attribute in case of SPRINT. For each list, all records of a particular node are present, then all records for the next node, and so on. Within the contiguous section for a particular node, there are contiguous sections for each of the attribute types for that node.

Let there be n leaves of the tree at level i and these are being split into children at level $i + 1$. Also, let k be the number of attribute types. SPRINT does $\theta(k)$ disk seeks for splitting each decision node of the tree. A seek has to be done for each attribute, since it is present in a different attribute list. Thus, for processing n nodes at a level, $\theta(k*n)$ disk seeks have to be done.



Level i left list for Age attribute

Age	Class	Rec. ID.
17	High	1
20	High	5
23	High	0
32	Low	4
43	High	2
68	Low	3

Level i left list for Car Type attribute

CarType	Class	Rec. ID.
family	High	0
sports	High	1
sports	High	2
family	Low	3
truck	Low	4
family	High	5

Level i+1 left list for Age attribute

Age	Class	Rec. ID.
17	High	1
20	High	5
23	High	0

Level i+1 left list for Car Type attribute

CarType	Class	Rec. ID.
family	High	0
sports	High	1
family	High	5

Contiguous sections for other nodes precede and follow the above section for node N1.

Contiguous sections for other nodes precede and follow the above section for node N1.

Figure 4. Data storage using attribute list in SPRINT

In contrast, using the new disk layout only a constant number of seeks have to be done for each node in the decision tree. These consist of seeking to the splitting attribute, and then seeking to the beginning of the contiguous section of the file, which stores all the attribute lists for a node. Therefore, only $O(n)$ seeks are required for processing n leaf nodes at a level. Figure 5 shows the attribute list layout for node N1. Note that lists for Age and CarType attributes are stored contiguously in a single file.

Level i+1 left list

Age	**Class**	**Rec. ID.**
17	High	1
20	High	5
23	High	0

CarType	**Class**	**Rec. ID.**
family	High	0
sports	High	1
family	High	5

All node N1 records are contiguous.
Records for other nodes precede and follow this contiguous section.

Figure 5. Data storage with attribute lists for a node stored contiguously.

Instead of generating a complete hash table for n nodes at a level, the hash table for a single node can be generated at a time. All the attribute lists for this particular node can be split. Using this methodology, if the hash table for a particular node fits into memory, it does not be need to be written to or read from the disk. When a node is split, its hash table can be simply deleted. This advantage of memory resident hash tables applies to any hash table that does fit into the memory even if it does not apply to all of them.

The second advantage of the disk layout is that the processing of each node is decoupled from processing of other nodes at the same level. Therefore, other expansion strategies, besides the breadth-first expansion strategy, can be followed. Note that there will be two files for each level. So if a depth-first strategy is followed and we are five levels deep in the tree, ten attribute files are needed.

7. Conclusions

We have described three optimizations for reducing I/O in decision-tree classifiers in the context of SPRINT. However, it is easy to verify that these optimizations are very general and can work with many other classifiers that can use disk-resident datasets including SCALPARC [JKK98], SPEC [SSHK97], and CLOUDS [ARS98]. Where parallel versions of these algorithms are available, the optimizations work on parallel processors as well.

References

[ARS98] K. Alsabti, S. Ranka, and V. Singh. CLOUDS: Classification for large or out-of-core datasets. *http://www.cise.ufl.edu/∼ranka/dm.html*, 1998.

[BFOS84] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth, Monterrey, CA, 1984.

[Cat91] J. Catlett. *Megainduction: Machine Learning on Very Large Databases. PhD thesis.* University of Sydney, 1991.

[CS93a] Philip K. Chan and Salvatore J. Stolfo. Experiments on multistrategy learning by metalearning. In *Proc. Second Intl. Conference on Info. and Knowledge Mgmt.*, pages 314–323, 1993.

[CS93b] Philip K. Chan and Salvatore J. Stolfo. Metalearning for multistrategy learning and parallel learning. In *Proc. Second Intl. Conference on Multistrategy Learning*, pages 150–165, 1993.

[DMT94] D.J. Spiegelhalter D. Michie and C.C. Taylor. *Machine Learning, Neural and Statistical Classification*. Ellis Horwood, 1994.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimizations and Machine Learning*. morgan-kaufman, 1989.

[JKK98] M.V. Joshi, G. Karypis, and V. Kumar. ScalParC: A new scalable and efficient parallel classification alg orithm for mining large datasets. In *Proc. of the International Parallel Processing Symposium*, 1998.

[Lip87] R. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4(22), April 1987.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[SAM96] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. of the 22nd VLDB Conference*, 1996.

[SSHK97] Anurag Srivastava, Vineet Singh, Eui-Hong Han, and Vipin Kumar. An efficient, scalable, parallel classifier for data mining. Technical Report TR-97-010,http://www.cs.umn.edu/∼kumar, Department of Computer Science, University of Minnesota, M inneapolis, 1997.

[WC88] J. Wirth and J. Catlett. Experiments on the costs and benefits of windowing in ID3. In *5th Int'l Conference on Machine learning*, 1988.