1998

CLOUDS: A Decision Tree Classifier for Large Datasets

Khaled Alsabti
Syracuse University, kaalsabt@top.cis.syr.edu

Sanjay Ranka
University of Florida, ranka@cise.ufl.edu

Vineet Singh
Hitachi America, Ltd., vsingh@hitachi.com

CLOUDS: A Decision Tree Classifier for Large Datasets

Khaled Alsabti *
Syracuse University
kaalsabt@top.cis.syr.edu

Sanjay Ranka
University of Florida
ranka@cise.ufl.edu

Vineet Singh
Hitachi America, Ltd.
vsingh@hitachi.com

October 27, 1998

Abstract

Classification for very large datasets has many practical applications in data mining. Techniques such as discretization and dataset sampling can be used to scale up decision tree classifiers to large datasets. Unfortunately, both of these techniques can cause a significant loss in accuracy. We present a novel decision tree classifier called CLOUDS, which *samples* the splitting points for numeric attributes followed by an *estimation* step to narrow the search space of the best split. CLOUDS reduces computation and I/O complexity substantially compared to state of the art classifiers, while maintaining the quality of the generated trees in terms of accuracy and tree size. We provide experimental results with a number of real and synthetic datasets.

Keywords: Classification, Decision Trees, Data Mining, Large Datasets, Sampling, Estimation, Gini index

1 Introduction

Classification is the process of generating a description or a model for each class of a given dataset. The dataset consists of a number of records, each consisting of several fields called *attributes*. Each record belongs to one of the given (categorical) classes. The set of records available for developing classification methods is generally decomposed into two disjoint subsets, a training set and a test

*The work of this author was done while he was visiting the department of CISE at University of Florida.

set. The former is used for deriving the classifier, while the latter is used to measure the accuracy of the classifier. The accuracy of the classifier is determined by the percentage of the test-dataset examples that are correctly classified [19]. Another measure of a classifier is the size of the model generated; compact models are preferred as they generalize better and they are easier to understand.

Classification has been a well-studied problem in the area of statistics and machine learning and has found applications in several disciplines. Some prominent methods are decision trees [2, 14], statistical methods [8, 12, 19], rule induction [7], genetic algorithms [6], and neural networks [12, 19]. Many of these techniques are iterative in nature and require several passes over the entire dataset; efficient computation using these methods requires that all the records be stored in the main memory. This limitation and other inherent properties of these techniques generally limit them to classifying small datasets.

The datasets for data-mining applications are large and may involve several million records. Further, each record typically consists of ten to hundreds attributes. Using large datasets usually improves the accuracy of the classifier [3, 5], but the enormity and complexity of the data involved in these applications makes the classification task computationally intensive. Since datasets are large, they cannot reside completely in the main memory, which makes I/O a significant bottleneck. Performing classification for such large datasets requires the development of new techniques that limit accesses to the secondary memory in order to minimize the overall execution time. Techniques such as discretization and dataset sampling can be used to scale up decision tree classifiers to large datasets. Unfortunately, both of these techniques can cause a significant loss in accuracy [3].

The decision-tree classifiers SLIQ and SPRINT have been shown to achieve good accuracy, compactness and efficiency for very large datasets [15, 17], although the latter has substantially superior computational characteristics. Deriving a typical decision tree classifier from the training set consists of two phases:

- Construction phase: The initial decision tree is constructed in this phase based on the entire training set. It requires recursively partitioning the training set into two or more subpartitions using a *splitting* criteria until a *stopping* criteria is met. A hierarchical tree structure is generated with the root representing the entire dataset.
- Pruning phase: The tree constructed in the previous phase may not result in the best generalization due to overfitting. The pruning phase removes some of the lower branches and

nodes to improve the generalization capabilities (accuracy) of the classifier. Some techniques might replace a node by one of its child nodes [14].

CART and SPRINT use the *gini* index to derive the splitting criterion at every internal node of the tree. The use of the *gini* index to derive the splitting criterion is computationally challenging. Efficient methods such as SPRINT require sorting along each of the attributes, which requires the use of memory-resident hash tables to split the sorted attribute lists at every level. For datasets that are significantly larger than the available memory, this will require multiple passes over the entire dataset. In this paper we present a simple scheme that eliminates the need to sort along each of the attribute lists, by exploiting the following properties of the *gini* index for a number of real datasets:

1. The value of the *gini* index along a given attribute generally increases or decreases slowly. The number of good local minima is small compared to the size of the entire dataset. This is especially true for attributes along which the best splitting point is obtained.

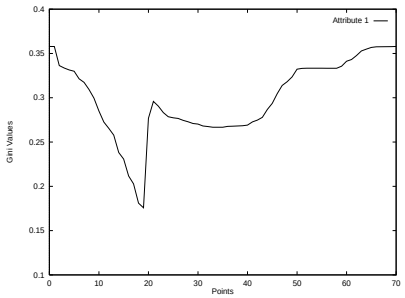
2. The minimum value of the *gini* index for the splitting point along the splitting attribute is (relatively) significantly lower than most of the other points along the splitting attribute as well as other attributes.

Figure 1 gives the value of the *gini* index along each of the nine numeric attributes of the Shuttle dataset.

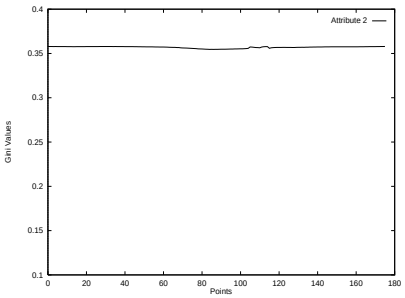
We show that the above properties can be used to develop an I/O and computationally efficient method for estimating the split at every internal node. Experimental results on real and synthetic datasets using our method show that in most cases the splitting point derived has the same *gini* index as the one obtained by sorting along each of the attributes.

Using this new method for splitting at every internal node, we have developed a new decision-tree classifier called CLOUDS (Classification for Large or OUt-of-core DataSets). It *samples* the splitting points for numeric attributes, followed by an *estimation* step to narrow the search space of the best split. CLOUDS is shown to have substantially lower computation and I/O requirements as compared to SPRINT for a number of real and synthetic datasets. The accuracy and compactness of the decision trees generated is the same or comparable as those of CART, C4, and SPRINT.

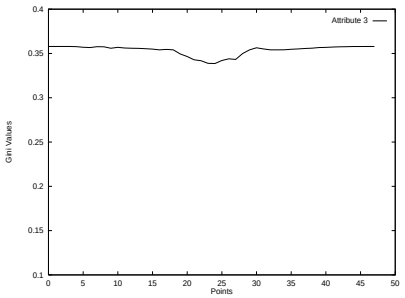
The rest of the paper is organized as follows. In Section 2 we briefly describe decision-tree



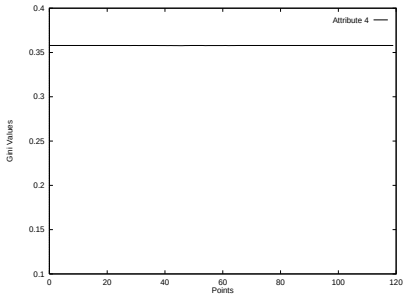
Attribute 1



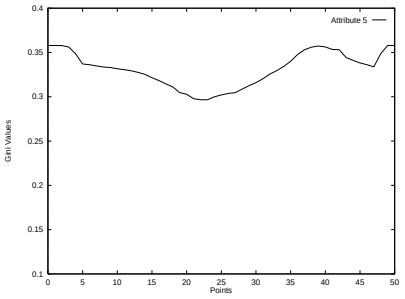
Attribute 2



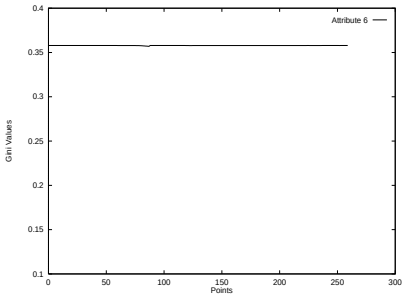
Attribute 3



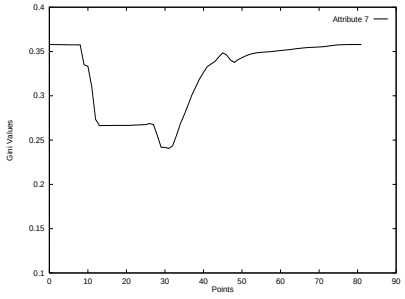
Attribute 4



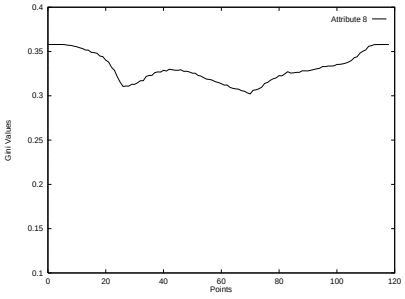
Attribute 5



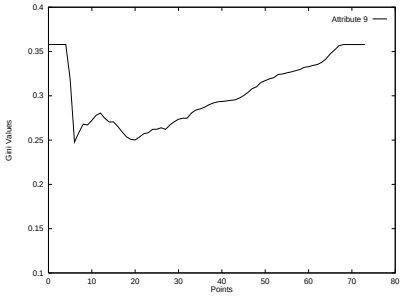
Attribute 6



Attribute 7



Attribute 8



Attribute 9

Figure 1: The value of *gini* index along the nine attributes of the Shuttle dataset

classifiers, the *gini* index, and related work. Our algorithm for estimating the splitting point with minimum *gini* index is presented in Section 3. The CLOUDS algorithms are described in Section 4, and we present our conclusion in Section 5.

2 Decision Tree Classifiers

The derivation of a decision tree classifier consists of two phases: construction phase and pruning phase. The former requires substantially more computational time and is the main focus of this work. The construction phase requires that the training set be recursively partitioned into two or more subpartitions until a stopping criteria is met, e.g., each subpartition dominantly (or entirely) consists of examples of one class. This involves applying a splitting criterion to every internal node of the tree. These splitting criteria are determined by applying some splitting function, e.g., the *gini* function. A decision tree is a hierarchical tree structure that is generated with the root representing the entire dataset. Figure 2 shows an example of such a decision tree. Similar trees are constructed using traditional decision tree classifiers such as CART [2] and C4.5 [14]. Figure 3 shows a high-level description of the traditional building phase of a binary (i.e., two partitions), univariate (i.e., one attribute is used for deriving the splitting criteria) decision tree.

The *gini* index function can be used to evaluate the goodness of all the potential split points along all the attributes [2]. Consider a dataset S consisting of n records, each belonging to one of the c classes. The *gini* index for set S is defined as:

$$gini(S) = 1 - \sum_{j=1}^{c} p_j^2 \tag{1}$$

where p_j is the relative frequency of class j in S.

If S is partitioned into two subsets S_1 and S_2, the index of the partitioned data $gini^D(S, sc)$ can be obtained by:

$$gini^D(S, sc) = \frac{n_1}{n} gini(S_1) + \frac{n_2}{n} gini(S_2) \tag{2}$$

where n_1 and n_2 are the number of examples of S_1 and S_2, respectively, and sc is the splitting criterion. One of the major advantages of the *gini* index is that its calculation requires only the class value distribution at each internal node of the tree. The *gini* index has been shown to be effective in determining splits that lead to good decision trees [2, 15, 17]. Many other splitting functions have been proposed in the literature, e.g. the towing criterion [2] and the gain ratio [14].

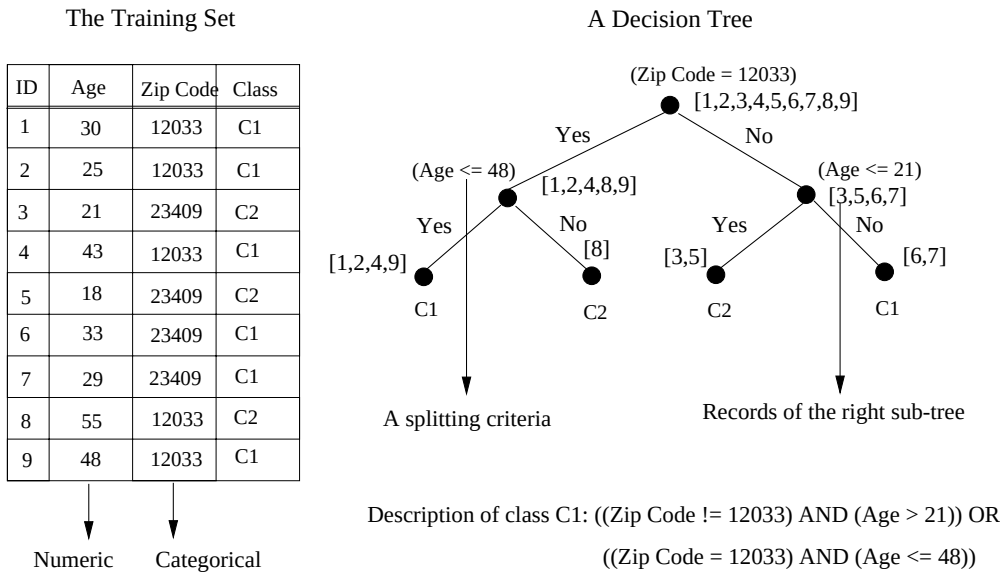


Figure 2: An example of a decision tree

---

function Decision-Tree-Classifier(Training set T)

if T meet the stopping criteria *then*
return

for each attribute A_i *do*
Find the value(s) V_i of A_i which results in the best split

Use the attribute A which gives best split among all the attribute to partition T into T_1 and T_2

Decision-Tree-Classifier(T_1)

Decision-Tree-Classifier(T_2)

Figure 3: A high-level description of the traditional building phase of decision trees

2.1 Related Work

Many techniques have been proposed to scale up the decision tree classifiers for large datasets. There are mainly two approaches: *exact* and *approximate*. Discretization and sampling are approximate techniques, whereas SPRINT is an exact technique.

Sampling the dataset randomly is a simple technique, and it will be studied in the next section. *Windowing* technique has been used by C4.5 [14], which works as follows. A small sample is drawn from the dataset, that is used to build an initial tree. This sample is augmented with the examples that were misclassified to build a new tree. This process is repeated for a number of iterations. *Stratification* is a selective sampling method in which the classes have approximately the same distribution [3]. The above three types of sampling are performed prior to the construction phase. A simple random-sampling technique has been proposed in [2] and evaluated in [3]. This method uses a sample for building the top levels of the tree and then switches to an exact algorithm for the remaining levels.

Discretization is the process of transforming the numeric attribute into an order discrete attribute (with a small number of values), by which the sorting operation is avoided in the construction phase. The static discretization techniques perform the discretization prior to building the tree. Three such techniques have been studied in [3], two of which are *class-blind* methods that perform the discretization using only the values of the continuous attribute. The first is based on an equal-width histogram that divides the range into an approximately equal widths, and the other is based on an equal-depth histogram (quantiling) that divides the range into sub-ranges of an approximately equal number of elements. A discretization algorithm, *D-2*, which discretizes the numeric attributes using the class distributions and the values, has been proposed in [3]. It entails sorting all the continuous attributes. ChiMerge and Chi2 algorithms use $\mathcal{X}^2$ statistics to perform the discretization [9] and [10], respectively. Both of these algorithms use a merging technique in a bottom-up fashion.

Peepholing method has been proposed to scale up the decision tree for large datasets [3]. It iterates several times to reduce the number of attributes that are considered for deriving the split point. Further, it narrows the range of the numeric attributes to which the optimal split point is presumed to lie. Peepholing uses a sample of the dataset at each iteration to perform the above process. The sample is enlarged as the algorithm iterates. This algorithm requires several passes

to derive the split point. It has been incorporated into C4.5 for, and evaluated for large in-memory datasets.

The SPEC algorithm is an approximation technique designed to deal with disk-resident data. It uses a clustering technique to derive the split point of the numeric attributes [18].

The SPRINT algorithm is an exact technique that has been proposed to scale up the decision tree for disk-resident data [17]. This algorithm is used to compare the performance of our algorithm (Section 4). A high-level description of the algorithm for building the initial tree using SPRINT is given in Figure 4.

```
function Decision-Tree-Classifier(Level l)
  for each node node in level l do
      if the stopping criteria is not met then
          for each attribute a do
              for each value v in attribute list do
                  Update the class histogram
                  if a is an numeric attribute then
                      Compute gini^D(S, a ≤ v)
              if a is a categorical attribute then
                  Apply the hybrid algorithm to find subset of a which gives the best split
              Find the value(s) v of a which results in the best split
          if there is at least one split then partition the records among the child nodes using
             the best split
  if a node at level l has been partitioned then
      Decision-Tree-Classifier(l+1)
```

Figure 4: A high-level description of the building phase of SPRINT

The SPRINT algorithm employs a pre-sorting technique to avoid the sorting operation at every node of the tree. Initially, SPRINT creates an attribute list for each attribute. Each row of an attribute list corresponds to a unique record; it contains the record number (or *rid*), the attribute value (for the attribute in question), and the class label of the record. For numeric attributes, the attribute lists are sorted by attribute values. SPRINT builds the tree in a *breadth-first* manner. The construction phase of the SPRINT algorithm for each internal node consists of two steps. The first step requires that the split points be evaluated and that attribute lists corresponding to the internal node be read. SPRINT assumes that the split can be performed along one of the attribute dimensions, i.e., the univariate split. The splitting attribute a is determined using the *gini* index. The splitting criterion of an attribute depends on the attribute type:

1. For a numeric attribute, SPRINT uses a binary split of the form $a \leq v$, where v is a real number and a is an attribute. For numeric attribute a with n distinct values, there are $n-1$ possible splits. The *gini* index is calculated at each of these $n-1$ points and the minimum value is chosen. Since the original attribute list is sorted at the root, subsequent subsets at lower levels can maintain this sorted property without additional sorting.

2. For categorical attributes, SPRINT uses a split of the form $b \in D$, where $D \subset \mathcal{D}$ and $\mathcal{D}$ is the domain of the categorical attribute b; for an attribute with m possible values, there are 2^m possible splits. A hybrid algorithm is used to evaluate splits for categorical attributes. In this hybrid algorithm a brute-force method of exhaustive search is used if the number of possible values is less than some threshold. This method evaluates all possible splits of the form $b \in D$ to find the split that gives the minimum *gini* value. If the value 2^m is very large, a greedy algorithm is used that does not evaluate all possible splits [4, 15]. This algorithm starts with an empty set as D and a candidate set C. Initially, C is set to $\mathcal{D}$. It proceeds by moving one candidate from C to D until no more suitable candidates can be found. The added candidate should satisfy the following:

   (a) Adding this candidate to D yields a smaller value of the *gini* index, and

   (b) it has the smallest *gini* index value among all the other candidates.

   If there is no such candidate, the algorithm stops evaluating splits.

The second step requires partitioning the data using the splitting criterion, which requires reading the attribute list of the splitting attribute and partitioning it into two subsets based on the splitting criterion. While reading the attribute list, a hash table is created with two fields *rid* and *subset*. The *rid* field is used as the key for hashing. This hash table is used to partition the other attribute lists by hashing each *rid* of the current record and determining the destination subset (right or left). The hash table is required, because the ordering of the data in different attribute lists is different (since they are sorted with different attributes). The partitioning process requires at least one read and one write for each of the attribute lists. If the size of the hash table is larger than the available memory, this process has to be staged and may require several reads and writes (equal to the number of stages used) of the hash table.

The pruning phase uses an algorithm based on the Minimum Description Length (MDL) principle for pruning nodes [16]. This algorithm was proposed in [15].

The main drawbacks of the SPRINT algorithm are the following:

1. The initial training dataset is transformed into another representation. The size of the new dataset is expected to be at least twice as large as the initial dataset for most datasets (see Section 4).
2. It requires sorting the entire training dataset.
3. It evaluates the *gini* index for each distinct value along all numeric attributes.
4. For cases where the hash table cannot fit in the main memory, the partitioning process will requires several passes over the datasets.

3 Estimating the Splitting Point for Numeric Attributes

In this section we present two new methods for estimating the splitting point of the numeric attributes. These methods are used by CLOUDS at each node of the tree. We compare the new methods with a direct method and a simple sampling method.

Direct Method (DM) This method sorts the values of each numeric attribute, and evaluates the *gini* index at each possible split point to find the splitting point with the minimum *gini* index. The sorting operation requires $O(n \lg n)$ time, where n is the number of values. It also requires several passes of I/O for disk-resident data. Evaluating the *gini* index at each possible split point is very expensive.

Dataset Sampling (DS) This method draws a random subset of the whole dataset. The DM method is applied to this sample to build the classifier. In order to study the quality of this method, the *gini* index of the splitter (the point used to partition the data) is obtained using the whole dataset.

Sampling the Splitting Points In this work we present two such methods, the first of which, Sampling the Splitting points (SS), derives the splitter from a limited number of splitting points. The second method, SSE (Sampling the Splitting points with Estimation), improves upon the SS

by estimating the *gini* values in order to narrow the search space of the best split, and it is designed to derive the best or near-best splitter with respect to the *gini* index. Both methods evaluate the *gini* index at only a subset of splitting points along each numeric attribute.

3.1 Sampling the Splitting points

In the SS method the range of each numeric attribute is divided into q intervals using a quantiling-based technique. Each interval contains approximately the same number of points. For each numeric attribute the *gini* index is evaluated at the interval boundaries to determine the minimum *gini* value ($gini_{min}$) among all interval boundaries of the numeric attributes and across all the categorical attributes. The split point with $gini_{min}$ will be used as a splitter. The SS method requires one pass over the dataset to derive the splitting point.

3.2 Sampling the Splitting points with Estimation

The SSE method divides the range of each numeric attribute and finds the minimum *gini* value ($gini_{min}$), as in the SS method. Further, it estimates a lower bound $gini^{est}$ of the *gini* value for each interval. All the intervals with $gini^{est} \geq gini_{min}$ are eliminated (prune) to derive a list of potential candidate intervals (*alive* intervals). For an *alive* interval we evaluate the *gini* index at every distinct point in the interval to determine the splitter. This may require another pass over the entire dataset.

The performance of the SSE method depends on the quality of the estimated lower bound of the *gini* index and on the fraction of elements in the *alive* intervals; this fraction is given by the *survival* ratio. Since the *gini* index is calculated for each of the splitting points in these intervals, the computational cost of this phase is directly proportional to the *survival* ratio. The SSE method estimates the minimum value of the *gini* index in a given interval $[v_l, v_u]$ using the statistics of the interval endpoints. In the following we describe our estimation heuristic. We use the following notation:

x_i — the number of elements of class i that are less than or equal to v_l

y_i — the number of elements of class i that are less than or equal to v_u

c_i — the total number of elements of class i

n_l — the number of elements that are less than or equal to v_l $(= \sum_{i=1}^{c} x_i)$

n_u — the number of elements that are less than or equal to v_u $(= \sum_{i=1}^{c} y_i)$

n — the size of the dataset

c — the number of classes

The $gini^D$ at point v_l of a numeric attribute a for dataset S is given by:

$$gini^D(S, a \leq v_l) = \frac{n_l}{n}(1 - \sum_{i=1}^{c}(\frac{x_i}{n_l})^2) + \frac{n - n_l}{n}(1 - \sum_{i=1}^{c}(\frac{c_i - x_i}{n - n_l})^2) \tag{3}$$

We use a hill-climbing algorithm to estimate the lower bound of the *gini* index in the given interval. This algorithm makes a series of decisions to estimate the lower bound, with each decision based on the values of *gradient* along a subset of the c classes. Gradients along a subset of these c classes are calculated and the direction along the minimum gradient is chosen. The value of the gradient along x_i is given by[1]:

$$\frac{\partial gini^D(S, a \leq v_l)}{\partial x_i} = \frac{2}{n_l(n - n_l)}(c_i\frac{n_l}{n} - x_i) - \frac{1}{n}(\frac{1}{(n - n_l)^2}\sum_{i=1}^{c}(c_i - x_i)^2 - \frac{1}{n_l^2}\sum_{i=1}^{c}x_i^2) \tag{4}$$

We calculate $gini^{est}$ for interval $[v_l, v_u]$ from left to right as well as right to left. Let class j give the minimum gradient at v_l. Then,

$$\frac{2}{n_l(n - n_l)}(c_j\frac{n_l}{n} - x_j) \leq \frac{2}{n_l(n - n_l)}(c_i\frac{n_l}{n} - x_i) \qquad \forall\ 1 \leq i \leq c \tag{5}$$

From formula (5), we obtain :

$$c_j\frac{n_l}{n} - x_j \leq c_i\frac{n_l}{n} - x_i \qquad \forall\ 1 \leq i \leq c \tag{6}$$

Thus,

$$c_j\frac{n_l + 1}{n} - (x_j + 1) \leq c_i\frac{n_l + 1}{n} - x_i \qquad \forall\ 1 \leq i \leq c \tag{7}$$

Assuming that all the points for the class with the minimum gradient appear consecutively, the gradient of the class with the minimum value will remain the minimum for the next splitting point;

[1]Since the $gini^D$ is a function in a (left-hand side), the derivative should be performed with respect to a. However, the $gini^D$ is defined in terms of x_i's (right-hand side). Mathematically speaking, the $gini^D$ defines x_i's, which are the input to an implicit function defined by the right-hand side of formula (3). For the sake of simplicity, we derive the derivative directly with respect to x_i's (formula (4)).

we do not need to evaluate formula (4) for each point in the interval. Our assumption is based on the intuition that the impurity of the potential partitions will decrease the most (assuming that all the points for the class with the minimum gradient appear consecutively) by which the *gini* value will decrease the most. Thus, the time requirements are proportional to the number of classes rather than the number of split points in the interval. Figure 5 gives a high-level description of our hill-climbing algorithm, *Est-GiniLR*, for determining $gini^{est}$ from left to right.[2] It starts from the left boundary and finds the class with the minimum gradient. Then it evaluates the *gini* index at a new splitting point, which is at the old splitting point (left boundary) plus the number of elements in the interval with the class with the minimum gradient. If the new *gini* value is less than the cuurent $gini^{est}$, it becomes the new $gini^{est}$ and the process is repeated for the remaining classes (and smaller interval). The estimation function for the other direction, *Est-GiniRL*, is similar to the *Est-GiniLR* function except that we need to exchange y_i's by x_i's, and use the maximum instead of the minimum (in step 4).

We determine $gini^{est}$ as follows:

$$gini^{est} = \min(\textit{Est-GiniLR}, \textit{Est-GiniRL}, gini^{D}(S, a \leq v_l), gini^{D}(S, a \leq v_u)) \tag{8}$$

The SSE method can be modified to further prune the intervals (from the *alive* intervals) as we update the global minimum *gini* based on computation of *alive* intervals processed at a given point. Further, it can prioritize the evaluation of the *alive* intervals such that we start with the intervals with the estimated least *gini* value. These optimizations have been incorporated in the SSE method.

One advantage of the SSE method is that it dynamically performs *feature selection*. Real datasets generally contain *irrelevant* attributes that affect the overall performance of the learning system. It is desirable to remove these irrelevant attributes prior to deriving the splitter. Our method would prune all the intervals of the irrelevant attributes. The SS method is simpler to implement and it requires less computation and I/O times for determining the splitter than the SSE method. On the other hand, the SSE method should generate more accurate splitters with respect to the *gini* index. Clearly, there is a trade-off between quality and computational and I/O requirements. We will study the effect of the two algorithms on the overall performance in Section 4.

[2]The second term for the gradient along a given class (formula 4) is the same for all the classes. Thus only the first term, $\frac{2}{n_l(n-n_l)}(c_i\frac{n_l}{n} - x_i)$, can be used to determine the class with minimum gradient.

function Est-GiniLR(n, n_l,$\overline{c}$,$\overline{x}$,$\overline{y}$)

Step 1: Set the candidate set $CS = \{1, 2, \ldots, c\}$

Step 2: Set $gini^{est} = 1$

Step 3: Evaluate Formula (4) for all classes $\in CS$. Let the value along class i be given by d_i

Step 4: Let d_j be the minimum among all d_i's

Step 5: Evaluate the *gini* index on vector x replacing x_j by y_j.

Step 6: if the new *gini* value is less than $gini^{est}$ then:

Step 6.1: Set $gini^{est}$ to the new *gini* value

Step 6.2: Remove class j from CS

Step 6.3: Go to Step 3

Figure 5: A high-level description of the *Est-GiniLR* function

3.3 Experimental Results

We experimentally evaluated the three methods for a number of real and synthetic datasets. Table 1 gives the characteristics of these datasets. The first four datasets are taken from the STATLOG project, which has been a widely used benchmark in classification.[3] The "Abalone," "Waveform," and "Isolet" datasets can be found in [13]. The "Synth1" and "Synth2" datasets have been used in [15, 17] for evaluating SLIQ and SPRINT; they have been referred to as the "Function2" dataset.

The main parameter of our algorithm is the number of intervals used along each of the numeric attributes. We studied the following characteristics of our algorithms for different numbers of intervals:

1. We compare the value of the *gini* index of the splitters generated by the SS and SSE methods to the *gini* index of the splitting point using the DS and DM methods.

2. For SSE, we obtained the percentage of intervals for which the estimated value was higher than the actual value. These are measured by the number of *misses* with our method. For the missed intervals, we also measured the maximum and the average difference in the estimated

[3]For more details about the STATLOG datasets, the reader is referred to http://www.kdnuggets.com/.

Dataset	No of examples	No. of attributes	No. of numeric attributes	No. of classes	Max. no. of distinct values
Letter	20,000	16	16	26	16
Satimage	6,435	36	36	6	256
Segment	2,310	19	19	7	1,769
Shuttle	58,000	9	9	7	260
Abalone	4,177	8	7	29	2,062
Waveform-21	5,000	21	21	3	884
Waveform-40	5,000	40	40	3	889
Isolet	7,797	617	617	26	4,471 *
Synth1	400,000	9	6	2	310,968
Synth2	800,000	9	6	2	523,639

Table 1: Description of the datasets

value and the actual minimum for that interval. Further, we obtained the fractional number of elements that survived for the next pass (*survival* ratio).

Table 2 shows the results of the estimated *gini* index using the DS method. The exact *gini* value was derived using the DM method. We used datasets for three random samples chosen from the dataset and have reported the minimum and the maximum *gini* index obtained. These results show that the DS method, in the worst case, may obtain splitters with relatively larger values of *gini* index as compared to a direct calculation. This may be partly due to the small size of the datasets. One may expect better accuracy for large datasets. Our results, described in the next section, show that this can have a negative impact on the overall classification error.

Table 3 shows the estimated *gini* index using the SS method and the DM method. These results show that the SS method usually missed the *exact gini* index for the test datasets; the difference between the exact and estimated *gini* was not substantial. The "Letter" dataset has 16 distinct values along each of the numeric attributes. The SS method, with number of intervals greater than 16, should work well for such a dataset. The same holds true for the "Satimage" and "Shuttle" datasets, as they have a small number of distinct values. These results show that the accuracy of the SS method decreases with the decrease in the number of intervals.

Dataset	Exact	10%		15%		20%	
	gini	Min	Max	Min	Max	Min	Max
Letter	0.940323	0.940323	**0.942294**	0.940323	**0.942326**	0.940323	**0.942326**
Satimage	0.653167	0.653167	**0.659033**	0.653167	**0.659033**	0.653167	**0.659033**
Segment	0.714286	0.714286	**0.717774**	0.714286	**0.714788**	0.714286	**0.721099**
Shuttle	0.175777	0.175777	0.175777	0.175777	0.175777	0.175777	0.175777
Abalone	0.862017	**0.862579**	**0.866956**	**0.862579**	**0.867807**	**0.862114**	**0.866992**
Waveform-21	0.546150	**0.547439**	**0.552724**	**0.547083**	**0.548120**	**0.548144**	**0.550018**
Waveform-40	0.541134	**0.548980**	**0.554132**	**0.543133**	**0.554132**	**0.542480**	**0.550520**
Isolet	0.926344	**0.926552**	**0.932994**	0.926344	**0.926883**	0.926344	**0.927038**

Table 2: Exact and estimated *gini* using DS for different sampling ratios (3 runs)

The results in Table 4 show the estimated *gini* index using the SSE method and the DM method. These results show that the SSE method missed the exact *gini* index only for a few cases. Further, the estimated *gini* generated by the SSE method is more accurate than those of the SS method (for the missed ones). Tables 3 and 4 show that the accuracy of the SS method is more sensitive to the number of intervals, which makes the SSE method more scalable. This experiment shows that the SSE method is a definite improvement, more scalable, and robust than the SS method.

The results in Table 5 show that the SSE method can estimate the lower bound accurately. In all cases the fraction of misses was small. Further, for the missed intervals, the difference between the actual minimum and estimated minimum was extremely small (Tables 6 and 7). From these results, 50–200 intervals are sufficient for achieving a very good performance in all the test datasets.

The results in Table 8 show that the *survival* ratio of the SSE method is less than 10% for all the test datasets using more than 10 intervals, and less than a few percent using more than 100 intervals. For three datasets, "Abalone," "Waveform-21," and "Waveform-40," and a very small number of intervals, the *survival* ratio is very large. As expected, the *survival* ratio generally decreased with an increase in the number of intervals. For a large number of intervals, the number of data points in the *alive* intervals is around two orders of magnitude smaller than the points in all the intervals.

The above results clearly demonstrate the effectiveness of the SSE method; it can achieve a

Dataset	Exact *gini*	Number of Intervals					
		200	100	50	25	10	5
Letter	0.940323	0.940323	0.940323	0.940323	0.940323	**0.941751**	**0.941751**
Satimage	0.653167	0.653167	0.653167	**0.654601**	**0.654601**	**0.654601**	**0.682317**
Segment	0.714286	0.714286	0.714286	**0.717774**	0.714286	**0.721154**	**0.740018**
Shuttle	0.175777	0.175777	0.175777	0.175777	0.175777	**0.181001**	**0.238166**
Abalone	0.862017	**0.862477**	**0.862697**	**0.862752**	**0.862907**	**0.862724**	**0.865768**
Waveform-21	0.546150	**0.547003**	**0.547464**	**0.547353**	**0.547424**	**0.548090**	**0.550608**
Waveform-40	0.541134	**0.541680**	**0.541943**	**0.542257**	**0.542894**	**0.543224**	**0.543224**
Isolet	0.926344	**0.927403**	**0.927290**	**0.927424**	**0.927403**	**0.929721**	**0.930125**

Table 3: The exact *gini* and estimated *gini* based on the SS method using a different number of intervals

Dataset	Exact *gini*	Number of Intervals					
		200	100	50	25	10	5
Letter	0.940323	0.940323	0.940323	0.940323	0.940323	**0.941751**	0.940323
Satimage	0.653167	0.653167	0.653167	0.653167	0.653167	0.653167	0.653167
Segment	0.714286	0.714286	0.714286	0.714286	0.714286	**0.715799**	**0.721099**
Shuttle	0.175777	0.175777	0.175777	0.175777	0.175777	0.175777	0.175777
Abalone	0.862017	0.862017	0.862017	0.862017	0.862017	0.862017	0.862017
Waveform-21	0.546150	0.546150	0.546150	0.546150	0.546150	0.546150	0.546150
Waveform-40	0.541134	0.541134	0.541134	0.541134	0.541134	0.541134	0.541134
Isolet	0.926344	**0.926807**	**0.926807**	0.926344	**0.927138**	0.926344	0.926344

Table 4: The exact *gini* and estimated *gini* based on the SSE method using a different number of intervals

Dataset	Number of Intervals					
	200	100	50	25	10	5
Letter	0.095	0.038	0.061	0.100	0.132	0.021
Satimage	0.006	0.005	0.002	0.000	0.025	0.006
Segment	0.055	0.051	0.042	0.044	0.080	0.195
Shuttle	0.045	0.083	0.068	0.059	0.081	0.111
Abalone	0.017	0.009	0.009	0.000	0.043	0.029
Waveform-21	0.017	0.017	0.009	0.004	0.000	0.000
Waveform-40	0.020	0.015	0.009	0.003	0.000	0.000
Isolet	0.208	0.168	0.110	0.076	0.068	0.057

Table 5: The fractional misses of the SSE method using a different number of intervals

Dataset	Number of Intervals					
	200	100	50	25	10	5
Letter	0.002587	0.002587	0.006108	0.008234	0.009153	0.000022
Satimage	0.000551	0.000551	0.000882	0.000882	0.010371	0.006121
Segment	0.002355	0.002355	0.002121	0.005901	0.040869	0.067179
Shuttle	0.000175	0.001892	0.007662	0.007397	0.008198	0.115875
Abalone	0.000094	0.000018	0.000050	0.000050	0.000342	0.000588
Waveform-21	0.000966	0.000966	0.000275	0.000664	0.000664	0.000664
Waveform-40	0.000867	0.000241	0.000285	0.000217	0.000217	0.000217
Isolet	0.002592	0.002592	0.003889	0.006350	0.009310	0.010187

Table 6: The maximum value of the fractional difference between the estimated (generated by the SSE method) and exact minimum *gini* for the missed intervals

Dataset	Number of Intervals					
	200	100	50	25	10	5
Letter	0.001299	0.002587	0.003096	0.002566	0.004067	0.000022
Satimage	0.000301	0.000338	0.000882	–	0.005459	0.006121
Segment	0.000240	0.000318	0.000602	0.001490	0.017226	0.026430
Shuttle	0.000085	0.000392	0.001433	0.001635	0.003010	0.046172
Abalone	0.000034	0.000013	0.000047	–	0.000268	0.000588
Waveform-21	0.000137	0.000119	0.000080	0.000374	–	–
Waveform-40	0.000071	0.000045	0.000053	0.000130	–	–
Isolet	0.000022	0.000031	0.000052	0.000116	0.000290	0.000743

Table 7: The average value of the fractional difference between the estimated (generated by the *SSE* method) and exact minimum *gini* for the missed intervals

Dataset	Number of Intervals					
	200	100	50	25	10	5
Letter	0.000	0.000	0.000	0.002	0.029	0.174
Satimage	0.000	0.000	0.001	0.006	0.022	0.092
Segment	0.000	0.000	0.003	0.000	0.011	0.011
Shuttle	0.000	0.000	0.000	0.003	0.001	0.032
Abalone	0.005	0.009	0.027	0.085	0.361	0.436
Waveform-21	0.007	0.012	0.022	0.036	0.143	0.347
Waveform-40	0.001	0.002	0.005	0.014	0.062	0.175
Isolet	0.000	0.000	0.000	0.001	0.002	0.019

Table 8: The fractional *survival* ratio using a different number of intervals with the *SSE* method

very good estimate of the splitter with minimum *gini* index and it requires a limited amount of memory. Further, it has a better performance than algorithms that utilize only a small subset of the overall sample for obtaining the splitters.

For each point in an *alive* interval the class information, as well as the value of the numeric attribute corresponding to the *alive* interval, needs to be stored and processed. Assuming that all categorical and numeric attributes as well as class field are of the same size (in bytes), the amount of data to be processed for all the *alive* intervals is proportional to $O(2 \times SurvivalRatio \times NumberOfNumericAttributes \times Numberof Records)$. This corresponds to $(2 \times SurvivalRatio \times \frac{NumberOfNumericAttributes}{NumberOfAllAttributes+1})$ fraction of the original dataset.

The size of the domains of the numeric attribute does not have a major impact on the performance of algorithm, due the intervals generation method. However, the number of attributes, the number of distinct values, and the class membership affect the overall performance. The conducted experiments showed that 50–200 intervals are sufficient to achieve an acceptable performance. However, we believe that the number of intervals might affect the overall performance for some datasets. It is not easy to determine the optimal value of the number of intervals. Further investigation is needed to determine a reasonable value for the number of intervals q. We will discuss a method for determining the value of q in Section 4.

4 The CLOUDS Algorithm

In this section we describe our classifier in more detail. A high-level description of the CLOUDS algorithm is given in Figure 6. The CLOUDS algorithm uses a breadth-first strategy to build the decision tree and uses the *gini* index for evaluating the numeric attributes. It uses either the SS method or the SSE method to derive the splitter at each node of the tree. The processing of categorical attributes (Step 1) and numeric attributes (Steps 2, 4, and 5) are different, since they have different splitting criteria. Below, we describe the details of the processing required to derive the splitting criterion for both types of attributes. We also describe the partitioning step for decomposing an internal node (Step 6).

Categorical Attribute CLOUDS evaluates the split points for categorical attributes as in SPRINT. As described in Section 2, the processing of categorical attributes does not require sort-

function Decision-Tree-Classifier(Level l)

for each node *node* in level l *do*

if the stopping criteria is not met *then*

Step 1:

for each categorical attribute a *do*

for each value v in attribute list *do*

Update the count matrix

Find subset of a which gives the best split

Step 2:

for each numeric attribute a *do*

Divide the range of attribute a into q intervals

for each interval $[v_l..v_r]$ *do*

Compute $gini^D$ for interval boundary v_l

Step 3:

Find the minimum $gini$ ($gini_{min}$) across all the categorical attributes and the interval boundaries for numeric attributes.

Step 4: (For the SSE methods only)

for each numeric attribute a *do*

for each interval *do*

Estimate a lower bound of the $gini$ index $gini^{est}$ in this interval

if $gini^{est} \geq gini_{min}$ *then* prune this interval

Step 5: (For the SSE methods only)

for each numeric attribute a *do*

for each *alive* interval I *do*

Update the class histogram

for each value $v \in I$ *do*

Compute $gini^D(a \leq v)$

Find the value v of a which results in the best split

Step 6:

if there is at least one split *then*

Partition the records between the child nodes using the best split

partition *node* into its child nodes

if all nodes at level l have been partitioned *then* Decision-Tree-Classifier(l+1)

Figure 6: CLOUDS: A high-level description of the construction phase

ing, but it requires the frequency of each pair of $(value, class)$. For the categorical attribute a with m values, a matrix ($CountMatrix$) of size $m \times c$ is created, where c is the number of classes. Entry $CountMatrix[i, j]$ gives the number of examples belonging to class j and having value i for attribute a. This matrix is used by the hybrid algorithm (described in Section 2) for deriving the splitting criterion along attribute a.

Numeric Attribute For each node at level l of the tree, the range of each numeric attribute is partitioned into q intervals. A random sample S of size s is derived from the dataset. This step requires one read operation of the whole dataset, which is needed only at the root level. The samples for the lower levels are obtained using the sample for the higher levels; $q - 1$ points are derived from S by regular sampling. This corresponds to q regularly spaced quantiles of the set S. If q is much smaller than S, we expect that the q intervals have close to $\frac{n}{q}$ records with high probability, where n is the total number of records.[4] The range of interval i is $[z_{i-1}..z_i)$; z_0 and z_q are the minimum and the maximum value of the attribute, respectively, and z_i is the i^{th} quantile of the sample points.

The dataset is read and the class frequencies are computed for each of the q intervals as follows:

> For each value v with class i, determine the corresponding interval using binary search and update the frequency of class i.

At this stage the *gini* index is evaluated for each interval boundary using formula (3) and the minimum value of the *gini* index among all the intervals and across all the numeric attributes is found. Let this value be given by $gini_{min}$.[5]

The CLOUDS algorithm (with the SSE method) estimates a lower bound for the *gini* index $gini^{est}$ for each interval using the functions *Est-GiniLR* and *Est-GiniRL*, which were described in the previous section. The intervals with $gini^{est}$ greater than or equal to $gini_{min}$ are eliminated. The remaining (*alive*) intervals are used in the next step of the algorithm to determine the splitter. For the *alive* intervals along each numeric attribute (using the SSE method), the class frequencies for intervals are computed. We assume that the size of each of the *alive* intervals is small enough

[4] Our algorithm does not require intervals to be of equal size.

[5] We also compare the $gini_{min}$ of the numeric attributes with the minimum *gini* value among all the categorical attributes and use the minimum in the next step.

that it can fit into the main memory.[6] Since we are choosing the intervals based on regular sampling of a random subset, the number of records from the whole dataset in each interval are close to $\frac{n}{q}$ with a high probability. Thus each of these intervals will fit into the memory with high probability when q is $O(\frac{n}{M})$, where M is the size of the main memory. For each *alive* interval we sort all the points. Since we know the class frequencies at interval boundaries and the points of a *alive* intervals are sorted, the class frequencies at each point in the *alive* interval can be computed easily. For each distinct point we evaluate the *gini* index and keep track of the minimum value, which will be used to determine the best splitting criterion.

The above operation requires reading the entire dataset corresponding to the internal node and storing only the points that belong to *alive* intervals. If all the *alive* intervals do not fit into the main memory, they are written to the disk. Another pass is required to read these *alive* intervals one at a time. If the *survival* ratio is small, and all the *alive* intervals can fit in the main memory, this extra pass is not required.

Partitioning the Dataset The splitting criterion is applied to each record to determine its partition (left or right). The number of records read and written is equal to the number of records represented by the internal node. For node i at level l of the tree, set S is partitioned, using the splitting criteria, into two sample sets S_1 and S_2. The sample sets S_1 and S_2 are used for left and right subtrees of node i, respectively. While reading the data for splitting a given node the class frequencies for the interval boundaries are updated. Thus frequency vectors corresponding to S1 and S2 are updated, which avoids a separate pass over the entire data to evaluate these frequencies.

Assuming that all the *alive* intervals fit in the main memory, we need to read all the records at most twice and write all the records for each internal node (except the root node). The records read and written correspond only to the internal node(s) being split. An extra read is required on the root to create the sample list.

4.1 I/O and Computational Requirements

We expect that CLOUDS (with the *SSE* method) and the SPRINT algorithm will build the same tree for most datasets, which is demonstrated by the results obtained in a later subsection. Hence

[6] Otherwise, the processing of the *alive* intervals needs to be staged.

our comparisons of the two algorithms are based solely on the I/O and computational requirements in the tree construction phase, assuming that similar trees are built by both algorithms.

The following notation is needed for our comparisons.

1. N — the total number of examples
2. n — the number of examples at a node
3. R — the size of each record
4. r — the size of each record in the attribute list (in SPRINT)
5. F — the number of attributes
6. f — the number of the numeric attributes
7. c — the number of classes
8. q — the number of intervals
9. A — the number of *alive* intervals
10. n_i — the number of elements in *alive* interval i
11. V — the *survival* ratio
12. h — the cost of performing hashing for one record of the attribute list (in SPRINT)

We divide our comparison into two parts, one for the preprocessing stage and the other for the processing required on every node in the tree construction phase. We assume that the record in each attribute list is of the same size (in SPRINT). The cost of performing I/O is assumed to be equal to the total amount of data read or written. This model, though simplistic, is sufficient for our comparison.

The computations required by our algorithm for the root node are slightly different than for the other nodes of the tree. This includes sampling and updating the class frequencies of the boundaries of intervals. Each of these steps requires a pass over the whole dataset. We assume that these steps are part of a preprocessing phase for our algorithm, which makes the processing of root and other nodes relatively uniform for our algorithm.

Preprocessing Phase The pre-processing phase in SPRINT partitions the dataset into attribute lists. This partitioning requires one read operation (of size NR) and one write operation (of size NFr). External sorting of each numeric attribute list requires at least two reads and two writes. The total amount of data read as well as written is $4Nfr$. The computational time for sorting and splitting is $O(fN \log N)$.

Our algorithm requires drawing a random sample that needs at most one read of the whole dataset (of size NR).[7] Updating of the class frequencies of the boundaries of intervals for the root node requires an additional read over the whole dataset (of size NR). The computational cost for the preprocessing phase of our algorithm is $O(s + fN \log q + fs \log q)$ time, where s is the size of the sample chosen and q is the number of intervals, for drawing the sample points, deriving the quantile values (interval boundaries), and updating the class frequencies. The quantile values can be determined in $O(s \log q)$ using an extended median-finding algorithm [1].

We expect the value of Fr to be at least twice as large as R for most datasets, because the transaction ID rid and the class attribute are repeated once for each of the F attribute lists. Hence the I/O requirements of the SPRINT algorithm for the preprocessing phase is larger by a factor of at least $1.5 + 2\frac{fr}{R}$.[8] The computational cost for preprocessing using our algorithm is superior to SPRINT by a factor of $O(\frac{\log N}{\log q})$.

Processing on each node SPRINT requires one read of all the attribute lists (of size nFr) to derive the splitting criterion. Another read and write (of size nFr) on all the attribute lists is required for splitting. One can optimize SPRINT so that the first read (for calculating the splitting criterion) for the node at the next level can be done along with the reading of the data for splitting the current node. This optimization has not been described in [17].

The above description assumes that the hash table along the splitting attribute can be stored in the main memory. For large datasets, splitting may require several stages, which will substantially increase the number of reads and writes of each of the attribute lists.

The computational time required to calculate the $gini$ index is $O(nc)$ for each numeric attribute. The cost of partitioning the dataset is $O(nFh)$.

[7] There are approximate techniques available for random sampling that do not require reading the whole dataset.

[8] It should be noted that SPRINT might perform random I/O, whereas CLOUDS performs a sequential I/O.

Our method requires two reads (each of size nR) and one write (of size nR). If the *alive* intervals do not fit in the main memory, an extra read and write (each of size $2V \times \frac{f}{F+1}nR$) may be required.

The computation time required for each numeric attribute using our method is as follows:

1. $O(s \log q)$ to derive the $q-1$ interval boundaries.
2. $O(n \log q)$ to compute the class frequencies for the interval boundaries,
3. $O(qc)$ to evaluate the *gini* at the interval boundaries
4. $O(qc^2)$ to estimate the lower bound for all the intervals (For the SSE methods),
5. $O(n \log q + \sum_{i=1}^{A}(n_i \log n_i + n_i c))$ to determine all the elements belonging to the *alive* intervals, sort the elements in each of the *alive* intervals, and compute the exact *gini* index (For the SSE methods).[9]

The cost of partitioning the dataset for each node is $O(n)$.

The computational cost of evaluating the splitting criterion for a categorical attribute is the same for both algorithms. Assuming that $\log q$ is a small constant and $q \ll n$, and the number of *alive* intervals is a small fraction of the whole dataset, the computational requirements of our method per element are comparable or better than with the SPRINT algorithm. SPRINT requires hashing for each record of the attribute list, and for numeric attributes it requires that the *gini* index be calculated at all the points. Also, the I/O requirements of the SPRINT algorithm for the processing phase on each node is larger than those of CLOUDS (with the SSE method).

We expect the relative performance to be much better for larger datasets where the hash table used in SPRINT does not fit the main memory; the number of passes over the entire data set in this case will be proportional to the size of the hash table, divided by the amount of available memory.

4.2 Experimental Results

In this section we describe the accuracy, compactness, and efficiency of the CLOUDS classifier and compare (with corresponding results) for SLIQ and/or SPRINT, IND-C4, IND-CART and the DM

[9] If the *width* of an *alive* interval is not large, we could use the direct-address table instead of sorting. The cost of using the direct-address table is $O(n_i + width)$ ($O(n_i \log n_i)$ for sorting).

method. The first two are identical in terms of the accuracy and size of the tree generated, differing only in time requirements for constructing the decision tree. SPRINT is shown to be substantially superior computationally to SLIQ for large datasets, but underperforms for smaller datasets. The results for the first four algorithms have been derived from the literature [15, 17].

Table 9 gives the average accuracy and tree size of the classifiers generated by the DS method. These results show that one can potentially lose a significant amount of accuracy if the whole dataset is not used for building the decision tree. This observation has also been made in [3, 5]. However, we would like to note that this may be a manifestation of the size of the dataset. It is not clear whether this will be the case for larger datasets.

Dataset	5%		20%		40%		**100%**	
	Accuracy	Tree size	Accuracy	Tree size	Accuracy	Tree size	Accuracy	Tree size
Letter	53.6	115.0	69.1	335.0	76.7	592.3	**83.4**	**881**
Satimage	75.9	17.0	81.7	45.7	83.6	69.7	**86.4**	**127**
Shuttle	99.5	9.0	99.9	21.0	99.9	25.7	**99.9**	**27**
Abalone	24.1	15.0	24.0	39.0	24.2	85.7	**26.3**	**137**
Isolet	55.6	57.7	73.1	107.0	78.0	169.0	**82.9**	**261**

Table 9: The average (of three trials) pruned tree size and accuracy for a number of datasets using the DS method

Dataset	Accuracy		Pruned-tree size	
	SS	SSE	SS	SSE
Letter	83.3	83.3	893	893
Satimage	85.9	85.9	135	135
Segment	94.0	94.7	47.0	55.2
Shuttle	99.9	99.9	41	41
Abalone	25.5	26.4	175	147
Waveform-21	76.5	76.8	169.6	172.8
Waveform-40	76.5	76.0	188.8	189.8
Isolet	81.9	82.6	285	255

Table 10: Comparison between the two methods, using 200 intervals at the root. The DM method was applied for nodes with dataset sizes smaller than (roughly) 2.5% of the original dataset

There are two main parameters for the new algorithm (the number of intervals at the root level and the size of the main memory), which may affect the overall accuracy and performance. As soon as the size of the dataset is small enough that it fits in the main memory the DM method is applied to compute the exact *gini* index.

We wanted to study the quality of results obtained by CLOUDS based on the size of the main memory. Our current implementation recursively decomposes the number of intervals into two subsets weighted-assignment strategy. The number of intervals assigned to each of the datasets is proportional to the size of the dataset. Since the size of the datasets available is reasonably small, we achieve the effects of having limited memory by using a stopping criterion based on the number of intervals assigned to a given data subset. The DM method is used when the stopping criterion is met.

We conducted a set of experiments to evaluate the SS and SSE methods based on the overall results. Table 10 shows the overall results of using the two methods for nodes of sizes larger than (roughly) 2.5% of original datasets. The pruned-tree sizes generated by the SS method are comparable to those of the SSE method. However, the SSE method achieved slightly better accuracy for most of the cases. Although, the SSE method generally requires an extra pass to process the alive intervals at each node, it requires at most 30% more time than the SS algorithm (see Table 15). These results show that the overhead of using SSE over SS is not very large. Given that the SSE method is more scalable and less sensitive to the number of intervals, it may be the better option for most datasets. The results presented in the rest of this section assume that SSE method was used.

We conducted experiments assuming 5 and 10 intervals as a stopping criterion. Tables 11, 12, 13 and 14 give the quality of decision trees in terms of accuracy and size (after FULL MDL-based pruning [15, 16, 11]) using CLOUDS for different number of intervals at the root. The corresponding results for CART, C4, SPRINT and the DM method are also presented.[10] These results show that the quality of results obtained is similar or comparable to those of CART, C4, SPRINT and DM method both in terms of accuracy and the size of the decision tree obtained. Further, the accuracy obtained is relatively independent of the number of intervals chosen for the root level and the number of intervals after which the DM method was used. Assuming that we

[10]The DM and SPRINT should generate the same results. However, they generated different trees because (probably) of different implementation choices.

started with 200 intervals and divided the datasets into approximately equal datasets at each node, a stopping criterion based on 10 intervals will correspond to application of the heuristic method for around 4 levels.

Dataset	IND-CART	IND-C4	SPRINT	DM	CLOUDS with SSE	
					$q = 200$	$q = 100$
Letter	84.7	86.8	84.6	83.4	83.1	83.5
Satimage	85.3	85.2	86.3	86.4	85.5	85.2
Segment	94.9	95.9	94.6	94.4	94.5	95.0
Shuttle	99.9	99.9	99.9	99.9	99.9	99.9
Abalone	–	–	–	26.3	26.1	25.4
Waveform-21	–	–	–	77.3	77.0	77.6
Waveform-40	–	–	–	76.4	76.1	77.0
Isolet	–	–	–	82.9	82.2	82.7

Table 11: Accuracy obtained for different datasets. The DM method was applied in CLOUDS for nodes with dataset sizes smaller than (roughly) $\frac{5}{q}$ of the original dataset

Dataset	IND-CART	IND-C4	SPRINT	DM	CLOUDS with SSE	
					$q = 200$	$q = 100$
Letter	84.7	86.8	84.6	83.4	83.3	83.4
Satimage	85.3	85.2	86.3	86.4	85.9	84.9
Segment	94.9	95.9	94.6	94.4	94.7	95.0
Shuttle	99.9	99.9	99.9	99.9	99.9	99.9
Abalone	–	–	–	26.3	26.4	25.9
Waveform-21	–	–	–	77.3	76.8	77.7
Waveform-40	–	–	–	76.4	76.0	77.2
Isolet	–	–	–	82.9	82.6	82.7

Table 12: Accuracy obtained for different datasets. The DM method was applied in CLOUDS for nodes with dataset sizes smaller than (roughly) $\frac{10}{q}$ of the original dataset

Table 15 gives the time requirements of our algorithms for different datasets and a different number of intervals at the root level. These results assume that the *survival* ratios are small enough that the *alive* intervals can be processed in the main memory. The results are presented for CLOUDS using the two estimation methods. These timing were obtained from the IBM RS/6000

Dataset	IND-CART	IND-C4	SPRINT	DM	CLOUDS with SSE	
					$q = 200$	$q = 100$
Letter	1199.5	3241.3	1141	881	895	903
Satimage	90	563	159	127	129	113
Segment	52	102	18.6	41.0	54.6	53.6
Shuttle	27	57	29	27	41	35
Abalone	–	–	–	137	137	147
Waveform-21	–	–	–	168.4	176.2	172.8
Waveform-40	–	–	–	187.4	188.8	189.4
Isolet	–	–	–	261	247	255

Table 13: Pruned tree sizes obtained for different datasets. The DM method was applied in CLOUDS for nodes with dataset sizes smaller than (roughly) $\frac{5}{q}$ of the original dataset

Dataset	IND-CART	IND-C4	SPRINT	DM	CLOUDS with SSE	
					$q = 200$	$q = 100$
Letter	1199.5	3241.3	1141	881	893	881
Satimage	90	563	159	127	135	111
Segment	52	102	18.6	41.0	55.2	51.2
Shuttle	27	57	29	27	41	33
Abalone	–	–	–	137	147	135
Waveform-21	–	–	–	168.4	172.8	172.8
Waveform-40	–	–	–	187.4	189.8	184.6
Isolet	–	–	–	261	255	255

Table 14: Pruned tree sizes obtained for different datasets. The DM method was applied in CLOUDS for nodes with dataset sizes smaller than (roughly) $\frac{10}{q}$ of the original dataset

Model 590 workstation running AIX 4.1.4. It is based on the Power2 architecture with 256KB data cache and a 32KB instruction cache.

This table assumes that the DM method is applied when the size of the data is small enough that less than 10 intervals are assigned to it. CLOUDS with the SSE method requires roughly (at most) 30% more time than the SS method. This extra time can be attributed to the processing of the *alive* intervals in the SSE method.

SPRINT spent around 1,000 and 2,000 seconds in classifying Synth1 and Synth2 datasets, respectively [17]. These timings may not be directly comparable due to somewhat different experimental setups.

Dataset	SSE		SS	
	$q = 200$	$q = 100$	$q = 200$	$q = 100$
Letter	8.76	8.30	8.43	8.26
Satimage	5.40	4.57	5.43	4.60
Segment	1.06	0.9	0.97	0.87
Shuttle	3.98	3.44	3.97	3.41
Abalone	2.89	2.52	2.40	2.25
Waveform-21	4.86	4.48	4.34	4.27
Waveform-40	9.07	8.69	8.53	8.26
Isolet	377.63	331.14	289.13	270.72
Synth1	35.74	34.86	35.43	34.76
Synth2	73.35	72.06	74.09	71.79

Table 15: The time requirements (in seconds) of CLOUDS using the SS and SSE methods for different datasets. The DM method was applied for nodes with dataset sizes smaller than (roughly) $\frac{10}{q}$ of the original dataset

The number of intervals The number of intervals q affects the computational requirements for determining and processing the *alive* intervals. A large value of q will increase the computational time for determining the *alive* intervals, and has a higher probability of reducing the *survival* ratio, which decreases the computational time for processing the *alive* intervals. A small value of q will have the opposite effect. Since CLOUDS starts with some q intervals at the root node and divides these intervals to the child nodes using weighted-assignment strategy, the value of q must be

determined only once. The preprocessing phase of the CLOUDS can be extended to dynamically determine an appropriate value of q for the root node of the tree as follows. Choose a set of values for q (determined based on the memory size and the dataset parameters, e.g., the size, number of classes and the number of attributes), and determine the *survival* ratio for each of them. An appropriate value of q is chosen based on the *survival* ratios. This process does not require any extra I/O; however, it requires extra computational requirements that would be considered small compared to the overall execution time for large datasets which generally produce a large unpruned tree. We are not able to evaluate this method experimentally due to the lack of very large, real datasets.

5 Conclusion

The main goal of developing an effective classifier is to have good generalization properties. The methods described in this work implicitly achieve this by choosing splitters with a minimum *gini* index. There are several options for developing efficient algorithms to perform such data-mining techniques on large datasets. One option is to develop algorithms that reduce computational and I/O requirements, but perform essentially the same operations. Another option is to develop algorithms that reduce computational and I/O requirements by performing an approximation of the actual operation without significant loss (or no loss) of accuracy. The CLOUDS classifier falls in the latter category, while SPRINT falls in the former category. However, CLOUDS has been shown to be as accurate as SPRINT and to have substantially superior computational characteristics. We believe that such approaches may provide attractive alternatives to direct methods for other data-mining applications, especially for cases in which an implicit optimization needs to be performed.

It will be interesting to derive heuristics that will guarantee that the estimate is always lower than the actual value. However, this value should be close enough to the actual minimum for the *survival* ratio to be small. The new algorithms can potentially be extended to other splitting functions, e.g., the towing function and the gain ratio.

Further investigation is needed to study the behavior of the proposed methods on very large real datasets, and to study the effect of the size of the intervals on the overall performance of the proposed methods.

Further investigation is needed to deal with a large *survival* ratio, i.e., all the *alive* intervals do

not fit in the main memory. Three potential techniques for dealing with a large *survival* ratio are:

1. Process a subset of the *alive* intervals to derive the splitter. This subset can be generated using the values of the estimated *gini* for the *alive* intervals. Our preliminary results suggest that this technique is a promising one.

2. Apply the SSE method on each *alive* interval to further reduce the *survival* ratio.

3. Process a (good) subset of the *alive* intervals and apply the SSE method on the remaining ones.

References

[1] K. Alsabti, S. Ranka, and V. Singh. A One-Pass Algorithm for Accurately Estimating Quantiles for Disk-Resident Data. *Proc. of VLDB'97 Conference*, pages 346–355, August 1997.

[2] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth, Belmont, 1984.

[3] J. Catlett. *Megainduction: Machine Learning on Very Large Databases.* Ph.D. Thesis, University of Sydney, 1991.

[4] NASA Ames Research Center. Introduction to IND Version 2.1. *GA23-2475-02 edition*, 1992.

[5] P. K. Chan and S. J. Stolfo. Meta-Learning for Multistrategy and Parallel Learning. *Proc. Second Int'l Workshop on Multistrategy Learning*, pages 150–165, May 1993.

[6] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Morgan Kaufmann, 1989.

[7] S. J. Hong. R-MINI: A Heuristic Algorithm for Generating Minimal Rules from Examples. *the 3rd Pacific Rim Int'l Conf. on Artificial Intelligence*, pages 331–337, August 1994.

[8] M. James. *Classification Algorithms.* Wiley, 1985.

[9] R. Kerber. ChiMerge: Discretization of Numeric Attributes. *Proc. AAAI-92, the 10th Int'l Conf. Artificial Intelligence*, pages 123–128, July 1992.

[10] H. Liu and R. Setiono. Feature Selection via Discretization. *IEEE Transactions on Knowledge and Data Engineering*, 9:4:642–645, July/August 1997.

[11] M. Mehta, J. Rissanen, and R. Agrawal. MDL-Based Decision Tree Pruning. *Int'l Conf. on Knowledge Discovery in Databases and Data Mining*, pages 216–221, August 1995.

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

[13] P. M. Murphy and D. W. Aha. UCI Repository of Machine Learning Databases. *http://www.ics.uci.edu/ mlearn/MLRepository.html, Irvine, CA: University of California, Department of Information and Computer Science*, 1994.

[14] J. Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.

[15] M. Mehta R., Agrawal, and J. Rissanen. SLIQ: A Fast Scalable Classifier for Data Mining. *Proc. of the Fifth Int'l Conference on Extending Database Technology*, pages 18–32, March 1996.

[16] J. Rissanen. Stochastic Complexity in Statistical Inquiry. World Scientific Publication Co., 1989.

[17] J. C. Shafer, R. Agrawal, and M. Mehta. SPRINT: A Scalable Parallel Classifier for Data Mining. *Proc. of the 22th Int'l Conference on Very Large Databases*, pages 544–555, September 1996.

[18] A. Srivastava, V. Singh, and E. Han Vipin Kumar. An Efficient, Scalable, Parallel Classifier for Data Mining. *Technical Report, Department of Computer Science and Engineering, University of Minnesota*, 1996.

[19] S. M. Weiss and C. A. Kulikowski. *Computer Systems that Learn: Classification and Prediction Methods from Statistics, Neural Nets, Machine Learning, and Expert Systems.* Morgan Kaufmann, 1991.