PDT SSE: A Scalable Parallel Decision Tree Algorithm Based on MapReduce

Yan Cui†, Yuanyang Yang*, Shizhong Liao†
*School of Computer Science and Technology
Tianjin University, Tianjin 300072, P. R. China,
†Department of Common Required Courses
Tianjin University of Traditional Chinese Medicine, Tianjin 300193, P. R. China
‡Corresponding author: szliao@tju.edu.cn

Abstract—Parallel decision tree learning is an effective and efficient approach to scaling the decision tree to large data mining application. Aiming at large scale decision tree learning, we present a novel parallel decision tree learning algorithm in MapReduce framework, called PDT SSE (Parallel Decision Tree via Sampling Splitting points with Estimation). We first propose an estimation method for sampling splitting points, which can effectively handle both categorical and numeric attributes over large scale data. We also derive an error bound for the algorithm, and analyze the computational complexities of the algorithm. Finally, we describe the implementation procedures in MapReduce framework. Theoretical analysis and experimental results show that PDT SSE has low computational cost compared to state of the art classifiers while maintaining the quality of the generated trees in terms of accuracy, and can scale to large scale data mining application.

Keywords-Classification; Decision Tree; Sampling; Parallel, MapReduce

I. INTRODUCTION

Decision trees are elementary and effective classifiers. They have been used widely for data mining because of their high efficiency and accuracy, good readability and robustness when compared with other classification methods [1], [2]. Decision tree algorithms retain integral dataset in memory and sort all numerical attributes to decide which is used to split child nodes. Memory-resident and sorting result in high requirements on running time and memory storage. These requirements are two of most significant drawbacks of decision tree algorithms.

Techniques improving these drawbacks include discretization, sampling [3] and parallelization [4]. Catlett [3] proposed a method that samples at each node of the classification tree, but this study didn’t consider datasets that were too large to fit in main memory. Even more unfortunately, discretization and sampling cause significant losses in accuracy and lack of reliable [5]. Chan and Stolfo [6] proposed a method that partitioned the dataset such that each subset fitted in main memory. However, their results showed that the performance of resulting decision tree was decreased. Alsabti [7] presented a decision tree classifier for large datasets called CLOUDS. CLOUDS reduced the loss of accuracy and narrowed the search space of the best split through sampling the splitting points and an estimation step for numeric attributes. Since most of the decision tree algorithms are designed only for memory-resident data, much work has been dedicated to improve them. SLIQ [8], SPRINT [9] and ScalParC [10] performed pre-sorting to increase efficiency when computing the best numeric split for each numeric attribute at the beginning of tree growth. BOAT [11], CLOUDS [7] and SPIES [12] replaced sorting with approximate representations of the training data. Although pre-sorting and approximate techniques resulted in more accurate, they could not accommodate very large data sets. RainForest [2] significantly improved performance over the SPRINT classification algorithm by adapting the size of dataset to the amount of main memory. Parallel decision tree algorithms were studied over these years [4], [13] as well. SPD T [14] was a parallel decision tree algorithm which was scalable for streaming data. Most parallel decision trees were constructed in a breath-first mode [8], [10], [14], [15] because it balanced loads better.

In this work, we propose a novel parallel decision tree algorithm namely PDT SSE based on MapReduce framework to improve these drawbacks. The rest of the paper is structured as follows. Section II gives a detailed description of the PDT SSE and its scalable algorithms implemented based on Hadoop platform. We give the complexity analysis and error bound of PDT SSE in the Section III. In Section IV, we present experimental results from a detailed performance evaluation of PDT SSE algorithm and dwell upon the advantages of PDT SSE over existing methods. Finally, we conclude in Section V.

II. PDT SSE ALGORITHM

In this section, we initially give detailed description and design of the PDT SSE algorithm. Two critical procedures under the framework of MapReduce that handle splitting nodes and growing subtrees are given as well. We also describe the entire tree induction process in the Hadoop environment, which is a successful implementation of MapReduce [16].

A. Algorithm Design

Consider the following problem: given a very large training set $D$ which is much larger than main memory, each instance in $D$ has $d$ dimensions (i.e. $x_i \in \mathbb{R}^d$, $i \in \{1, 2, \ldots, |D|\}$) with label $y_i \in \{1, \ldots, c\}$. Our goal is to construct a decision tree
based on the training set with multiple processors in a parallel distributed environment. In parallel decision tree algorithms, the primary problems remain finding good splitting points and partitioning the training set to generate new nodes. Considering the characteristics of various parallel methods, PDTDSS uses a breath-first strategy with hybrid parallelism to build the decision tree. Moreover, in order to reduce running time and computation cost while achieving high predictive accuracy and scalability, the following improvements are made: First, SSE method is employed to derive the best splitters among all alive intervals for each internal node of the tree. After SSE, domain sizes of the numeric attributes don’t have any impact on the performance of our algorithm. Second, for the reason that the quality of split for a numeric attribute can be computed independently, we evaluate it in parallel environment based on gini index after one scan over the data set. Last, the main data structures used in PDTDSS are the distributed count matrices which contain summary statistics computed from training set.

Before tree induction, same as SPRINT and ScalParC algorithms, PDTDSS sorts the values of numeric attributes only once. The difference is that only numeric attribute lists are split separately in the following step. Every attribute list contains attribute value and class value associated with every record, and then we run a MapReduce on training data and compute approximate equi-depth histograms for every numerical attribute to get all SS split points. For each categorical attribute, there is a count matrix associated with it, for each numeric attribute, a count matrix is associated with every sampling split. Every count matrix contain the frequency of each class in each partition, the value of gini index for each split can be calculated from them. To minimize bookkeeping and communication cost, PDTDSS passes a hash table of entire training dataset.

A controller lies at the kernel of PDTDSS. The Controller manages the entire tree induction process using a set of MapReduce jobs, each of which builds different parts of the tree at the same time. The controller maintains the components as following:

- **ModelFile(M):** In initial state, it’s empty. At any point, the model file contains the complete classification tree constructed so far.
- **MRExpandQueue(MRQ):** This queue contains nodes to be extended to which the input $D$ is greater than a given threshold or too large to fit in memory.
- **InMemoryQueue(IMQ):** This queue contains nodes to be extended to which the input $D$ is less than a given threshold or fits in memory completely.

As tree induction proceeds, the Controller dequeues nodes off MRQ and schedules jobs to find split predicates for the expanding nodes. Each job takes as input a set of nodes $S$, the training set $D^*$ and the current state of the model $M$. Once the split predicates are determined, the nodes in $S$ are expanded, and then the Controller will update model $M$ with $S$ and their split predicates, MRQ and IMQ are updated with new child nodes according to the node size at the end of this step.

The PDTDSS algorithm breaks up the process of constructing a decision tree into a set of MapReduce tasks which can be divided into two different type of MapReduce jobs according to the different stages of tree building.

Dependencies exist between the two different tasks.

- **MR_ExpandNodes** : This job is responsible for collecting summary statistics and computing a set of candidate splits for nodes in MRQ during tree building. After then, it will expand tree in parallel.
- **MR_InMemBuildNodes** : This job completes tree induction for nodes in IMQ with task parallelism.

**B. Controller Design**

The Controller schedules two types of jobs containing ScheduleMR_ExpandNode and ScheduleMR_InMemory to run repeatedly until all the queues are all empty and none of the jobs it schedules remain running.

```
Algorithm 1 CONTROLLER
Input: Training Data $D$, Model $M=\emptyset$, IMQ=\emptyset, MRQ=\emptyset
1: MRQ.append(root)
2: while MRQ not empty do
3:    ExpandNodeSet $S$ = getExpandNodes(MRQ,M)
4:    ScheduleMR_ExpandNode(S)
5: end while
6: while IMQ not empty do
7:    ScheduleMR_InMemory(IMQ,M)
8: end while
9: if MRQ empty and IMQ empty and jobs finished then
10:   Exit
11: end if
```

**C. MR_ExpandNodes: Expanding Nodes In Parallel**

MR_ExpandNodes is the kernel component that allows PDTDSS to train on datasets that are too large to fit in memory. Given a set of nodes $S$ at one level of the tree, the training dataset $D^*$, and the current tree model $M$, this job computes a set of good splits for each node in $S$ and generates new child nodes. It mainly has two important steps in processing: Summary Statistics Process and SubTree Building Process.

1) **Summary Statistics Process:** In the Summary Statistics process, Each mapper maintains two tables $T_{[n,a]}[c]$, $T_{[n,a,s,l]}[c]$ for categorical and numerical attributes respectively. We also denoted them as $T_{\text{categorical}}$, $T_{\text{numerical}}$ for simplification, where $n$, $a$ denotes the node and corresponding split attribute respectively, $s$ is a split point of numeric attribute and $l$ is an indicator variable set to 0 to indicate leaf child, otherwise set to 1 to indicate right child.

In the Map phase, the training dataset $D$ is partitioned across a set of mappers. Each mapper loads the current model $M$ and the input nodes $S$ into memory, and goes through the assigned subset $D^*$ and applies a Map function to each record in $D^*$. Local summary statistics can be collected on subsets of the training data and later aggregated.

The Algorithms executed by each mapper is outlined in Algorithm 2 and Algorithm 3. Given a training record $(x, y)$, a mapper will first determine whether the record current reading is part of the input dataset for a node in $S$ by traversing the current model $M$. Once the node is determined, the next step is to update count matrix $T_{[n,a]}$ or $T_{[n,a,s]}$ associated with the current splitter. After all mappers have processed its
Assign Reduce

of reducers gives the global count matrix corresponding with key and aggregates values for every key. Finally, the output each reducer takes the values associated with the particular reduce communication and I/O cost. In the Reduce phase, function will aggregate the values with the same key to matrix \( T \) as values. Subsequently, a combine of all the output by the mappers. These partial statistical simultaneously to evaluate the gini index at every split point.

The gini index of categorical attributes can be calculated from all sampling split points. All the intervals with \( \text{gini} \geq \text{gini}_{\text{min}} \) are eliminated to derive alive intervals. For numeric attributes, \( gini \geq gini_{\text{min}} \) are eliminate to derive alive intervals. For each alive interval, we begin to scan attribute list \( S \) quality of all possible sampling splits for each node in \( S \) same key, we can get expanding nodes \( S \) as values. Otherwise, the tuple will be of the form \( \{n, a,s\} \) and \( T_{n,a,s} \) as values. Subsequently, a combine function will aggregate the values with the same key to reduce communication and I/O cost. In the Reduce phase, each reducer takes the values associated with the particular key and aggregates values for every key. Finally, the output of reducers gives the global count matrix corresponding with each candidate split. Thus we can evaluate possible splits for nodes \( S \) in the next step.

2) SubTree Building Process: In the SubTree Building process, every mapper works on the output from the summary statistics job, by aggregating the local count matrices with the same key, we can get expanding nodes \( S \) and evaluate the quality of all possible sampling splits for each node in \( S \) at the same time, then expand tree and generate new child nodes concurrently in the same way as serial decision tree algorithm. The gini index of categorical attributes can be calculated from count matrix \( T_{n,a} \). For numeric attributes, \( gini_{\text{min}} \) can be computed from all sampling split points. All the intervals with \( gini_{\text{est}} \geq gini_{\text{min}} \) are eliminate to derive alive intervals. For each alive interval, we begin to scan attribute list \( L \) associated with this attribute from the boundary, and update count matrix simultaneously to evaluate the gini index at every split point.

The algorithm executed on each mapper is outlined in Algorithm 4 and Algorithm 5. Each mapper processes two types of keys. The first is of the form \( \{n,a\} \) with a value list \( V \) of all the output by the mappers. These partial statistical

\[ \text{Outlook} \quad \text{Temp} \quad \text{Humidity} \quad \text{Windy} \]

<table>
<thead>
<tr>
<th>Outlook</th>
<th>Temp</th>
<th>Humidity</th>
<th>Windy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunny</td>
<td>69</td>
<td>70</td>
<td>false</td>
</tr>
<tr>
<td>Rainy</td>
<td>75</td>
<td>80</td>
<td>false</td>
</tr>
<tr>
<td>Overcast</td>
<td>2</td>
<td>0</td>
<td>true</td>
</tr>
<tr>
<td>Sunny</td>
<td>2</td>
<td>3</td>
<td>false</td>
</tr>
<tr>
<td>Rainy</td>
<td>3</td>
<td>2</td>
<td>true</td>
</tr>
</tbody>
</table>

Find the best split

Algorithm 2 MR_EXPANDNODESMAP::MAP

Input: ExpandNodeSet \( S \), TreeModelFile \( M \), Training record \( (x, y) \in D \)
Output: Table \( T_{\{n, a, v\}}[c], T_{\{n, a, s, l\}}[c] \)

1: \( n = \text{TraverseTree} (M, x) \)
2: if \( n \in S \) then
3: \( \text{candidate attributes } A_{\text{candidate}} \leftarrow \text{getCandidateAttr}(M, n) \)
4: for all attribute \( a \in A_{\text{candidate}} \) do
5: \( x_v \leftarrow \text{attribute value on } a \) in \( x \)
6: if \( a \) is categorical then
7: \( \text{countMatrix } T_{\{n,a\}} = \text{getCountMatrix}(n, a) \)
8: \( \text{addToTable}((x, y), T_{\{n,a\}}) \)
9: else
10: for all split point \( s \) of attribute \( a \) do
11: \( \text{addToTable}((x, y), T_{\{n,a,s\}}) \)
12: end for
13: end if
14: end for
15: end if

input records, they output local summary statistics to master node. If the candidate attribute is categorical, the key will be a tuple of the form \( \{n, a\} \) and the corresponding count matrix \( T_{\{n,a\}} \) as values. Otherwise, the tuple will be of the form \( \{n, a, s\} \) and \( T_{\{n,a,s\}} \) as values. Subsequently, a combine function will aggregate the values with the same key to reduce communication and I/O cost. In the Reduce phase, each reducer takes the values associated with the particular key and aggregates values for every key. Finally, the output of reducers gives the global count matrix corresponding with

Algorithm 3 MR_EXPANDNODESMAP::MAP_FINALIZE

Input: Table \( T_{\text{numerical}}, T_{\text{categorical}} \)
Output: countMatrix \( T_{\{n,a\}}, T_{\{n,a,s\}} \) on each node \( n \) with attribute \( a \)

1: for all countMatrix \( T_{\{n,a\}} \) in \( T_{\text{categorical}} \) do
2: Output \((n, a), T_{\{n,a\}}) \)
3: end for
4: for all countMatrix \( T_{\{n,a,s\}} \) in \( T_{\text{numerical}} \) do
5: Output \((n, a, s), T_{\{n,a,s\}}) \)
6: end for

...
matrices are aggregated to get a single count matrix with the values for all local statistics information about node $n$ with categorical attribute $a$. The other type of key that a reducer processes belongs to numerical attributes. The keys corresponding to numeric attribute are of the form $\{n, a, s\}$. Here the set $V$ associated with each key is a set of count matrices consisting of statistics information about SS split point $s$ corresponding with node $n$, attribute $a$. Candidate attributes on each node can be derived from model $M$, for every node being expanded in $S$, if the node’s size falls below a threshold then labeled with majority label. Otherwise, the best splitter will be chosen from candidate attributes.

Algorithm 4 MR_EXPANDTREENODES_MAP::MAP
Input: Key $k$, Value Set $V$, TreeModelFile $M$, Table $T_{\text{categorical}}, T_{\text{numerical}}$

1: if $k == n, a$ then
2:   add $V$ to $T_{\text{categorical}}$
3: else
4:   add $V$ to $T_{\text{numerical}}$
5: end if

Algorithm 5 MR_EXPANDTREENODES_MAP::MAP_FINALIZE
Input: Key $k$, Value Set $V$, TreeModelFile $M$, Table $T_{\text{categorical}}, T_{\text{numerical}}$ and Attribute Lists $L$

1: expandNodeSet $S \leftarrow$ getExpandNodes($T_{\text{categorical}}, T_{\text{numerical}}$)
2: for all expand Node $n$ in $S$ do
3:   $y_{\text{freq}} \leftarrow$ getMajorityLabelOfData($n$, $T_{\{n,a,v\}}[c]$, $T_{\{n,a,s,t\}}[c]$)
4:   $A_{\text{candidate}} = \text{getCandidateAttrs}(n, T_{\{n,a,v\}}[c], T_{\{n,a,s,t\}}[c])$
5:   Node size $|n| \leftarrow \text{SizeOf}(n, T_{\{n,a,v\}}[c], T_{\{n,a,s,t\}}[c])$
6:   if $|n| \leq \text{num\_threshold}$ then
7:     labelNode($n, y_{\text{freq}}$
8: else if identicalLabel($n, T_{\{n,a,v\}}[c], T_{\{n,a,s,t\}}[c]$) or isIdentical($n, T_{\{n,a,v\}}[c], T_{\{n,a,s,t\}}[c]$)
9:   label $y_{\text{id}eqal} \leftarrow$ identicalLabel($n$
10: labelNode($n, y_{\text{id}eqal}$)
11: else
12:   $\text{Split } split_{\text{best}} \leftarrow \text{findBestSplit}(T_{\{n,a,v\}}[c], T_{\{n,a,s,t\}}[c], L)$
13: for all Node $d$ in $n_c$ do
14:   UpdateQueues($d$, $M, d, A_{\text{candidate}}, L$)
15: UpdateTreeModel($M, n$)
16: UpdateNodeCandidateAttrs($n, A_{\text{candidate}}$
17: end for
18: end if
19: end for

Figure 1 gives an overview of PDTsSSE algorithm. In each iteration, a new level of nodes is appended to the tree, that is, the tree’s depth is incremented by 1. There needs one scan of entire dataset, once the data scan is complete, local statistical information is merged and send to the controller, which makes the splitting decision for each expanding node of the tree and build child nodes where needed. If the node is already pure enough, the splitting is stopped and the node is assigned a label. This building procedure is efficient because only the count matrices, which are fixed in their size, are kept in main memory.

D. MR_InMemBuildNodes: Tree Induction In Memory

As tree induction progresses, the node size becomes small enough to fit in memory at lower levels of trees. At such point, rather than continuing tree induction using MR_ExpandNodes task, the Controller completes tree induction in memory using a different MapReduce job namely MR_InMemBuildNodes with task parallelism. This job partitions $D'$ across a set of mappers. The map function then processes every record $(x, y)$ and traverses the tree in $M$ to see whether the record is input to some node $n$, where $n \in S$. If such a node is found, the map function outputs the node $n$ as the key and a tuple of the form $(x, y)$ as the value. The reduce function receives as input a node $n$ (as key) and the set of training records that are input to the node (as values). At last, The reducer loads the training records for $n$ into memory and completes subtree construction rooted at $n$ using in-core algorithm.

III. THEORETICAL ANALYSIS

In this section, we will give the complexity of splitting and training error bound of PDTsSSE.

A. Complexity of Splitting

Every iteration consists of two steps: an updating step performed simultaneously by all the processors and a merging step performed by the master processor. Whenever MR_ExpandNodes job is executed, it needs a scan over dataset and attribute lists once for each numeric attribute during the procedure of summary statistics. After getting the global information about class distribution for all possible SS splits, we can choose a split with minimum gini value denoted as $gini_{\text{min}}$ among all of the SS splits. If the test attribute $a$ is numeric, then go to scan through all the records in the attribute list corresponding with attribute $a$ in the selected sampling split, and meanwhile use hill-climbing method to evaluate the lower bound of each interval to derive survival intervals. The time complexity of this process is $O(c \times q_{\text{alive}})$, which is relevant to the number of label values instead of the size of the interval. Finally, the best splits is obtained using attribute lists $L$ and count matrix relevant to the selected split is updated for each record current read. Thus the computation complexity of the selection of the best split $gini_{\text{best}}$ depends only on the number of classes $c$, dataset size $|D|$ and the number of survival interval $q_{\text{alive}}$. Since $q_{\text{alive}} \ll q$ where $q$ denotes the number of intervals, compared with SLIQ and SPRINT, PDTsSSE significantly decreases the computation cost in selecting the best split. For large datasets that can’t be loaded into memory completely once, the number of candidate attributes in each internal node decreases, so the time complexity of choosing splitter and communication cost decreases. The only memory allocation is for the count matrices. As the number of intervals is constant, operations on count matrices take a constant amount of time. Every processor performs at most $|D|/W$ matrix updates. There are totally $W \times |S| \times (A_c + A_n \times q)$ count matrices at most, where $|S|$ is the number of records that are to be expanded in the current iteration, $W$ is the number of processors and $A_c, A_n$ are the number of categorical attributes and numeric attributes respectively. Assuming that $W, |S|, c, and d$ are all independent of the dataset size $|D|$, thus space and communication complexities is constant.
B. Error Bound of PDTSSE

In this section, we investigate the training error of PDTSSE. Assume that impurity function \( G \), the gap in the impurity function before and after splitting is denoted as \( \Delta \), suppose that \( s \) is the best split point of splitting attribute \( a_i \) for a tree node \( n \), so that node \( n \) is split according to the rule \( x^{(i)} \leq s \). Denote by \( \tau \) the probability that a record reaching \( n \) is directed to its left child node. Denote further by \( p_{n,j}, p_{L,j} \) and \( p_{R,j} \) the probability of label \( j \) in this node \( n \) and its left and right child nodes respectively. Define the function for the change in value of impurity before and after splitting \( \Delta(a, \{p_j\}, \{p_{L,j}\}, \{p_{R,j}\}) = \Delta(n, i, s) \) as

\[
\Delta = G(\{p_j\}) - \tau G(\{p_{L,j}\}) - (1 - \tau) G(\{p_{R,j}\})
\]

\( \Delta \) can be calculated precisely at every candidate split using (1). However, for PDTSSE, it initially divided the values of a numeric attribute into \( B \) roughly equal-size intervals with \( B-1 \) split points \( s_1, s_2, \ldots, s_{B-1} \) where \( s_1 \leq s_2 \leq \cdots \leq s_{B-1} \). Suppose that the best split point is \( s \), and \( s_k \leq s \leq s_{k+1} \). Since the number of points in the interval \([s_k, s] \) is bounded, there is a bound on the degree of change in \( \Delta \) if one node is split at \( s_k \) instead of the best split point \( s \).

For Decision tree \( T \), the training error rate of \( T \) is

\[
e_T = \frac{1}{N} \sum_{\text{leaf } n \text{ in } T} |n|(1 - \max(p_{n,j}))
\]

Assume that the impurity function \( G \) is continuous and satisfies \( G(\{p_j\}) \geq 1 - \max(p_j) \). Thus the inequality implies that \( e_T \leq G_T \) where \( G_T \) is defined as follows

\[
G_T = \frac{1}{N} \sum_{\text{leaf } n \text{ in } T} |n|G(\{p_{n,j}\})
\]

For our analysis, only one new child node is generated in each expanding. As the split of categorical attribute is identical to serial algorithm, here we only consider the numeric attributes. Let \( T_i \) be the tree after the \( i \)-th iteration for the decision tree only one node is expanded per iteration, and \( n_L, n_R \) denote its left and right child nodes respectively. Then

\[
G_{T_{i+1}} - G_{T_i} = \frac{|n|}{N} \Delta(n, i, s)
\]

**Definition 1.** An internal node \( n \) split by a rule \( x^{(i)} \leq s \) is said to perform locally well with respect to a function \( f(\{p_{n,j}\}) \) if it satisfies \( \Delta(n, i, s) \geq f(\{p_{n,j}\}) \). A tree \( T \) is said to perform locally well if every internal node \( n \) in it performs locally well. Finally, a decision tree building algorithm performs locally well if for every training set, the output tree performs locally well.

**Proof:** From (3), we can get that the lower bound of \( \Delta(n, i, s) \) is \( f(\{p_{n,j}\}) \), also the upper bound of \( G_T \) and \( e_T \) is related to \( f(\{p_{n,j}\}) \). Suppose that \( T_i \) is a leaf which for \( \frac{|n|}{N} f(\{p_{n,j}\}) \) can be lower-bounded by a quantity \( h(t, G_{T_{i-1}}) \) which depends only on \( t \) and \( G_{T_{i-1}} \). Then from the recurrence \( G_T \leq G_{T_{i-1}} - h(t, G_{T_{i-1}}) \), we can derive a lower bound on the training error rate of an algorithm that performs locally well. As a simple example \( f(\{p_{n,j}\}) = \alpha G(\{p_{n,j}\}) \), \( \alpha \geq 0 \). By (2), and since the number of leaves in \( T_{i-1} \) is \( t \), there exists a leaf \( n \) in \( T_{i-1} \) for which \( \frac{|n|}{N} G(\{p_{n,j}\}) \geq G_{T_{i-1}} / t \).

Let \( \hat{n} \) be the node which is split in the \( t \)-th iteration, thus \( \frac{|\hat{n}|}{N} f(\{p_{\hat{n},j}\}) \geq \frac{\tau}{t} G_{\hat{T}_{i-1}} \), where \( n, \hat{n} \) represents the best splits for \( \Delta_n, \Delta_{\hat{n}} \) respectively. Then we get

\[
G_{T_{i-1}} - h(t, G_{T_{i-1}}) = \frac{|n|}{N} \Delta_n \geq \frac{|\hat{n}|}{N} \Delta_{\hat{n}} \geq \frac{1}{t} G_{T_{i-1}}
\]

Let \( G_0 \) be an upper bound on \( G_T \), we obtain \( G_T \leq G_0(t - 1)^{-\alpha/2} \) by (4), therefore

\[
e_T \leq G_0(t - 1)^{-\alpha/2}
\]

Kearns and Mansour [17] give a proof that top-down decision tree learning algorithms are boosting algorithms, that is to say, if the functions that label the internal nodes of the decision tree can weakly approximate the unknown target function, the algorithm will amplify this weak advantages to build a tree achieving any desired level of accuracy, so the performance of resulting tree depends on the local performance of internal tree nodes.

**Theorem 1.** Assume that the intervals which PDTSSE operate on are equal-size. Let \( x^{(i)} < s \) is the best split predicate for a leaf \( n \), then \( \forall \delta \geq 0 \), there exists \( B \) that only depends on \( \tau, \{p_j\}, \{p_{L,j}\}, \{p_{R,j}\} \) and \( \delta \), such that the split \( x^{(i)} \leq \hat{s} \) chosen by PDTSSE algorithm with \( B \) equal-size intervals satisfies \( \Delta(n, i, \hat{s}) \geq \Delta(n, i, s) - \delta \).

**Proof:** Assume that \( B \) is fixed, and consider the split \( x^{(i)} < u_k \), \( u_k \leq s \leq u_{k+1} \) and \( \hat{\tau}, \hat{\rho}_{L,j}, \hat{\rho}_{R,j} \) denote the quantities relevant to this split. Let \( p_j \) denote the probability that a training record \( x \) with label \( j \) that satisfies \( u_k \leq x^{(i)} \leq s \). Then \( \hat{\tau} = \tau - \rho_{L,j} - \rho_{R,j} = (\tau \cdot \rho_{L,j} - \rho_{L,j})/\tau, \hat{\rho}_{R,j} = \{(1 - \tau) p_{R,j} + \rho_{L,j})/(1 - \tau) \).

By the continuity of \( \Delta(\tau, \{p_j\}, \{p_{L,j}\}, \{p_{R,j}\}) \), for every \( \delta > 0 \), there exists \( \epsilon \) such that

\[
\Delta(\tau, \{p_j\}, \{p_{L,j}\}, \{p_{R,j}\}) \leq \delta
\]

\( \forall \rho_j \leq \epsilon \). Since \( p_j \leq \frac{1}{\rho_j} \), we can guarantee that \( \rho_j \leq \epsilon \) by setting \( B = 1/\epsilon \). We thus have \( \Delta(n, i, s) \geq \Delta(n, i, \hat{s}) \geq \Delta(n, i, s) - \delta \).
### Table I. Data set summary.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Attributes</th>
<th>#Classes</th>
<th>#Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment</td>
<td>22</td>
<td>2</td>
<td>8124</td>
</tr>
<tr>
<td>Letter</td>
<td>16</td>
<td>26</td>
<td>20000</td>
</tr>
<tr>
<td>Abalone</td>
<td>8</td>
<td>29</td>
<td>4177</td>
</tr>
<tr>
<td>Adult</td>
<td>14</td>
<td>2</td>
<td>48842</td>
</tr>
<tr>
<td>ISOLET</td>
<td>617</td>
<td>26</td>
<td>7797</td>
</tr>
<tr>
<td>Satimage</td>
<td>36</td>
<td>6</td>
<td>6435</td>
</tr>
<tr>
<td>Shuttle</td>
<td>9</td>
<td>7</td>
<td>58000</td>
</tr>
<tr>
<td>Connect</td>
<td>42</td>
<td>3</td>
<td>67557</td>
</tr>
<tr>
<td>Covertype</td>
<td>54</td>
<td>7</td>
<td>581012</td>
</tr>
<tr>
<td>Poker hand</td>
<td>10</td>
<td>10</td>
<td>1000000</td>
</tr>
</tbody>
</table>

### IV. Experimental Evaluation

In this section, we show our experiments and present a detailed performance evaluation of PDTSSE.

#### A. Experimental Setup and Datasets

We run our experiments on ten datasets from the UCI repository, the characteristics of these datasets are shown in Table I.

All of our experiments were performed on a MapReduce equipped cluster that consists of 5 processors, each of which was configured to use 2GB of RAM and 320GB of hard drive space. One as the master node and the others work as slave nodes. To mitigate the effects of varying cluster conditions, all the running times have been averaged over multiple runs.

#### B. Experimental Results

To verify the effectiveness and efficiency of PDTSSE in the parallel distributed setting, in our first experiment, we compare the values of gini index of the best splitters generated by SSE methods to those using direct method (DM), dataset sampling (DS) and sampling the splitting points (SS) for different number of intervals. We have presented the results using 10 and 30 intervals for each numeric attribute. Table II shows the results of exact and estimated gini index using different methods.

These results show that the SS method missed the minimum gini index most of the time, the same holds true for the DS method. The accuracy of the SS method is more sensitive to the number of intervals, the estimated gini may be not good when there are not enough intervals. The SSE method we applied in PDTSSE missed the exact gini index only for a few cases and the estimated gini generated by the SSE method is more accurate than those of the SS method. These results demonstrate that S can achieve a very good estimate of the splitter with minimum gini index.

Our next experiment presents a comparison of time taken in selecting the best split predicate on the Poker Hand dataset by PDTSSE, SPRINT and SLIQ under the condition of different number of processors.

The results of this experiment are shown in Figure 2. It is apparent that SPRINT needs less of time than SLIQ, and PDTSSE outperforms other methods when using the same number of workers, in addition, the time for selecting best splits significantly decreases as the number of workers is increased.

In next experiment, we give a detailed comparison of PDTSSE’s accuracy to C4.5, CART and SPRINT using different number of intervals based on 5-fold cross validation technique. Results in Table III show that the accuracy obtained by PDTSSE is quite similar or comparable to those of C4.5, CART and SPRINT. The greater the number of intervals, the higher accuracy will PDTSSE achieve.

The last set of experiments present the scalability performance of PDTSSE in terms of the parallel running time obtained for various training set sizes. For these experiments, we randomly split the dataset Covertype into 5 roughly equal-size groups, first on a single group, then two groups, and so on up to five groups. These increasingly larger training datasets are named $D_1$, $D_2$, $D_3$, $D_4$ and $D_5$ respectively. In these experiments, the size of training set is kept constant while the number of workers changes from 1 to 4.

Figure 3 shows the runtime scalability of PDTSSE. As expected, training time increases as the amount of training data is increased. The time, which is related to speedup tends,
Table II. Exact and estimated Gini using different methods.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>DM</th>
<th>DS</th>
<th>SS</th>
<th>SSE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10%</td>
<td>30%</td>
<td>q=10</td>
<td>q=30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segment</td>
<td>0.7143</td>
<td>0.7313</td>
<td>0.7219</td>
<td>0.7162</td>
</tr>
<tr>
<td>Letter</td>
<td>0.9403</td>
<td>0.9421</td>
<td>0.9403</td>
<td>0.9418</td>
</tr>
<tr>
<td>Abalone</td>
<td>0.8620</td>
<td>0.8649</td>
<td>0.8625</td>
<td>0.8624</td>
</tr>
<tr>
<td>Adult</td>
<td>0.3228</td>
<td>0.3412</td>
<td>0.3285</td>
<td>0.3234</td>
</tr>
<tr>
<td>ISOLET</td>
<td>0.9263</td>
<td>0.9475</td>
<td>0.9362</td>
<td>0.9417</td>
</tr>
<tr>
<td>Satimage</td>
<td>0.6532</td>
<td>0.6591</td>
<td>0.6546</td>
<td>0.6541</td>
</tr>
<tr>
<td>Shuttle</td>
<td>0.1758</td>
<td>0.1758</td>
<td>0.1758</td>
<td>0.1758</td>
</tr>
</tbody>
</table>

Fig. 3. Parallel running time versus data size and number of workers.

The results fit the theoretical analysis. For large datasets, computation becomes a significant factor of the overall running time, therefore, PDTSSE can be used to classify large-scale datasets.

V. CONCLUSION

In this work, we have presented PDTSSE, a new algorithm for large-scale decision tree learning in MapReduce framework, which has low memory requirement, high efficiency and good scalability. We show that sampling splitting points with estimation is central to PDTSSE. Future works include extending this algorithm to larger datasets and clusterings. We will also utilize PDTSSE in practical applications.

ACKNOWLEDGMENT

This work is supported by the National Natural Science Foundation of China under grant No.61170019 and the Tianjin Key Laboratory of Cognitive Computing and Application.

REFERENCES