Generating CHAID Trees on Large and Distributed Data

Damir Spisic[1], Jing Xu[2], Xue Ying Zhang[2]
[1]IBM SPSS Predictive Analytics, 233 S. Wacker Dr. 11th Fl., Chicago, IL 60606, USA
[2]IBM SPSS Predictive Analytics, 2&3 Floor, Building C, Outsourcing Park Phase I, No. 11 Jinye 1st Rd., High Tech Zone, Xi'an, China

Abstract
CHAID is one of the most established and popular algorithms for building tree models. Target variable can be either continuous or categorical while the predictors must be all categorical. Any continuous predictors are binned before using them in the model. When generating the model, each node can be split into multiple children nodes. The original algorithm is effective for small and medium data sets. However, generating tree models on distributed data containing large number of records and predictors requires a new approach. We present a distributed algorithm for CHAID tree implemented in the MapReduce model of distributed computation. It executes the same number of data passes as the original approach, but the computation in each data pass is fully parallelized taking advantage of all available computational resources. We demonstrate scalability of the distributed algorithm through experiments on a Hadoop cluster.

Key Words: CHAID, distributed data, MapReduce

1. Introduction

CHi-squared Automatic Interaction Detection (CHAID) is a well established and popular method for building tree models. It can be used for prediction and generate easy to interpret rules that constitute a knowledge base for decision support. It has been applied to numerous applications in different domains.

Given a categorical or continuous target variable and a number of categorical attributes or predictors, CHAID constructs a tree model by iteratively partitioning input training data according to categories of the selected predictors. The root node represents the whole data set while each successive node in the tree represents a subset of the input training data records. Each node is split into multiple children nodes by selecting a predictor and merging the predictor's categories into groups that define the new nodes. Record data in the parent node are divided among the children nodes according to the corresponding merged categories of the selected predictor. When splitting a node, CHAID algorithm selects a predictor and merges its categories so that the values of the target variable are differentiated as much as possible among divided record data in its children nodes. The algorithm was first proposed in the seminal paper [1] by Kass.

CHAID trees are traditionally built on training data which is present at a single location. With emergence of massive and distributed data, data mining algorithms that can handle multiple data sources without the need to merge them before analysis, have drawn great attention.

Extension of traditional CHAID to the distributed data sources has to efficiently address the following performance issues:

(1) Passing massive and distributed data sources to obtain contingency table (i.e. cross-table) for each predictor and each tree node.
    Each cross-table for data with categorical target variable contains number of records corresponding to every combination of predictor categories with the target categories. In order to compute cross-tables efficiently, the counting of records must be performed concurrently on multiple data sources.
(2) Merging categories for each predictor to find the best split predictor for each tree node.
    With the cross-table computed for each predictor and tree node combination, predictor categories are compared pairwise in order to merge the categories with similar target variable distribution. The merging process is repeated until a stopping criterion is met. The final merged categories constitute perspective predictor categories for splitting the given node. This computation is time consuming when the number of predictors and tree nodes becomes large. Therefore, parallel calculation is needed in order to improve the algorithm performance.

The initial work on extending traditional CHAID was done by Ouyang et al [2] and [3]. Their extension addresses the performance issue (1) above. Every local site collects local cross-tables and sends them to the central site. The central site forms global cross-tables and performs category merging and node splitting. However, the performance issue (2) above is not addressed. Computation on the central site becomes a performance bottleneck if the number of predictors and/or tree nodes is large. Increasing number of local sites also adversely affects the performance on the central site.

We propose a new algorithm extending traditional CHAID to the distributed data sources. It addresses both performance issues (1) and (2) thereby achievies full parallelism and scalability.

In Section 2 we present a MapReduce implementation of the parallelized CHAID describing an efficient approach for using available computation resources. Sub-sections clarify the role of mappers, reducers and the controller. Also provided is an example and experiment results. In Section 3 we state the conclusions.

2. MapReduce Implementation

For each node in the process of building a tree model and each predictor in the data set, we can aggregate a cross-table for each local data source. Merging of local cross-tables as well as merging of categories for each pair of predictor and tree node can be executed independently from the other predictors and tree nodes. This makes it possible to distribute computations across multiple computing resources.

First, the massive and distributed data is processed by the local computing resources. Then local cross-tables are merged into global ones according to the two-dimensional keys each formed by a predictor and a tree node. These keys correspond to independent computation tasks, which can be accomplished using multiple computing resources. Our approach to generating CHAID trees is very efficient and highly scalable regarding both computational issues encountered in the distributed setting.

Given are distributed data sources as well as an operating system capable of accessing data and running the model building program. An example of such operating environment is Hadoop with the MapReduce interface as described in [4] by Dean et al. We use it as a concrete example as the implementation is similar for comparable systems.

Under the MapReduce framework, jobs using multiple mappers and reducers fit perfectly for the parallel calculations. Given a partial tree model, all the leaf nodes (i.e. terminal nodes) in the last tree level are considered for splitting. Each mapper passes local data and collects local cross-tables as required, while each reducer merges local cross-tables and decides the best splitting points for a subset of predictors and nodes. The controller selects the best splitting predictor by comparing p-values across all predictors for each node, and determines whether the tree stops its growth. Since the tree grows iteratively level by level, a series of MapReduce jobs will be required. The number of jobs corresponds to the depth of the generated tree model.

Figure 1 below represents the process steps in a flowchart. It starts by splitting the root node containing the whole data set.

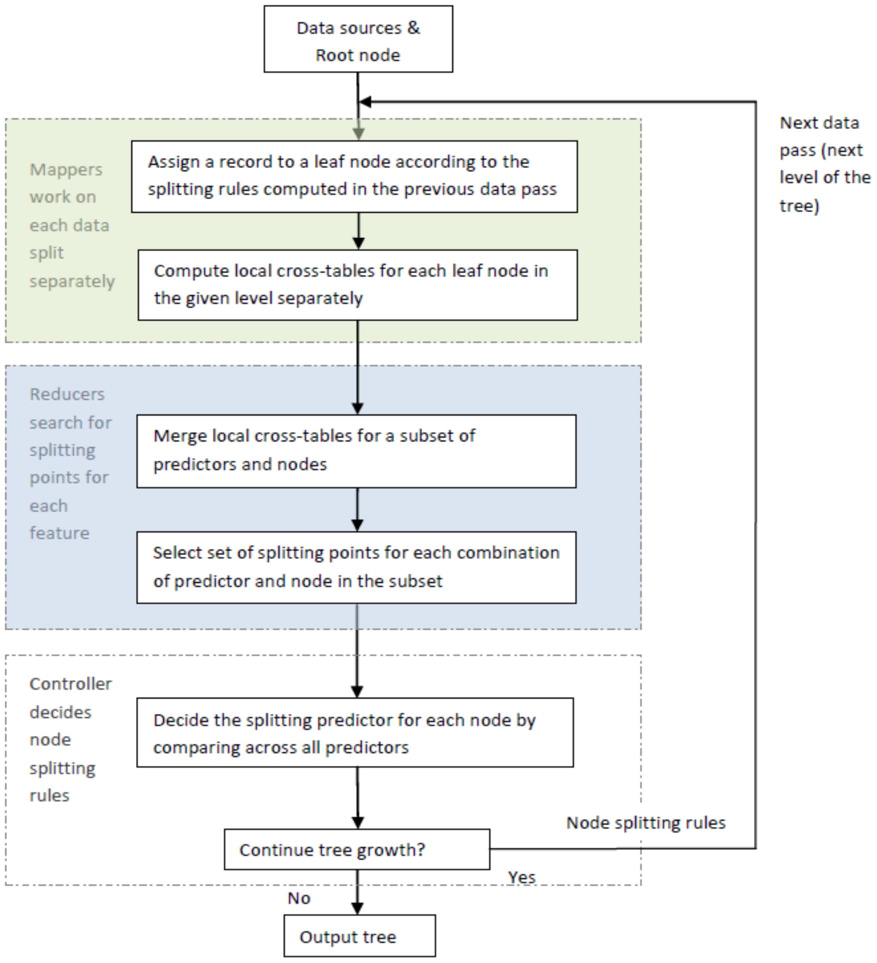
Data sources & Root node
Next data pass (next level of the tree)
Mappers work on each data split separately
Assign a record to a leaf node according to the splitting rules computed in the previous data pass
Compute local cross-tables for each leaf node in the given level separately
Reducers search for splitting points for each feature
Merge local cross-tables for a subset of predictors and nodes
Select set of splitting points for each combination of predictor and node in the subset
Controller decides node splitting rules
Decide the splitting predictor for each node by comparing across all predictors
Node splitting rules
Continue tree growth?
Yes
No
Output tree

Figure 1: Flowchart for scalable CHAID tree growth process.

2.1 Mappers

Under the MapReduce framework, mappers work on each data split (i.e. distributed data source) separately, and each mapper passes local data and performs its tasks as follows:

1. Assign a record to a node according to the splitting rule computed in the previous data pass;
2. Compute cross-tables for each predictor and each node in the given level separately.

The output of each mapper consists of (key, value) pairs where key=key(predictor, node) encodes a predictor and a tree node, while value=value(table) contains the corresponding cross-table data.

2.2 Reducers

Multiple reducers are employed to maximize the parallel calculations, and each reducer is responsible for a subset of keys. For each key, a reducer first merges local cross-tables received from every mapper into the global cross-table. Next, it merges similar predictor categories according to the standard CHAID algorithm, as illustrated in the Figure 2. The resulting set of merged categories constitutes a set of splitting points for the corresponding predictor and the tree node. A p-value corresponding to the chi-square test for the cross-table with the selected predictor splitting points and target categories is also computed.

The output of each reducer is a set of predictor splitting points and the p-value for each combination of predictors and tree nodes determined by the subset of keys processed by the reducer.

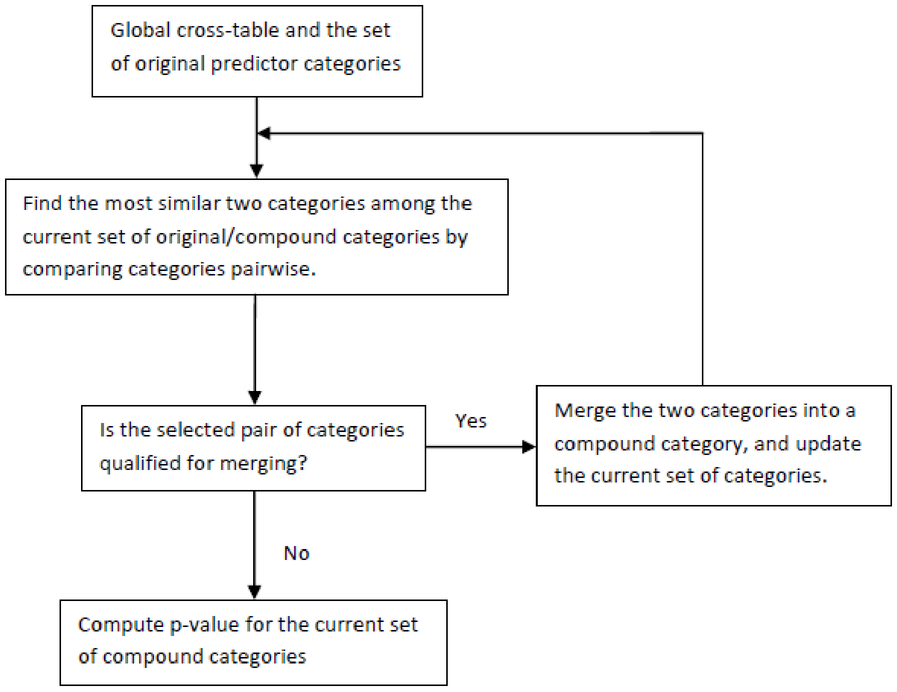
Global cross-table and the set of original predictor categories
Find the most similar two categories among the current set of original/compound categories by comparing categories pairwise.
Is the selected pair of categories qualified for merging?
Yes
Merge the two categories into a compound category, and update the current set of categories.
No
Compute p-value for the current set of compound categories

Figure 2: Flowchart for finding the set of predictor splitting points.

2.3 Controller

The controller collects outputs from the multiple reducers, and selects the best splitting predictor by comparing p-values across all the predictors for each node. Then it checks the tree stopping rules to determine whether to initiate another map-reduce job to grow the next level of the tree.

2.4 Example

Figure 3 illustrates an example of the MapReduce job executed during a single data pass, which corresponds to the growth of one tree level. Without loss of generality, suppose we have 3 predictors (f1, f2, f3), 2 tree nodes (n1, n2), 2 mappers, and 2 reducers. Then the set of keys consists of 6 keys: (f1, n1), (f1, n2), (f2, n1), (f2, n2), (f3, n1) and (f3, n2). Each mapper generates all the keys, while each reducer collects identical subset of keys from every mapper. The keys are divided randomly and evenly across reducers. If subset of keys for reducer 1 is {(f1, n1), (f1, n2), (f2, n1)}, then the subset of keys for reducer 2 is {(f2, n2), (f3, n1), (f3, n2)}.

Additional notations used in Figure 3 are the following:

$T_{ij}^{(k)}$	Local cross-table for the i-th predictor, j-th node, and k-th mapper
M_{ij}	Set of merged categories, i.e. splitting points, for the i-th predictor and j-th node
p_{ij}	p-value corresponding to the splitting points for the i-th predictor and j-th node

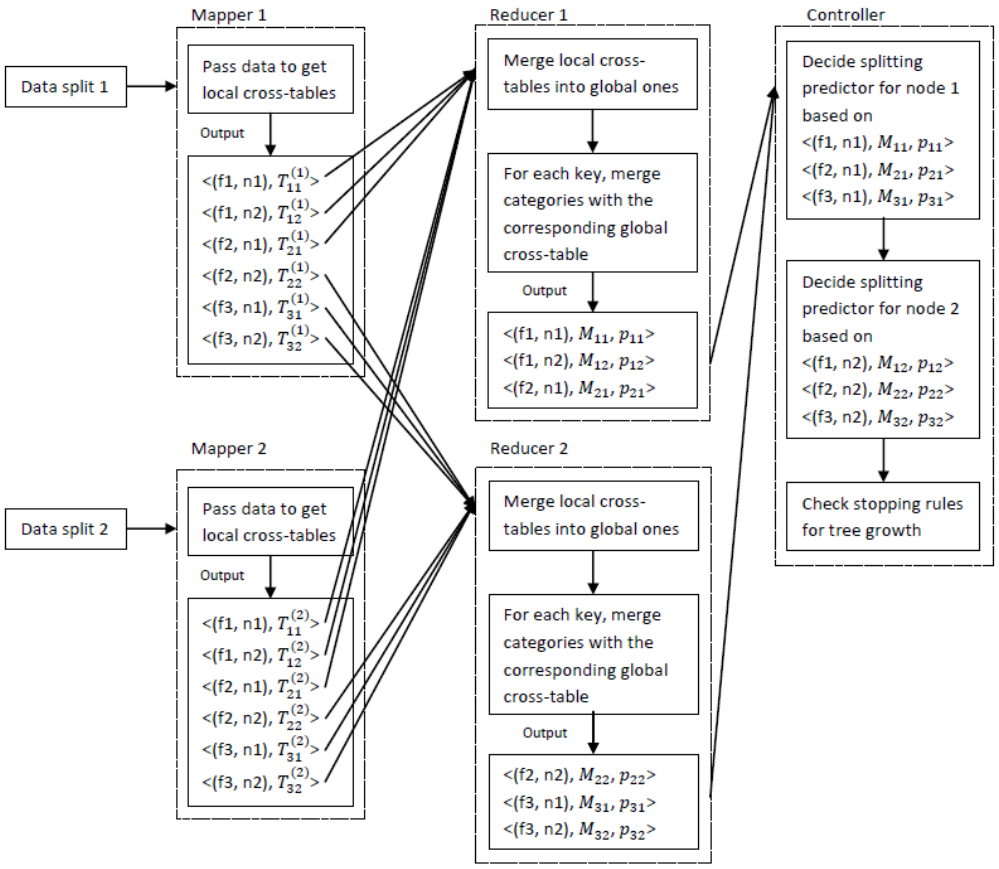
Mapper 1
Data split 1
Pass data to get local cross-tables
Output
<(f1, n1), $T_{11}^{(1)}$>
<(f1, n2), $T_{12}^{(1)}$>
<(f2, n1), $T_{21}^{(1)}$>
<(f2, n2), $T_{22}^{(1)}$>
<(f3, n1), $T_{31}^{(1)}$>
<(f3, n2), $T_{32}^{(1)}$>
Mapper 2
Data split 2
Pass data to get local cross-tables
Output
<(f1, n1), $T_{11}^{(2)}$>
<(f1, n2), $T_{12}^{(2)}$>
<(f2, n1), $T_{21}^{(2)}$>
<(f2, n2), $T_{22}^{(2)}$>
<(f3, n1), $T_{31}^{(2)}$>
<(f3, n2), $T_{32}^{(2)}$>
Reducer 1
Merge local cross-tables into global ones
For each key, merge categories with the corresponding global cross-table
Output
<(f1, n1), M_{11}, p_{11}>
<(f1, n2), M_{12}, p_{12}>
<(f2, n1), M_{21}, p_{21}>
Reducer 2
Merge local cross-tables into global ones
For each key, merge categories with the corresponding global cross-table
Output
<(f2, n2), M_{22}, p_{22}>
<(f3, n1), M_{31}, p_{31}>
<(f3, n2), M_{32}, p_{32}>
Controller
Decide splitting predictor for node 1 based on
<(f1, n1), M_{11}, p_{11}>
<(f2, n1), M_{21}, p_{21}>
<(f3, n1), M_{31}, p_{31}>
Decide splitting predictor for node 2 based on
<(f1, n2), M_{12}, p_{12}>
<(f2, n2), M_{22}, p_{22}>
<(f3, n2), M_{32}, p_{32}>
Check stopping rules for tree growth

Figure 3: MapReduce processing example for a single data pass.

2.5 Experiment Results

For testing purposes we generated several simulated data sets all containing 200 predictors and with different number of records. The total sizes of data sets increased from 64M to 256G in multiples of 4. The intent was to show scalability of the CHAID MapReduce algorithm.

Computing resources used in experiment include a Hadoop cluster with 7 data nodes, each with 4 physical cores and 15G RAM. In addition there is 1 name node and 1 server node, each with 2 physical cores and 7G RAM.

In total there are 32 mappers and 16 reducers available for processing. Given that the size of each data split processed by a single mapper is 64M, the available 32 mappers can process up to 2G of data in parallel.

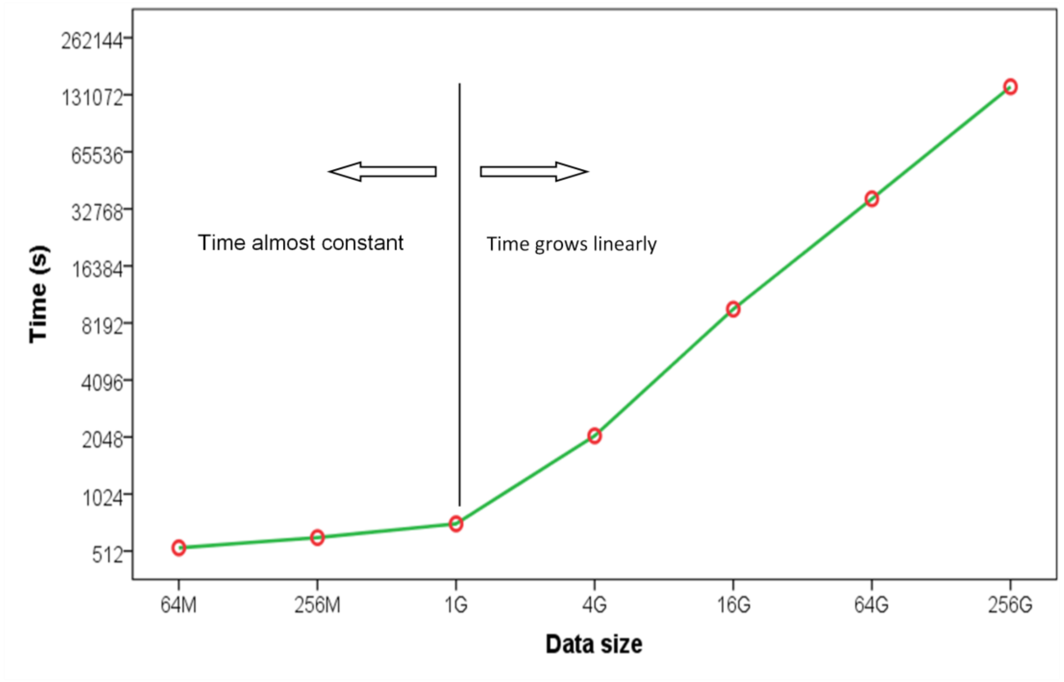

Figure 4: Time vs. Data size for running MapReduce CHAID on a Hadoop cluster. Both axes are in log scale.

Figure 4 shows that the algorithm running time for data sets with size below 1G remains almost constant. This is largely as expected because parallel data processing capacity of the involved mappers is about 2G. In other words, when mappers can process all data in parallel, their processing time is the same as for a single mapper processing its data split of 64M. In addition, there are 16 reducers running in parallel so that the issue due to merging categories and finding the best splits does not adversely affect the running time.

The running time grows linearly for data sets with size above 4G. Sequential processing begins for data sizes above 2G due to mappers being used beyond their parallel processing capacity.

3. Conclusions

CHAID algorithm can be fully parallelized within the MapReduce framework. In addition to processing input data in parallel by mappers, the intermediate results can also be processed in parallel by reducers. This renders the algorithm efficient and scalable. Increasing the size of the computer cluster allows processing of proportionally larger amount of data within almost constant time, while the time grows linearly in the number of records when processing data beyond the system parallel processing capacity.

Presented parallel CHAID algorithm generates tree models explicitly for categorical target variables. Its extension to the continuous target variables is straightforward.

References

[1] Kass, G. V. (1980). An exploratory technique for investigating large quantities of categorical data. *Applied statistics*, 119-127.
[2] Ouyang, J., Patel, N., & Sethi, I. K. (2008). Chi-Square Test Based Decision Trees Induction in Distributed Environment. In *Data Mining Workshops, 2008. ICDMW'08*, pp. 477-485.
[3] Ouyang, J., Patel, N., & Sethi, I. K. (2011). From centralized to distributed decision tree induction using CHAID and fisher's linear discriminant function algorithms. *Intelligent Decision Technologies*, *5*(2), 133-149.
[4] Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107-113.