CHAPTER 2

MapReduce and Its Application to Massively Parallel Learning of Decision Tree Ensembles

Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo

In this chapter we look at leveraging the MapReduce distributed computing framework (Dean and Ghemawat, 2004) for parallelizing machine learning methods of wide interest, with a specific focus on learning ensembles of classification or regression trees. Building a production-ready implementation of a distributed learning algorithm can be a complex task. With the wide and growing availability of MapReduce-capable computing infrastructures, it is natural to ask whether such infrastructures may be of use in parallelizing common data mining tasks such as tree learning. For many data mining applications, MapReduce may offer scalability as well as ease of deployment in a production setting (for reasons explained later).

We initially give an overview of MapReduce and outline its application in a classic clustering algorithm, k-means. Subsequently, we focus on PLANET: a scalable distributed framework for learning tree models over large datasets. PLANET defines tree learning as a series of distributed computations and implements each one using the *MapReduce* model. We show how this framework supports scalable construction of classification and regression trees, as well as ensembles of such models. We discuss the benefits and challenges of using a MapReduce compute cluster for tree learning and demonstrate the scalability of this approach by applying it to a real-world learning task from the domain of computational advertising.

MapReduce is a simple model for distributed computing that abstracts away many of the difficulties in parallelizing data management operations across a cluster of commodity machines. By using MapReduce, one can alleviate, if not eliminate, many complexities such as data partitioning, scheduling tasks across many machines, handling machine failures, and performing inter-machine communication. These properties have motivated many companies to run MapReduce frameworks on their compute clusters for data analysis and other data management tasks. MapReduce has become in some sense an industry standard. For example, there are open-source implementations such as Hadoop that can be run either in-house or on cloud computing services such as Amazon EC2.[1]

[1] http://aws.amazon.com/ec2/.

Startups such as Cloudera[2] offer software and services to simplify Hadoop deployment, and companies including Google, IBM, and Yahoo! have granted several universities access to MapReduce clusters to advance parallel computing research.[3]

Despite the growing popularity of MapReduce, its application to standard data mining and machine learning tasks needs to be better studied. In this chapter we focus on one such task: tree learning. We believe that a tree learner capable of exploiting a MapReduce cluster can effectively address many scalability issues that arise in building tree models on massive datasets. Our choice of focusing on tree models is motivated primarily by their popularity. Tree models are used in many applications because they are interpretable, can model complex interactions, and can easily handle both numerical and categorical features. Recent studies have shown that tree models, when combined with ensemble techniques, provide excellent predictive performance across a wide variety of domains (Caruana et al., 2008; Caruana and Niculescu-Mizil, 2006). The effectiveness of boosted trees has also been separately validated by other researchers; for example, Gao et al. (2009) present an algorithm for model interpolation and ensembles using boosted trees that performs well on *web search ranking*, even when the test data is quite different from the training data.

This chapter describes our experiences with developing and deploying a MapReduce-based tree learner called PLANET, which stands for Parallel Learner for Assembling Numerous Ensemble Trees. The development of PLANET was motivated by a real application in sponsored search advertising, in which massive clickstreams are processed to develop a model that can predict the quality of user experience following the click on a sponsored search ad (Sculley et al., 2009). We show how PLANET effectively scales to large datasets, describe experiments that highlight the performance characteristics of PLANET, and demonstrate the benefits of various optimizations that we implemented within the system. We show that although MapReduce is not a panacea, it still provides a powerful basis on which scalable tree learning can be implemented.

2.1 Preliminaries

Let us first define some notation and terminology that we will use in the rest of the chapter. Let $\mathcal{X} = \{X_1, X_2, \ldots X_N\}$ be a set of features with domains $\mathbb{D}_{X_1}, \mathbb{D}_{X_2}, \ldots \mathbb{D}_{X_N}$ respectively. Let Y be the class label with domain $\mathbb{D}_Y$. Consider a dataset $D = \{(\mathbf{x}_i, y_i) | \mathbf{x}_i \in \mathbb{D}_{X_1} \times \mathbb{D}_{X_2} \times \ldots \mathbb{D}_{X_N}, y_i \in \mathbb{D}_Y\}$ sampled from an unknown distribution, where the ith data vector $\mathbf{x}_i$ has a class label y_i associated with it. Given the dataset D, the goal of supervised learning is to learn a function (or *model*) $F : \mathbb{D}_{X_1} \times \mathbb{D}_{X_2} \times \ldots \mathbb{D}_{X_N} \rightarrow \mathbb{D}_Y$ that minimizes the difference between the predicted and the true values of Y, on unseen data drawn from the same distribution as D. If $\mathbb{D}_Y$ is continuous, the learning problem is a regression problem; if $\mathbb{D}_Y$ is categorical, it is a classification problem. In contrast, in unsupervised learning (e.g., clustering), the goal is to learn a function

[2] www.cloudera.com/.

[3] For example, see www.youtube.com/watch?v=UBrDPRlplyo and www.nsf.gov/news/news_summ.jsp?cntn_id=111470.

$F : \mathbb{D}_{X_1} \times \mathbb{D}_{X_2} \times \ldots \mathbb{D}_{X_N} \times \mathbb{D}_Y$ that best approximates the joint distribution of X and Y in D. For notational simplicity, we will use Y both to denote a class label in supervised methods and to denote a cluster label in clustering.

Let $\mathcal{L}$ be a function that quantifies the disagreement between the value of the function $F(\mathbf{x}_i)$ (predicted label) and the actual class label y_i, for example, the squared difference between the actual label and the predicted label, known as the *squared loss*. A model that minimizes the net loss $\sum_{(\mathbf{x}_i, y_i) \in D} \mathcal{L}(F(\mathbf{x}_i), y_i)$ on the *training set* D may not generalize well when applied to unseen data (Vapnik, 1995). Generalization is attained through controlling model complexity by various methods, e.g., pruning and ensemble learning for tree models (Breiman, 2001). The learned model can be evaluated by measuring its net loss when applied to a holdout dataset.

2.1.1 MapReduce

MapReduce (Dean and Ghemawat, 2004) provides a framework for performing a two-phase distributed computation on large datasets, which in our case is a training dataset D. In the *Map* phase, the system partitions D into a set of disjoint units that are assigned to worker processes, known as *mappers*. Each mapper (in parallel with the others) scans through its assigned data and applies a user-specified map function to each record. The output of the map function is a set of key–value pairs that are collected by the *Shuffle* phase, which groups them by key. The master process redistributes the output of shuffle to a series of worker processes called *reducers*, which perform the *Reduce* phase. Each reducer applies a user-specified reduce function to all the values for a key and outputs the value of the reduce function. The collection of final values from all of the reducers is the final output of MapReduce.

MapReduce Example: Word Histogram

Let us demonstrate how MapReduce works through the following simple example. Given a collection of text documents, we would like to compute the word histogram in this collection, that is, the number of times each word occurs in all the documents. In the Map phase, the total set of documents is partitioned into subsets, each of which is given to an individual mapper. Each mapper goes through the subset of documents assigned to it and outputs a series of $\langle word_i, count_i \rangle$ values as the key–value pair, where $count_i$ is the number of times $word_i$ occurs among the documents seen by the mapper. Each reducer takes the values associated with the a particular key (in this case, a word) and aggregates the values (in this case, word counts) for each key. The output of the reducer phase gives us the counts per word across the entire document collection, which is the desired word histogram.

MapReduce Example: k-means Clustering

MapReduce can be used to efficiently solve supervised and unsupervised learning problems at scale. In the rest of the chapter, we focus on using MapReduce to learn ensembles of decision trees for classification and regression. In this section, we briefly describe how MapReduce can be used for k-means clustering, to show its efficiency in unsupervised learning.

The k-means clustering algorithm (MacQueen, 1967) is a widely used clustering method that applies iterative relocation of points to find a locally optimal partitioning of a dataset. In k-means, the total distance between each data point and a representative point (centroid) of the cluster to which it is assigned is minimized. Each iteration of k-means has two steps. In the cluster assignment step, k-means assigns each point to a cluster such that, of all the current cluster centroids, the point is closest to the centroid of that cluster. In the cluster re-estimation step, k-means re-estimates the new cluster centroids based on the reassignments of points to clusters in the previous step. The cluster re-assignment and centroid re-estimation steps proceed in iterations until a specified convergence criterion is reached, such as when the total distance between points and centroids does not change substantially from one iteration to another.

In the MapReduce implementation of k-means, each mapper in the Map phase is assigned a subset of points. For these points, the mapper does the cluster assignment step – it computes y_i, the index of the closest centroid for each point $\mathbf{x}_i$, and also computes the relevant cluster aggregation statistics: S_j, the sum of all points seen by the mapper assigned to the jth cluster; and n_j, the number of points seen by the mapper assigned to the jth cluster. At the end of the Map phase, the cluster index and the corresponding cluster aggregation statistics (sum and counts) are output. The Map algorithm is shown in Algorithm 1.

Algorithm 1: k-means::Map

```
Input: Training data $\mathbf{x} \in D$, number of clusters $k$, distance measure $d$
1:  If first Map iteration then
2:      Initialize the $k$ cluster centroids $\mathbf{C}$ randomly
3:  Else
4:      Get the $k$ cluster centroids $\mathbf{C}$ from the previous Reduce step
5:  Set $S_j = 0$ and $n_j = 0$ for $j = \{1, \cdots, k\}$
6:  For each $\mathbf{x}_i \in \mathbf{D}$ do
7:      $y_i = \arg\min_j d(\mathbf{x}_i, \mathbf{c}_j)$
8:      $S_{y_i} = S_{y_i} + \mathbf{x}_i$
9:      $n_{y_i} = n_{y_i} + 1$
10: For each $j \in \{1, \cdots, k\}$ do
11:     Output($j$, $< S_j, n_j >$)
```

The reducer does the centroid re-estimation step – it combines the values for a given clusterid key by merging the cluster statistics. For each cluster j, the reducer gets a list of cluster statistics $[< S_j^l, n_j^l >]$, where l is an index over the list – the lth partial sum S_j^l in this list represents the sum of some points in cluster j seen by any particular mapper, whereas the lth number n_j^l in the list is the count of the number of points in that set. The reducer calculates the average of S_j^l to get the updated centroid $\mathbf{c}_j$ for cluster j. The Reduce algorithm is shown in Algorithm 2.

```
Algorithm 2: $k$-means::Reduce
Input: List of centroid statistics – partial sums and counts $[< S_j^l, n_j^l >]$ – for each
    centroid $j \in \{1, \cdots, k\}$
1: For each $j \in \{1, \cdots, k\}$ do
2:     Let $\lambda$ be the length of the list of centroid statistics
3:     $n_j = 0, S_j = 0$
4:     For each $l \in \{1, \cdots, \lambda\}$ do
5:         $n_j = n_j + n_j^l$
6:         $S_j = S_j + S_j^l$
7:     $\mathbf{c}_j = \frac{S_j}{n_j}$
8:     Output($j$, $\mathbf{c}_j$)
```

The whole clustering is run by a Master, which is responsible for running the Map (cluster assignment) and Reduce (centroid re-estimation) steps iteratively until k-means converges.

2.1.2 Tree Models

Classification and regression trees are one of the oldest and most popular data mining models (Duda et al., 2001). Tree models represent F by recursively partitioning the data space $\mathbb{D}_{X_1} \times \mathbb{D}_{X_2} \times \ldots \mathbb{D}_{X_N}$ into non-overlapping regions, with a simple model in each region.

Figure 2.1 shows an example tree model. Non-leaf nodes in the tree define region boundaries in the data space. Each region boundary is represented as a predicate

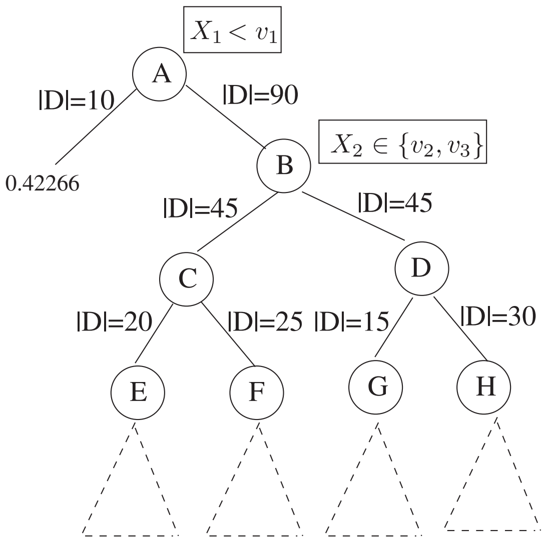

Figure 2.1 Example Tree. Note that the labels on the nodes (in boxes) are the split predicates, whereas the labels on the edges are the sizes of the dataset in each branch (|D| denotes the dataset size in that branch).

on a feature in $\mathcal{X}$. If the feature is numerical, the predicate is of the form $X < v$, $v \in \mathbb{D}_X$ (e.g., Node A in Figure 2.1). Categorical features have predicates of the form $X \in \{v_1, v_2, \ldots v_k\}$, $v_1 \in \mathbb{D}_X, v_2 \in \mathbb{D}_X, \ldots v_k \in \mathbb{D}_X$, (e.g., Node B in Figure 2.1). The path from the root to a leaf node in the tree defines a region. Leaf nodes (e.g., the left child of A in Figure 2.1) contain a region prediction that in most cases is a constant value or some simple function. To make predictions on an unknown **x**, the tree is traversed to find the region containing **x**. The region containing **x** is the path from the root to a leaf in the tree along which all non-leaf predicates are true when evaluated on **x**. The prediction given by this leaf is used as the value for $F(\mathbf{x})$.

In our example tree model, predicate evaluations at non-leaf nodes have only two outcomes, leading to binary splits. Although tree models can have non-binary splits, for the sake of simplicity we focus only on binary splits for the remainder of this chapter. All our techniques also apply to tree algorithms with non-binary splits with straightforward modifications.

Tree models are popular because they are interpretable, capable of modeling complex classification and regression tasks, and handle both numerical and categorical domains. Caruana and Niculescu-Mizil (2006) show that tree models, when combined with ensemble learning methods such as bagging (Breiman, 1996), boosting (Freund and Schapire, 1996), and forests (Breiman, 2001), outperform many other popular learning methods in terms of prediction accuracy. A thorough discussion of tree models and different ensemble methods is beyond the scope of this chapter – see Rokach and Maimon (2008) for a good review.

2.1.3 Learning Tree Models

Previous work on learning tree models is extensive. For a given training dataset D, finding the optimal tree is known to be NP-Hard; thus most algorithms use a greedy top-down approach to construct the tree (Algorithm 3) (Duda et al., 2001). At the root of the tree, the entire training dataset D is examined to find the *best* split predicate for the root. The dataset is then partitioned along the split predicate and the process is repeated recursively on the partitions to build the child nodes.

Algorithm 3: InMemoryBuildNode

```
Input: Node $n$, Data $D$
1: ($n \rightarrow$split,$D_L$,$D_R$)=FindBestSplit($D$)
2: If StoppingCriteria($D_L$) then
3:     $n \rightarrow$left_prediction=FindPrediction($D_L$)
4: Else
5:     InMemoryBuildNode($n \rightarrow$left,$D_L$)
6: If StoppingCriteria($D_R$) then
7:     $n \rightarrow$right_prediction=FindPrediction($D_R$)
8: Else
9:     InMemoryBuildNode($n \rightarrow$right,$D_R$)
```

Finding the best split predicate for a node (Line 3 of Algorithm 3) is the most important step in the greedy learning algorithm and has been the subject of much of the research in tree learning. Numerous techniques have been proposed for finding the right split at a node, depending on the particular learning problem. The main idea is to reduce the *impurity* (I) in a node. Loosely defined, the impurity at a node is a measure of the dissimilarity in the Y values of the training records D that are input to the node. The general strategy is to pick a predicate that maximizes $I(D) - (I(D_L) + I(D_R))$, where D_L and D_R are the datasets obtained after partitioning D on the chosen predicate. At each step, the algorithm greedily partitions the data space to progressively reduce region impurity. The process continues until all Y values in the input dataset D to a node are the same, at which point the algorithm has isolated a pure region (Lines 3 and 7). Some algorithms do not continue splitting until regions are completely pure and instead stop once the number of records in D falls below a predefined threshold.

Popular impurity measures are derived from measures such as entropy, Gini index, and variance (Rokach and Maimon, 2008), to name only a few. PLANET uses an impurity measure based on variance (Var) to evaluate the quality of a split. The higher the variance in the Y values of a node, the greater the node's impurity. Further details on the split criteria are discussed in Section 2.1.4. Although we focus concretely on variance as our split criterion for the remainder of this chapter, as long as a split metric can be computed on subsets of the training data and later aggregated, PLANET can be easily extended to support it.

Scalability Challenge

The greedy tree induction algorithm we have described is simple and works well in practice. However, it does not scale well to large training datasets. FindBestSplit requires a full scan of the node's input data, which can be large at higher levels of the tree. Large inputs that do not fit in main memory become a bottleneck because of the cost of scanning data from secondary storage. Even at lower levels of the tree where a node's input dataset D is typically much smaller than D, loading D into memory still requires reading and writing partitions of D to secondary storage multiple times.

Previous work has looked at the problem of building tree models from datasets that are too large to fit completely in main memory. Some of the known algorithms are disk-based approaches that use clever techniques to optimize the number of reads and writes to secondary storage during tree construction (e.g., Mehta, Agrawal, and Rissanen, 1996). Other algorithms scan the training data in parallel using specialized parallel architectures (e.g., Bradford, Fortes, and Bradford, 1999). We defer a detailed discussion of these approaches and how they compare to PLANET to Section 2.7. As we show in Section 2.7, some of the ideas used in PLANET have been proposed in the past; however, we are not aware of any efforts to build massively parallel tree models on commodity hardware using the MapReduce framework.

Post-pruning learned trees to prevent overfitting is also a well studied problem. However, with ensemble models (Section 2.4), post-pruning is not always needed. Because PLANET is primarily used for building ensemble models, we do not discuss post-pruning in this chapter.

2.1.4 Regression Trees

Regression trees are a special case of tree models where the output feature Y is continuous (Breiman, 2001). We focus primarily on regression trees within this chapter because most of our use cases require predictions on continuous outputs. Note that any regression tree learner also supports binary (0-1) classification tasks by modeling them as instances of logistic regression. The core operations of regression tree learning in Algorithm 3 are implemented as follows.

FindBestSplit(D): In a regression tree, D is split using the predicate that results in the largest reduction in variance. Let $Var(D)$ be the variance of the class label Y measured over all records in D. At each step the tree learning algorithm picks a split that maximizes

$$|D| \times Var(D) - (|D_L| \times Var(D_L) + |D_R| \times Var(D_R)), \tag{2.1}$$

where $D_L \subset D$ and $D_R \subset D$ are the training records in the left and right subtree after splitting D by a predicate.

Regression trees use the following policy to determine the set of predicates whose split quality will be evaluated:

- For numerical domains, split predicates are of the form $X_i < v$, for some $v \in \mathbb{D}_{X_i}$. To find the best split, D is sorted along X_i, and a split point is considered between each adjacent pair of values for X_i in the sorted list.
- For categorical domains, split predicates are of the form $X_i \in \{v_1, v_2, \ldots v_k\}$, where $\{v_1, v_2, \ldots v_k\} \in \mathcal{P}(\mathbb{D}_{X_i})$, the power set of $\mathbb{D}_{X_i}$. Breiman et al. (1984) present an algorithm for finding the best split predicate for a categorical feature without evaluating all possible subsets of $\mathbb{D}_{X_i}$. The algorithm is based on the observation that the optimal split predicate is a subsequence in the list of values for X_i sorted by the average Y value.

StoppingCriteria(D): A node in the tree is not expanded if the number of records in D falls below a threshold. Alternatively, the user can also specify the maximum depth to which a tree should be built.

FindPrediction(D): The prediction at a leaf is simply the average of all the Y values in D.

2.2 Example of PLANET

The PLANET framework breaks up the process of constructing a tree model into a set of MapReduce tasks. Dependencies exist between the different tasks, and PLANET uses clever scheduling methods to efficiently execute and manage them. Before delving into the technical details of the framework, we begin with a detailed example of how tree induction proceeds in PLANET.

The example introduces the different components in PLANET, describes their roles, and provides a high-level overview of the entire system. To keep the example simple, we discuss only the construction of a single tree. The method extends naturally to ensembles of trees, as we discuss in Section 2.4.

Example Setup: Let us assume that we have a training dataset D^* with 100 records. Further assume that tree induction stops once the number of training records at a node falls below 10. Let the tree in Figure 2.1 be the model that will be learned if we run Algorithm 3 on a machine with sufficient memory. Our goal in this example is to demonstrate how PLANET constructs the tree in Figure 2.1 when there is a memory constraint limiting Algorithm 3 to operating on inputs of size 25 records or less.

2.2.1 Components

At the heart of PLANET is the *Controller*, a single machine that initiates, schedules, and controls the entire tree induction process. The Controller has access to a compute cluster on which it schedules MapReduce jobs. In order to control and coordinate tree construction, the Controller maintains the following:

- *ModelFile* (M): The Controller constructs a tree using a set of MapReduce jobs, each of which builds different parts of the tree. At any point, the model file contains the entire tree constructed so far.

Given the ModelFile (M), the Controller determines the nodes at which split predicates can be computed. In the example of Figure 2.1, if M has nodes A and B, then the Controller can compute splits for C and D. This information is stored in two queues.

- *MapReduceQueue* (MRQ): This queue contains nodes for which D is too large to fit in memory (i.e., > 25 in our example).
- *InMemoryQueue* (IMQ): This queue contains nodes for which D fits in memory (i.e., ≤ 25 in our example).

As tree induction proceeds, the Controller dequeues nodes off MRQ and IMQ and schedules MapReduce jobs to find split predicates at the nodes. Once a MapReduce job completes, the Controller updates M with the nodes and their split predicates and then updates MRQ and IMQ with new nodes at which split predicates can be computed. Each MapReduce job takes as input a set of nodes (N), the training data set (D^*), and the current state of the model (M). The Controller schedules two types of MapReduce jobs:

- Nodes in MRQ are processed using MR_ExpandNodes, which for a given set of nodes N computes a candidate set of good split predicates for each node in N.
- Nodes in IMQ are processed using MR_InMemory. Recall that nodes in IMQ have input datasets D that are small enough to fit in memory. Therefore, given a set of nodes N, MR_InMemory completes tree induction at nodes in N using Algorithm 3.

We defer details of the MapReduce jobs to Section 2.3. In the remainder of this section, we tie the foregoing components together and walk through the example.

2.2.2 Walkthrough

When tree induction begins, M, MRQ, and IMQ are all empty. The only node the Controller can expand is the root (A). Finding the split for A requires a scan of the entire training dataset of 100 (≥ 25) records. Because this set is too large to fit in memory, A is pushed onto MRQ and IMQ stays empty.

After initialization, the Controller dequeues A from MRQ and schedules a job MR_ExpandNodes($\{A\}$, M, D^*). This job computes a set of good splits for node A along with some additional information about each split. Specifically, for each split we compute (1) the quality of the split (i.e., the reduction in impurity), (2) the predictions in the left and right branches, and (3) the number of training records in the left and right branches.

The split information computed by MR_ExpandNodes gets sent back to the Controller, which selects the best split for node A. In this example, the best split has 10 records in the left branch and 90 records in the right. The selected split information for node A is then added into the ModelFile. The Controller next updates the queues with new nodes at which split predicates can be computed. The left branch of A has 10 records. This matches the stopping criteria, and hence no new nodes are added for this branch. For the right branch with 90 records (≥ 25), node B can be expanded and is pushed onto MRQ.

Tree induction continues by dequeuing node B and scheduling MR_Expand Nodes($\{B\}$, M, D^*). Note that for expanding node B, we need only the records that went down the right subtree of A, but to minimize bookkeeping, PLANET passes the entire training dataset to the MapReduce. As we describe in Section 2.3.3, MR_ExpandNodes uses the current state of the ModelFile to determine the subset of D^* that will be input to B.

Once the Controller has received the results for the MapReduce on node B and updated M with the split for B, it can now expand both C and D. Both of these nodes get 45 records as input and are therefore pushed on to MRQ. The Controller can now schedule a single MR_ExpandNodes($\{C, D\}$, M, D^*) job to find the best splits for both nodes C and D. Note that by expanding C and D in a single step, PLANET expands trees breadth first as opposed to the depth first process used by the in-memory Algorithm 3.

Once the Controller has the obtained the splits for C and D, it can schedule jobs to expand nodes E, F, G, and H. Of these, H uses 30 records, which still cannot fit in memory and hence get added to MRQ. The input sets to E, F, G are small enough to fit into memory, and hence tree induction at these nodes can be completed in-memory. The Controller pushes these nodes into the IMQ.

The Controller next schedules two MapReduce jobs simultaneously. MR_In Memory($\{E,F,G\}$, M, D^*) completes tree induction at nodes E, F, and G because the input datasets to these nodes are small. MR_ExpandNodes($\{H\}$, M, D^*) computes good splits for H. Once the InMemory job returns, tree induction for the subtrees rooted at E, F, and G is complete. The Controller updates MRQ and IMQ with the children of node H and continues tree induction. PLANET aggressively tries to maximize the number of nodes at which split predicates can be computed in parallel and schedules multiple MapReduce jobs simultaneously.

2.3 Technical Details

In this section, we discuss the technical details of PLANET's major components – the two critical MapReduces that handle splitting nodes and growing subtrees and the Controller that manages the entire tree induction process.

2.3.1 MR_Expand Nodes: Expanding a Single Node

MR_ExpandNodes is the component that allows PLANET to train on datasets too large to fit in memory. Given a set of nodes (N), the training dataset (D^*), and the current model (M), this MapReduce job computes a set of good splits for each node in N.

Map Phase: The training dataset D^* is partitioned across a set of mappers. Each mapper loads into memory the current model (M) and the input nodes N. Note that the union of the input datasets to all nodes in N need not be equal to D^*. However, every MapReduce job scans the entire training dataset applying a Map function to every training record. We discuss this design decision in Section 2.3.3.

Pseudocode describing the algorithms that are executed by each mapper appear in Algorithms 4 and 5. Given a training record $(\mathbf{x}, y)$, a mapper first determines if the

```
Algorithm 4: MR_ExpandNodes::Map
Input: NodeSet $N$, ModelFile M, Training record $(\mathbf{x}, y) \in D^*$
1:  $n$ = TraverseTree(M, $\mathbf{x}$)
2:  If $n \in N$ then
3:      agg_tup$_n \leftarrow (y, y^2, 1)$
4:      For each $X \in \mathcal{X}$ do
5:          $v$ = Value on $X$ in $\mathbf{x}$
6:          If $X$ is numerical then
7:              For each Split point $s$ of $X$ s.t. $s < v$ do
8:                  $T_{n,X}[s] \leftarrow$ agg_tup$_n$
9:          Else
10:             $T_{n,X}[v] \leftarrow$ agg_tup$_n$
```

```
Algorithm 5: MR_ExpandNodes::Map_Finalize
Input: NodeSet $N$
1:  For each $n \in N$ do
2:      Output to all reducers($n$, agg_tup$_n$)
3:      For each $X \in \mathcal{X}$ do
4:          If $X$ is numerical then
5:              For each Split point $s$ of $X$ do
6:                  Output(($n, X, s$), $T_{n,X}[s]$)
7:          Else
8:              For each $v \in T_{n,X}$ do
9:                  Output(($n, X$), ($v, T_{n,X}[v]$))
```

record is part of the input dataset for any node in N by traversing the current model M with $(\mathbf{x}, y)$ (line 1, Algorithm 4). Once the input set to a node is determined, the next step is to evaluate possible splits for the node and select the best one.

Recall from Section 2.1.4 the method for finding the best split for a node n. For a numerical feature X, Equation 2.1 is computed between every adjacent pair of values for the feature that appears in the node's input dataset D. Performing this operation in a distributed setting would require us to sort D^* along each numerical feature and write out the results to secondary storage. These sorted records would then have to be partitioned carefully across mappers, keeping track of the range of values on each mapper. Distributed algorithms implementing such approaches are complex and end up using additional storage or network resources. PLANET makes a trade-off between finding the perfect split for a numerical feature and simple data partitioning. Splits are not evaluated between every pair of values of a feature. Rather, before tree induction we run a MapReduce on D^* and compute approximate equidepth histograms for every numerical feature (Manku, Rajagopalan, and Lindsay, 1999). When computing splits on a numerical feature, a single split point is considered from every histogram bucket of the feature.

On startup, each mapper loads the set of split points to be considered for each numerical feature. For each node $n \in N$ and feature X, the mapper maintains a table $T_{n,X}$ of key–value pairs. Keys for the table are the split points to be considered for X, and the values are tuples (agg_tup) of the form $\{\sum y, \sum y^2, \sum 1\}$. For a particular split point $s \in \mathbb{D}_X$ being considered for node n, the tuple $T_{n,X}[s]$ contains: (1) the sum of Y values for training records $(\mathbf{x}, y)$ that are input to n and have values for X that are less than s, (2) the sum of squares of these values, and (3) the number of training records that are input to n and have values of X less than s. Mappers scan subsets of D^* and compute agg_tups for all split points being considered for each node in N (lines 7, 8 in Algorithm 4). After processing all its input records, each mapper outputs keys of the form n, X, s and the corresponding $T_{n,X}[s]$ as values (line 6, Algorithm 5). Subsequently, a reduce function will aggregate the agg_tups with the same key to compute the quality of the split $X < s$ for node n.

For computing splits on a categorical feature X, Section 2.1.4 proposed computing Equation 2.1 for every subsequence of unique values of X sorted by the average Y. Each mapper performs this computation by maintaining a table $T_{n,X}$ of key, agg_tup pairs as described before. However, in this case keys correspond to unique values of X seen in the input records to node n. $T_{n,X}[v]$ maintains the same aggregate statistics as described earlier for all training records that are input to n and have an X value of v (line 10, Algorithm 4). After processing all input data, the mappers output keys of the form n, X and value $\langle v, T_{n,X}[v]\rangle$ (line 9, Algorithm 5). Note the difference in key–value pairs output for numerical and categorical features. Quality of a split on a numerical feature can be computed independently of other splits on that feature; hence, the split point s is part of the key. To run Breiman's algorithm, all values of a categorical feature need to be sorted by average Y value. Hence, the value v of an feature is not part of the key. A single reducer processes and sorts all the values of the feature to compute the best split on the feature.

In addition to the foregoing outputs, each mapper also maintains agg_tup$_n$ for each node $n \in N$ (line 3, Algorithm 4) and outputs them to all reducers (line 2, Algorithm 5).

Algorithm 6: MR_ExpandNodes::Reduce

```
Input: Key $k$,Value Set $V$
1: If $k == n$ then
2:     // Aggregate agg_tup$_n$'s from mappers by pre-sorting
3:     agg_tup$_n$ = Aggregate($V$)
4: Else If $k == n, X, s$ then
5:     // Split on numerical feature
6:     agg_tup$_{left}$ = Aggregate($V$)
7:     agg_tup$_{right}$ = agg_tup$_n$ - agg_tup$_{left}$
8:     UpdateBestSplit($S[n]$,$X$,$s$,agg_tup$_{left}$, agg_tup$_{right}$)
9: Else If $k == n, X$ then
10:    // Split on categorical feature
11:    For each $v$,agg_tup ∈ V do
12:        $T[v]$ ← agg_tup
13:    UpdateBestSplit($S[n]$,BreimanSplit($X$,$T$,agg_tup$_n$))
```

These tuples are computed over all input records to their respective nodes and help reducers in computing split qualities.

Reduce Phase: The reduce phase, which works on the outputs from the mappers, performs aggregations and computes the quality of each split being considered for nodes in N. Each reducer maintains a table S indexed by nodes. $S[n]$ contains the best split seen by the reducer for node n.

The pseudocode executed on each reducer is outlined in Algorithm 6. A reducer processes three types of keys. The first is of the form n with a value list V of all the agg_tup$_n$ tuples output by the mappers. These agg_tups are aggregated to get a single agg_tup$_n$ with the $\{\sum y, \sum y^2, \sum 1\}$ values for all input records to node n (line 3, Algorithm 6). Reducers process keys in sorted order so that they process all keys of type n first. The other types of keys that a reducer processes belong to numerical and categorical features. The keys corresponding to categorical features are of the form n, X. Here the set V associated with each key is a set of pairs consisting of a categorical feature value v and an agg_tup. For each v, the agg_tups are aggregated to get $\{\sum y, \sum y^2, \sum 1\}$ over all input records to n where the value of X is v. Once aggregated, Breiman's algorithm is used to find the optimal split for X, and $S[n]$ is updated if the resulting split is better than any previous split for n (lines 11–13, Algorithm 6). For numerical features, keys are of the form n, X, s and V is again a list of agg_tups. Aggregating these into agg_tup$_{left}$ gives the $\{\sum y, \sum y^2, \sum 1\}$ values for all records input to n that fall in the left branch of $X < s$ (line 6, Algorithm 6). Using agg_tup$_n$ and agg_tup$_{left}$, it is straightforward to compute the Var based quality of the split $X < s$. If this split $X < s$ is better than the best split seen by the reducer for n so far, then $S[n]$ is updated to the current split (lines 7, 8, Algorithm 6).

Finally, each reducer outputs the best split $S[n]$ that it has seen for each node. In addition to the split quality and predicate, it also outputs the average Y value and number of the training records in the left and right branches of the split. The Controller

Algorithm 7: UpdateQueues

```
Input: DataSetSize $|D|$, Node $n$
1: If not StoppingCriteria($|D|$) then
2:     If $|D| <$ in_memory_threshold then
3:         IMQ.append($n$)
4:     Else
5:         MRQ.append($n$)
```

Algorithm 8: Schedule_MR_ExpandNode

```
Input: NodeSet $N$,Current Model M
1: CandidateGoodSplits = MR_ExpandNodes($N$,M,$D^*$)
2: For each $n \in N$ do
3:     $n \rightarrow$split,$n \rightarrow$l_pred, $n \rightarrow$r_pred,$|D_L|$,$|D_R|$ =
       FindBestSplit($n$, CandidateGoodSplits)
4:     UpdateQueues($|D_L|$,$n \rightarrow$left)
5:     UpdateQueues($|D_R|$,$n \rightarrow$right)
6: $jobs\_running$ $--$
```

takes the splits produced by all the reducers and finds the best split for each node in N, then updates the ModelFile M with this information. The Controller updates the queues with the child nodes that should be expanded using information about the number of training records in each branch.

2.3.2 MR_InMemory: In Memory Tree Induction

As tree induction progresses, the size of the input dataset for many nodes becomes small enough to fit in memory. At any such point, rather than continuing tree induction using MR_ExpandNodes, the Controller completes tree induction in-memory using a different MapReduce job called MR_InMemory. Like MR_ExpandNodes, MR_InMemory partitions D^* across a set of mappers. The map function processes a training record $(\mathbf{x}, y)$ and traverses the tree in M to see if the $(\mathbf{x}, y)$ is input to some node $n \in N$. If such a node is found, then the map function outputs the node n as the key and $(\mathbf{x}, y)$ 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). The reducer loads the training records for n into memory and completes subtree construction at n using Algorithm 3.

2.3.3 Controller Design

The example in Section 2.2 provides the intuition behind functionality of the Controller. Here we provide a more detailed look at its roles and implementation.

The main Controller thread (Algorithm 10) schedules jobs off of its queues until the queues are empty and none of the jobs it schedules remain running. Scheduled

Algorithm 9: Schedule_MR_InMemory

```
Input: NodeSet $N$,Current Model M
1: MR_InMemory($N$,M,$D$)
2: jobs_running −−
```

Algorithm 10: MainControllerThread

```
Input: Model M = ∅, MRQ=∅, IMQ=∅
1: MRQ.append(root)
2: while true do
3:     while MRQ not empty do
4:         If TryReserveClusterResources then
5:             jobs_running ++
6:             NewThread(ScheduleMR_ExpandNode(⊆MRQ,M))
7:     while IMQ not empty do
8:         If TryReserveClusterResources then
9:             jobs_running ++
10:                NewThread(ScheduleMR_InMemory(⊆IMQ,M))
11:    If jobs_running==0 && MRQ empty && IMQ empty then
12:        Exit
```

MapReduce jobs are launched in separate threads so that the Controller can send out multiple jobs in parallel. When an MR_ExpandNodes job returns, the queues are updated with the new nodes that can now be expanded (Algorithm 8). Note that when MR_InMemory finishes running on a set of nodes N (Algorithm 9), no updates are made to the queues because tree induction at nodes in N is complete.

Although the overall architecture of the Controller is fairly straightforward, we would like to highlight a few important design decisions. First, in our example in Section 2.2, recall that the Controller was able to remake the existing nodes from MRQ and IMQ and schedule MapReduce jobs. Therefore, it may seem that the Controller need not maintain queues and can schedule subsequent MapReduce jobs directly after processing the output of a MapReduce job. However, in practice this is not always possible. The memory limitations on a machine and the number of available machines on the cluster often prevent the Controller from scheduling MapReduce jobs for all nodes on a queue at once.

Second, when scheduling MapReduce jobs for a set of nodes, recall that the Controller does not determine the set of input records required by the nodes. Instead, it simply sends the entire training dataset D^* to every job. If the input to the set of nodes being expanded by a node is much smaller than D^*, then this implementation results in the Controller sending much unnecessary input for processing. On the other hand, this design keeps the overall system simple. In order to avoid sending unnecessary input, the Controller would need to write out the input training records for each node to storage. This in turn would require additional bookkeeping for the Controller when operating normally and would further complicate important systems such as the checkpointing mechanism (Section 2.5.3) and ensemble creation (Section 2.4). The amount of unnecessary information sent by our implementation is also mitigated by breadth-first tree construction. If we can expand all nodes at level $i + 1$ in one MapReduce job, then every training record is part of the input to some node that is being expanded. Finally, MapReduce frameworks are already optimized for scanning data efficiently in a distributed fashion – the additional cost of reading in a larger dataset can be mitigated by adding more mappers, if necessary.

2.4 Learning Ensembles

Until now we have described how the PLANET framework builds a single tree. Ensemble-based tree models have better predictive power when compared to single tree models (Caruana et al., 2008; Caruana and Niculescu-Mizil, 2006). Bagging (Breiman, 1996) and boosting (Friedman, 2001) are the two most popular tree ensemble learning methods. In this section we show how PLANET supports the construction of tree ensembles through these two techniques.

Boosting is an ensemble learning technique that uses a weighted combination of weak learners to form a highly accurate predictive model (Freund and Schapire, 1996). Our current boosting implementation uses the GBM algorithm proposed by Friedman (2001). In the GBM algorithm, every weak learner is a shallow tree (depth $\approx$ 2 or 3). Model construction proceeds as follows: Assume $k - 1$ weak learners (shallow trees) have been added to the model. Let F_{k-1} be the boosted model composed of those trees. Tree k is trained on a sample of D^* and residual predictions (z). For a given training record ($\mathbf{x}, y$), the residual prediction for tree k is $z = y - F_{k-1}(\mathbf{x})$ for a regression problem and $z = y - \frac{1}{1+exp(-F_{k-1}(\mathbf{x}))}$ for a classification problem. The boosting process is initialized by setting F_0 as some aggregate defined over the Y values in the training dataset. Abstracting out the details, we need three main features in our framework to build boosted models:

- Building multiple trees: Extending the Controller to build multiple trees is straightforward. Because the Controller manages tree induction by reducing the process to repeated node expansion, the only change necessary for constructing a boosted model is to push the root node for tree k onto the MR after tree $k - 1$ is completed.
- Residual computation: Training trees on residuals is simple because the current model is sent to every MapReduce job in full. If the mapper decides to use a training record as input to a node, it can compute the current model's prediction and hence the residual.
- Sampling: Each tree is built on a sample of D^*. Mappers compute a hash of a training record's ID and the tree ID. Records hashing into a particular range are used for constructing the tree. This hash-based sampling guarantees that the same sample will be used for all nodes in a tree, but different samples of D^* will be used for different trees.

Building an ensemble model using bagging involves learning multiple trees over independent samples of the training data. Predictions from each tree in the model are computed and averaged to compute the final model prediction. PLANET supports bagging as follows: When tree induction begins at the root, nodes of all trees in the bagged model are pushed onto the MRQ. The Controller then continues tree induction over dataset samples as already described. In this scenario, at any point in time the queues will contain nodes belonging to many different trees instead of a single tree, thereby allowing the Controller to exploit greater parallelism.

2.5 Engineering Issues

In developing a production-capable deployment of PLANET, we encountered several unanticipated challenges. First, because MapReduce was not intended to be used for highly iterative procedures such as tree learning, we found that MapReduce startup and tear-down down costs were primary performance bottlenecks. Second, the cost of traversing models in order to determine split points in parallel turned out to be higher than we expected. Finally, even though MapReduce offers graceful handling of failures within a specific MapReduce computation, and because our computation spans multiple MapReduce phases, dealing with shared and unreliable commodity resources remained an issue that we had to address. We discuss our solutions to each of these issues within this section.

2.5.1 Forward Scheduling

Immediately after our initial attempt at deploying PLANET on a live MapReduce cluster, we noticed that an inordinate amount of time was spent in setting up and tearing down MapReduce jobs. Fixing latency due to tear-down time was a simple change to the logic in Algorithms 8 and 10. Instead of waiting for a MapReduce job to finish running on the cluster, the Controller ran a thread that would periodically check for the MapReduce's output files. Once the output files were available, the thread would load them and run the FindBestSplit and UpdateQueues logic described in Algorithm 8.

Addressing the latency caused by job set up was a more interesting challenge. Setup costs include time spent allocating machines for the job, launching a master to monitor the MapReduce job, and preparing and partitioning the input data for the MapReduce. To get around this problem, we implemented a simple trick of forward scheduling MapReduce jobs. Figure 2.2 illustrates the basic idea. Suppose the Controller has to run two MapReduce jobs to expand level i and $i+1$ in the tree. According to our discussion, until now it would schedule Job-1 first and then Job-2 (upper part of

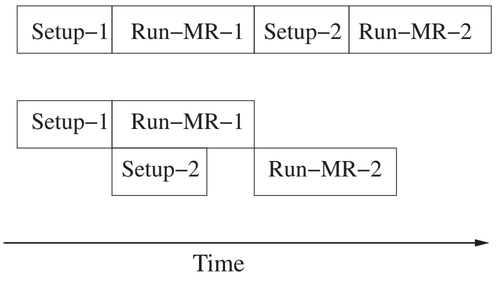

Figure 2.2 Forward scheduling.

Figure 2.2). However, to eliminate the latency due to Setup-2, the Controller sets up Job-2 while Job-1 is still running (lower part of Figure 2.2).

To implement forward scheduling, the Controller runs a background thread that continuously keeps setting up one or more MapReduce jobs on the cluster. Once the jobs are set up, the mappers for the job wait on the Controller to send them a model file and the set of nodes to expand. When the Controller finds work on MRQ or IMQ, it sends the work information out to the waiting mappers for a job using a remote procedure call (RPC). With forward scheduling, lines 6 and 10 of Algorithm 10 now make RPCs rather than spawning off new threads, and the previous lines try to reserve one of the spawned MapReduces.

In practice, the Controller can forward-schedule multiple jobs at the same time depending on the number of MapReduce jobs it expects to be running in parallel. A possible downside of forward scheduling is that the forward scheduling of too many jobs can result in wasted resources, where machines are waiting to receive task specifications, or in some cases receive no tasks because tree induction may be complete. Depending on availability in the cluster and the expected tree depth and ensemble type, we tune the amount of forward scheduling in the Controller.

2.5.2 Fingerprinting

Another significant source of latency that we observed in our MapReduce jobs was the cost of traversing the model: an operation performed on every mapper to determine if the training record being processed is part of the input to any node being expanded in the job. After careful examination and profiling, we found that predicate evaluations at nodes that split on categorical features were a bottleneck because a single predicate evaluation required multiple string comparisons, and some of our features were long strings, e.g., URLs. To get around this, for a predicate of the form $X \in \{v_1, v_2, \ldots v_k\}$, we fingerprint the v_i's and store a hash set at the node. This simple optimization provided about 40% improvement in tree traversal costs.

2.5.3 Reliability

Deploying PLANET on a cluster of commodity machines presents a number of challenges not normally posed when running an application on a single machine. Because our clusters are shared resources, job failures due to preemption by other users are not uncommon. Similarly, job failures because of hardware issues occur occasionally. Because of the frequency of job failures, we require PLANET to have a mechanism for recovering from failures. Fortunately, the MapReduce framework provides us guarantees in terms of job completion. Therefore, we can reason about the system by considering the expansion of a set of nodes as an atomic operation, and when a single MapReduce fails, the Controller will simply restart the MapReduce again.

To handle the failure of the Controller, we annotate the model file with metadata marking the completion of each splitting task. Then, when the Controller fails, we start a new Controller that reads in the annotated model file generated during the failed run. Given the annotated model file, it is simple for the Controller to reconstruct the state of MRQ and IMQ prior to any jobs that were running when the Controller failed. With MRQ, IMQ, and M, the Controller can then continue with tree induction.

Monitoring turned out to be another issue in deploying PLANET. As developers and users of the system, we often needed to be able to monitor the progress of model construction in real time. To support such monitoring, we added a dashboard to PLANET to track its currently running tasks as well as the pending tasks in MRQ and IMQ. The dashboard collects training and validation error statistics and renders a plot of the error of the model as it grows (and offers a precision-recall curve when training a model for classification).

2.6 Experiments

In this section we demonstrate the performance of PLANET on a real-world learning task in computational advertising. In particular, we study the scalability of the system and the benefits obtained from the different extensions and optimizations proposed in the chapter.

2.6.1 Setup

We measure the performance of PLANET on the *bounce rate prediction problem* (Kaushik, 2007a,b). A click on a sponsored search advertisement is called a *bounce* if the click is immediately followed by the user returning to the search engine. Ads with high bounce rates are indicative of poor user experience and provide a strong signal of advertisement quality.

The training dataset (AdCorpus) for predicting bounce rates is derived from all clicks on search ads from the Google search engine in a particular time period. Each record represents a click labeled with whether it was a bounce. A wide variety of features are considered for each click. These include the search query for the click, advertiser chosen keyword, advertisement text, estimated clickthrough rate of the ad clicked, a numeric similarity score between the ad and the landing page, and whether the advertiser keyword precisely matched the query. To improve generalization, we assigned the query and advertiser keywords into one of approximately 500 clusters, and used cluster properties as additional features. Overall, the dataset consisted of six categorical features varying in cardinality from 2 to 500, four numeric features, and 314 million records.

All of our experiments were performed on a MapReduce equipped cluster where each machine was configured to use 768MB of RAM and 1GB of hard drive space (peak utilization was $<$ 200MB RAM and 50MB disk). Unless otherwise noted, each MapReduce job used 200 machines. A single MapReduce was never assigned more than four nodes for splitting, and at any time a maximum of three MapReduce jobs were scheduled on the cluster. Running time was measured as the total time between the cluster receiving a request to run PLANET and PLANET exiting with the learned model as output. In each experiment, the first run was ignored because of the additional one-time latency to stage PLANET on the cluster. To mitigate the effects of varying cluster conditions, all the running times have been averaged over multiple runs.

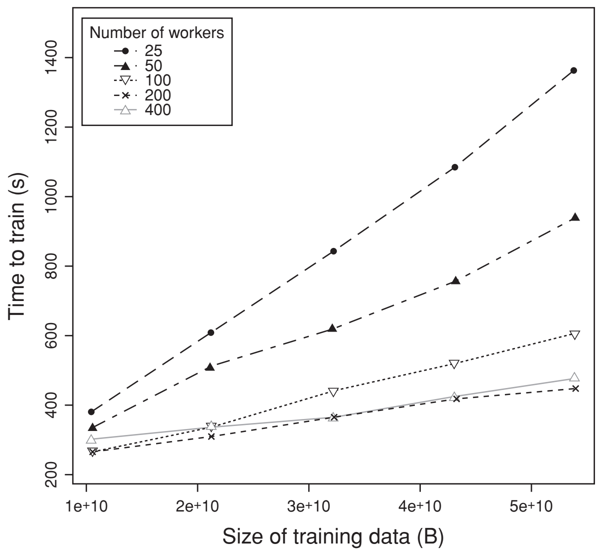

Figure 2.3 Running time versus data size.

To put the timing numbers that follow into perspective, we also recorded the time taken to train tree models in R using the GBM package (Ridgeway, 2006). This package requires the entire training data in memory, and hence we train on a sample of 10 million records (about 2GB). On a machine with 8GB RAM and sufficient disk, we trained 10 trees, each at depth between 1 and 10. Peak RAM utilization was 6GB (average was close to 5GB). The runtime for producing the different trees varied between 315 and 358 seconds (Figure 2.4).

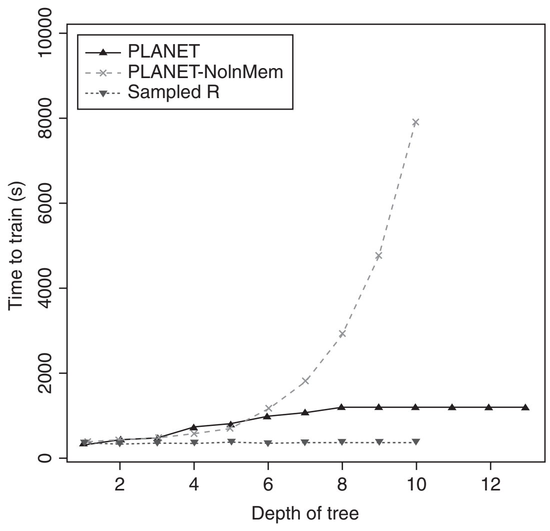

Figure 2.4 Running time versus tree depth. Note: The Sampled R curve was trained on 1/30 of the data used for the other curves.

2.6.2 Results

Scalability: Our first experiment measures the scalability of the PLANET framework. For this experiment, we randomly split the AdCorpus into 5 roughly equal-sized groups and trained a single depth-3 classification tree, first on a single group, then two groups, and so on up to five groups. For each of these increasingly larger training datasets, we examined the effects of using between 50 and 600 machines. In this experiment, the Controller never scheduled more than two MapReduce jobs at a time and was configured to schedule MR_ExpandNodes jobs only. In other words, we disabled the optimization to construct trees entirely in memory and limited forward scheduling to one job in order to evaluate the performance of the algorithm in a constrained (e.g., shared cluster) environment.

Figure 2.3 shows the results of this experiment. As expected, training time increases in proportion to the amount of training data. Similarly, adding more machines significantly decreases training time (ignoring the 400 machine curve for the moment). The most interesting observation in Figure 2.3 is the notion of marginal returns. When the dataset is large, adding more machine reduces costs proportionally, up to a point. For example, in our experiment, increasing the number of machines from 200 to 400 per MapReduce did not improve training time. Similarly, as the training set size decreases, the benefits of adding more machines also diminishes. In both these cases, after a certain point the overhead of adding new machines (networking overhead to watch the worker for failure, to schedule backup workers, to distribute data to the worker, and to collect results from the worker) dominate the benefits from each machine processing a smaller chunk of data. Empirically, it appears that for our dataset the optimal number of workers is under 400.

Benefits of MR_InMemory: Our next experiment highlights the benefits from in memory tree completion. Here, the Controller was configured to invoke MR_InMemory for nodes whose inputs contained 10M or fewer records. The reducers in MR_InMemory used the GBM package for tree construction and were configured with 8GB RAM in order to meet the memory requirements of the package. PLANET was used to train a single classification tree of varying depths on the entire AdCorpus.

Figure 2.4 shows the results. PLANET-NoInMem plots the training time when MR_InMemory is not used by the Controller. In this case training time keeps increasing with tree depth as the number of MR_ExpandNodes jobs keeps increasing. Note that even though we expand trees breadth first, the increase in training time is not linear in the depth. This happens because each MR_ExpandNodes job is configured (based on memory constraints in the mappers) to expand four nodes only. At lower levels of the tree, a single MapReduce can no longer expand all nodes in a level, and hence we see a superlinear increase in training time. On the other hand, PLANET using a mix of MR_ExpandNodes and MR_InMemory scales well and training time does not increase as significantly with tree depth.

As a reference point for the PLANET running times, we also provide the running time of Sampled-R in Figure 2.4, which shows the running time of the GBM in-memory algorithm on a 2GB sample of AdCorpus.

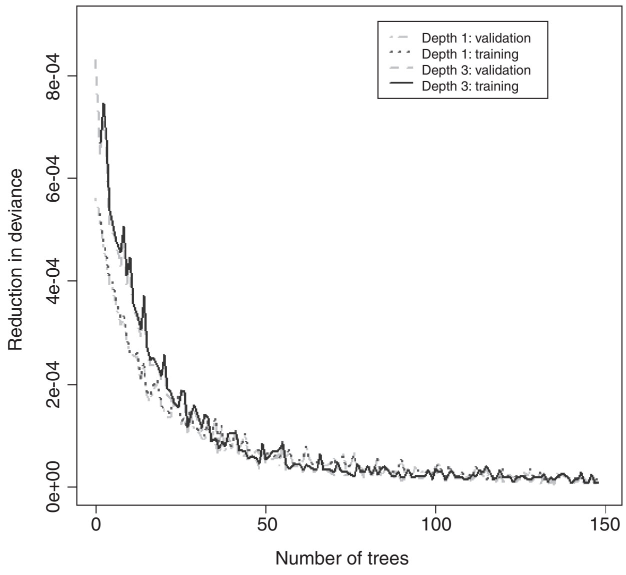

Figure 2.5 Error reduction as the number of trees increases.

Effect of Ensembles: The last experiment we report shows how error rates decrease in the bounce rate problem. Figure 2.5 shows the reduction in training and validation errors on a 90–10 split of the AdCorpus. The figure plots the reduction in deviance as more trees are added to a boosted tree model. Two scenarios are shown – one in which the weak learners are depth one trees, and the other where the trees have depth three. For the depth-3 tree ensemble, the reduction in error is initially higher than with the depth-1 tree ensemble, as expected; however, the reduction asymptotes after about 100 trees for this dataset. The PLANET dashboard updates and displays such error graphs in real time. This enables users to manually intervene and stop model training when the error converges or overfitting begins.

2.7 Related Work

Scaling up tree learning algorithms to large datasets is an area of active research interest. There have been two main research directions taken by previous work: (1) centralized algorithms for large datasets on disk to avoid in-memory limitations and (2) parallel algorithms on specific parallel computing architectures. In applying the MapReduce framework to large-scale tree learning, PLANET borrows and builds on several ideas from these previous approaches.

Centralized Algorithms: Notable centralized algorithms for scaling decision tree learning to large datasets include SLIQ (Mehta et al., 1996), CLOUDS (Alsabti, Ranka, and Singh, 1998), RAINFOREST (Gehrke, Ramakrishnan, and Ganti, 1998), and BOAT (Gehrke et al., 1999). SLIQ uses strategies such as pre-sorting and feature lists in breadth-first tree-growing to enable learning from large training data on disk.

Although PLANET does not use pre-sorting or feature lists, it grows the tree breadth-first like SLIQ. The key insight in RAINFOREST is that the splitting decision at a tree node needs a compact data structure of sufficient statistics (referred to as AVC group in the RAINFOREST algorithm), which in most cases can be fit in-memory. PLANET similarly maintains sufficient statistics on mappers during MR_ExpandNodes. CLOUDS samples the split points for numeric features and uses an estimation step to find the best split point, resulting in lower computation and I/O cost compared to other tree learning algorithms such as C4.5. For efficient estimation of the best split, PLANET uses equidepth histograms of numerical features to estimate split points. Finally, BOAT uses statistical sampling to construct a tree based on a small subset of the whole data and then does corrections to the tree based on estimated differences compared to the actual tree learned on the whole data. In contrast, PLANET builds the tree from the whole data directly.

Parallel Algorithms: Numerous approaches for parallelizing tree learning have been proposed. Provost and Fayyad (1999) give an excellent survey of existing approaches, along with the motivations for large-scale tree learning. Bradford et al. (1999) discuss how the C4.5 decision tree induction algorithm can be effectively parallelized in the ccNUMA parallel computing platform. It also mentions other parallel implementations of decision trees, namely SLIQ, SPRINT, and ScalParC for message-passing systems, and SUBTREE, MWK, and MLC++ for symmetric multiprocessors (SMPs). Most of these algorithms have been developed for specific parallel computing architectures, many of which have specific advantages, such as shared memory to avoid replicating or communicating the whole dataset among the processors. In comparison, PLANET is based on the MapReduce platform that uses commodity hardware for massive-scale parallel computing.

For deciding the split points of features, SPRINT (Shafer, Agrawal, and Mehta, 1996) uses feature lists like SLIQ. Each processor is given a sublist of each feature list, corresponding to the instance indices in the data chunk sent to the processor. While computing good split points, each processor determines the gains over the instances assigned to that processor for each numerical feature and sends the master a portion of the statistics needed to determine the best split. However, this requires an all-to-all broadcast of instance IDs at the end. PLANET takes a simpler and more scalable approach – instead of considering all possible split points, it computes a representative subset of the splits using approximate histograms, after which the selection of the best split can be done using only one MapReduce job (details in Section 2.3.1).

ScalParC (Joshi, Karypis, and Kumar, 1998), which builds on SLIQ and SPRINT, also splits each feature list into multiple parts and assigns each part to a processor. However, rather than building the tree in a depth-first manner (as done by C4.5, MLC++, etc.), it does a breadth-first tree growth like SLIQ (and PLANET) to prevent possible load imbalance in a parallel computing framework.

Other notable techniques for parallel tree learning include (1) parallel decision tree learning on an SMP architecture based on feature scheduling among processors, including task pipelining and dynamic load balancing for speedup (Zaki, Ho, and Agrawal, 1999); (2) meta-learning schemes that train multiple trees in parallel along with a final arbiter tree that combines their predictions (Chan and Stolfo, 1993);

(3) distributed learning of trees by boosting, which operates over partitions of a large dataset that are exchanged among the processors (Lazarevic, 2001); (4) the SPIES algorithm, which combines the AVC-group idea of RAINFOREST with effective sampling of the training data to obtain a communication- and memory-efficient parallel tree learning method (Jin and Agrawal, 2003b); and (5) a distributed tree learning algorithm that uses only 20% of the communication cost to centralize the data, but achieves 80% of the accuracy of the centralized version (Giannella et al., 2004).

On the theoretical side Caragea, Silvescu, and Honavar (2004) formulated the problem of learning from distributed data and showed different algorithm settings for learning trees from distributed data, each of which is provably exact, that is, they give the same results as a tree learned using all the data in a centralized setting. Approximate algorithms for parallel learning of trees on streaming data have also been recently proposed (Ben-Haim and Yom-Tov, 2008; Jin and Agrawal, 2003a).

MapReduce in Machine Learning: In recent years, some learning algorithms have been implemented using the MapReduce framework. Chu et al. (2007) give an excellent overview of how different popular learning algorithms (e.g., locally weighted linear regression, naïve Bayes classification, Gaussian discriminative analysis, *k*-means, logistic regression, neural networks, principal component analysis, independent component analysis, expectation maximization, and support vector machines) can be effectively solved in the MapReduce framework. However, these algorithms have all been implemented using a shared-memory multiprocessor architecture. Our focus is on scaling learning algorithms (especially ensemble tree learning) to massive datasets using a MapReduce framework deployed on commodity hardware.

2.8 Conclusions

We have presented PLANET, a framework for large-scale tree learning using a MapReduce cluster. We are currently applying PLANET to problems within the sponsored search domain. Our experience is that the system scales well and performs reliably in this context, and we expect results would be similar in a variety of other domains involving large-scale learning problems. Our initial goal in building PLANET was to develop a scalable tree learner with accuracy comparable to a traditional in-memory algorithm, but capable of handling much more training data. We believe our experience in building and deploying PLANET provides lessons in using MapReduce for other nontrivial mining and data processing tasks. The strategies we developed for handling tree learning should be applicable to other problems requiring multiple iterations, each requiring one or more applications of MapReduce.

For future work, our short-term focus is to extend the functionality of PLANET in various ways to support more learning problems at Google. For example, we intend to support split metrics other than those based on variance. We also intend to investigate how intelligent sampling schemes might be used in conjunction with the scalability offered by PLANET. Other future plans include extending the implementation to handle multi-class classification and incremental learning.

Acknowledgments

We thank Ashish Agarwal, Puneet Chopra, Mayur Datar, Oystein Fledsberg, Rob Malkin, Gurmeet Singh Manku, Andrew Moore, Fernando Pereira, D. Sculley, and Diane Tang for their feedback and contributions to this work.

References

Alsabti, K., Ranka, S., and Singh, V. 1998. *CLOUDS: A Decision Tree Classier for Large Datasets*. Technical Reports, University of Florida.

Ben-Haim, Y., and Yom-Tov, E. 2008. A Streaming Parallel Decision Tree Algorithm. In: *Large Scale Learning Challenge Workshop at the International Conference on Machine Learning (ICML)*.

Bradford, J. P., Fortes, J. A. B., and Bradford, J. 1999. *Characterization and Parallelization of Decision Tree Induction*. Technical Report, Purdue University.

Breiman, L. 1996. Bagging Predictors. *Machine Learning Journal*, **24**(2), 123–140.

Breiman, L. 2001. Random Forests. *Machine Learning Journal*, **45**(1), 5–32.

Breiman, L., Friedman, J. H., Olshen, R., and Stone, C. 1984. *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks.

Caragea, D., Silvescu, A., and Honavar, V. 2004. A Framework for Learning from Distributed Data Using Sufficient Statistics and Its Application to Learning Decision Trees. *International Journal of Hybrid Intelligent Systems*, **1**(1–2), 80–89.

Caruana, R., and Niculescu-Mizil, A. 2006. An Empirical Comparison of Supervised Learning Algorithms. Pages 161–168 of: *International Conference on Machine Learning (ICML)*.

Caruana, R., Karampatziakis, N., and Yessenalina, A. 2008. An Empirical Evaluation of Supervised Learning in High Dimensions. Pages 96–103 of: *International Conference on Machine Learning (ICML)*.

Chan, P. K., and Stolfo, S. J. 1993. Toward Parallel and Distributed Learning by Meta-learning. Pages 227–240 of: *Workshop on Knowledge Discovery in Databases at the Conference of Association for the Advancement of Artificial Intelligence (AAAI)*.

Chu, C.-T., Kim, S. K., Lin, Y.-A., Yu, Y., Bradski, G., Ng, A. Y., and Olukotun, K. 2007. Map-Reduce for Machine Learning on Multicore. Pages 281–288 of: *Advances in Neural Information Processing Systems (NIPS) 19*.

Dean, J., and Ghemawat, S. 2004. MapReduce: Simplified Data Processing on Large Clusters. In: *Symposium on Operating System Design and Implementation (OSDI)*.

Duda, R. O., Hart, P. E., and Stork, D. G. 2001. *Pattern Classification*, 2nd ed. New York: Wiley.

Freund, Y., and Schapire, R. E. 1996. Experiments with a New Boosting Algorithm. Pages 148–156 of: *International Conference on Machine Learning (ICML)*.

Friedman, J. H. 2001. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of Statistics*, **29**(5), 1189–1232.

Gao, J., Wu, Q., Burges, C., Svore, K., Su, Y., Khan, N., Shah, S., and Zhou, H. 2009 (August). Model Adaptation via Model Interpolation and Boosting for Web Search Ranking. Pages 505–513 of: *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*.

Gehrke, J., Ramakrishnan, R., and Ganti, V. 1998. RainForest – A Framework for Fast Decision Tree Construction of Large Datasets. Pages 416–427 of: *International Conference on Very Large Data Bases (VLDB)*.

Gehrke, J., Ganti, V., Ramakrishnan, R., and Loh, W.-Y. 1999. BOAT – Optimistic Decision Tree Construction. Pages 169–180 of: *International Conference on ACM Special Interest Group on Management of Data (SIGMOD)*.

Giannella, C., Liu, K., Olsen, T., and Kargupta, H. 2004. Communication Efficient Construction of Decision Trees over Heterogeneously Distributed Data. Pages 67–74 of: *International Conference on Data Mining (ICDM)*.

Jin, R., and Agrawal, G. 2003a. Efficient Decision Tree Construction on Streaming Data. Pages 571–576 of: *SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*.

Jin, R., and Agrawal, G. 2003b. Communication and Memory Efficient Parallel Decision Tree Construction. Pages 119–129 of: *SIAM Conference on Data Mining (SDM)*.

Joshi, M. V., Karypis, G., and Kumar, V. 1998. ScalParC: A New Scalable and Efficient Parallel Classification Algorithm for Mining Large Datasets. Pages 573–579 of: *International Parallel Processing Symposium (IPPS)*.

Kaushik, A. 2007a (August). *Bounce Rate as Sexiest Web Metric Ever*. MarketingProfs. http://www.marketingprofs.com/7/bounce-rate-sexiest-web-metric-ever-kaushik.asp?sp=1.

Kaushik, A. 2007b (May). *Excellent Analytics Tip 11: Measure Effectiveness of Your Web Pages*. Occam's Razor (blog). www.kaushik.net/avinash/2007/05/excellent-analytics-tip-11-measure-effectiveness-of-your-web-pages.html.

Lazarevic, A. 2001. The Distributed Boosting Algorithm. Pages 311–316 of: *SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*.

MacQueen, J. B. 1967. Some Methods for Classification and Analysis of Multivariate Observations. Pages 281–297 of: Cam, L. M. Le, and Neyman, J. (eds), *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1. Berkeley: University of California Press.

Manku, G. S., Rajagopalan, S., and Lindsay, B. G. 1999. Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Datasets. Pages 251–262 of: *International Conference on ACM Special Interest Group on Management of Data (SIGMOD)*.

Mehta, M., Agrawal, R., and Rissanen, J. 1996. SLIQ: A Fast Scalable Classifier for Data Mining. Pages 18–32 of: *International Conference on Extending Data Base Technology (EDBT)*.

Provost, F., and Fayyad, U. 1999. A survey of methods for scaling up inductive algorithms. *Data Mining and Knowledge Discovery*, **3**, 131–169.

Ridgeway, G. 2006. *Generalized Boosted Models: A Guide to the GBM Package*. http://cran.r-project.org/web/packages/gbm.

Rokach, L., and Maimon, O. 2008. *Data Mining with Decision Trees: Theory and Applications*. World Scientific.

Sculley, D., Malkin, R., Basu, S., and Bayardo, R. J. 2009. Predicting Bounce Rates in Sponsored Search Advertisements. Pages 1325–1334 of: *SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*.

Shafer, J. C., Agrawal, R., and Mehta, M. 1996. SPRINT: A Scalable Parallel Classifier for Data Mining. Pages 544–555 of: *International Conference on Very Large Data Bases (VLDB)*.

Vapnik, V. N. 1995. *The Nature of Statistical Learning Theory*. Berlin: Springer.

Zaki, M. J., Ho, C.-T., and Agrawal, R. 1999. Parallel Classification for Data Mining on Shared-Memory Multiprocessors. Pages 198–205 of: *International Conference on Data Engineering (ICDE)*.