SAND2011-0741C

COMET: A Recipe for Learning and Using Large Ensembles on Massive Data

Justin D. Basilico, M. Arthur Munson, Kevin R. Dixon,
Tamara G. Kolda, W. Philip Kegelmeyer
Sandia National Laboratories
Livermore, CA 94551, USA
{jdbasil, mamunso, krdixon, tgkolda, wpk}@sandia.gov

ABSTRACT

The collection of massive volumes of data requires machine learning algorithms that can be applied to distributed data. We describe COMET (Cloud of Massive Ensemble Trees), a recipe for distributed supervised learning consisting of three components: (1) MapReduce is used to parallelize the learning and evaluation tasks and collect the results, (2) an IVoting Random Forest is used for the learning task on each local data partition, and (3) the results of all local ensembles are combined into one massive ensemble and lazy evaluation is used to dynamically subsample it. We propose a new Gaussian approach for lazy ensemble evaluation that is easier to implement and faster to compute than previous approaches. Empirical experiments on two large datasets demonstrate that a) COMET leads to dramatically faster learning than serial IVoting or improved accuracy if training time is equal, and b) lazy ensemble evaluation drastically reduces the cost of making predictions with massive ensembles.

Categories and Subject Descriptors

I.5 [**Pattern Recognition**]: Models—*Statistical*; I.2.6 [**Artificial Intelligence**]: Learning—*induction, concept learning*; H.2.8 [**Database Management**]: Database Applications—*data mining*

General Terms

Algorithms, Performance

Keywords

MapReduce, Decision Tree Ensembles, IVoting, Lazy Ensemble Evaluation, Distributed Learning

1. INTRODUCTION

The integration of computer technology into science and daily life has enabled the collection of massive volumes of data that cannot be practically analyzed on a single commodity computer because these datasets are too large to fit in memory. Consider website transaction logs, credit card records, high-throughput biological assay data, sensor readings, GPS locations of cell phones, and more. Analyzing massive data requires either a) subsampling the data down to a size small enough to be processed on a workstation; b) restricting analysis to streaming methods that analyze a fixed size, evolving working data subset; or c) distributing the data across multiple computers that perform the analyses in parallel.

In this paper, we focus on the problem of learning on massive data. Distributed approaches are attractive because they can exploit multiple processors to construct models faster and/or more accurately than a subsampling or streaming approach running on a single processor. Moreover, the MapReduce framework [16] makes distributed computations straightforward to implement, and the Amazon Elastic Compute Cloud (EC2) makes cluster computing accessible to all by allowing users to rent cluster cycles as needed.

We use a divide-and-conquer approach: the data is partitioned across multiple compute nodes, and each node *independently* constructs one or more classifiers from its data partition. The resulting classifiers (from all nodes) form an ensemble, or committee, model that makes predictions by combining predictions from the constituent classifiers. The MapReduce framework is used for handling data distribution and resource scheduling; in general, our method needs only a single pass for all the individual learners.

Each compute node builds a local IVoting Random Forest. Random Forests [7] is a method for building decision tree ensembles which has been shown to produce accurate classifiers for a wide variety of problem domains [11, 10]. We use a variant of Random Forests that uses IVoting [6] rather than bagging [4] to sample training subsets because it has been shown to work better in a distributed context [14].

All the local ensembles are combined into a mega-ensemble containing thousands of classifers in total. Using such a large ensemble is computationally expensive and generally overkill for data points that are easy to classify. Thus, we employ a *lazy ensemble evaluation* scheme that only uses as many ensemble members as are needed to make a confident prediction. We propose a new Gaussian-based approach for Lazy Ensemble Evaluation (GLEE) that is easier to implement and more scalable than previously proposed approaches.

Thus, as a recipe for massive data analysis, we propose to combine (1) MapReduce for naturally parallelizing the learning and evaluation tasks and collecting the results, (2) an IVoting Random Forest for learning on each local data partition, and (3) lazy ensemble evaluation for efficiently applying the massive ensemble. We call our approach COMET, short for Cloud Of Massive Ensemble Trees.

Our main contributions are as follows:

- We describe COMET, a MapReduce-based framework for distributed IVoting Random Forest ensemble learning which uses a divide-and-conquer approach for learning on massive data. Unlike recent work using MapReduce to learn decision tree ensembles [29], COMET requires only a single MapReduce pass for training.

- We propose a new approach for lazy ensemble evaluation based on a Gaussian confidence interval. Our GLEE technique is easier to implement and more scalable than a previous Bayesian approach [23], and our experimental comparisons show that there is no performance degradation with our new approach.
- Using two publicly available datasets (the larger of which contains 200M examples), we show that COMET produces more accurate models than learning from a subsample. Alternatively, distributed learning with COMET achieves the same accuracy as subsample learning but trains 5–10 times faster.

2. RELATED WORK

2.1 Supervised Learning on MapReduce

Distributed versions of many supervised learning algorithms have been implemented using MapReduce. Chu et al. [15] note that several common learning algorithms can be written in summation forms and that the sums can be computed in parallel by MapReduce operations. Often the summation represents a single computational step in an iterative algorithm. This iterative category includes logistic regression [15], linear [15] and non-linear support vector machines [13], backpropagation neural networks [15, 24], decision trees [29], and belief propagation for graphical models [22]. Deodhar et al.'s hybrid co-clustering and linear regression algorithm, SCOAL, involves three summations per iteration [17]. The downside with these iterative algorithms is that they require multiple MapReduce jobs, which means multiple scans through the data as well as overhead from setting up and shutting down MapReduce jobs. Coordinating multiple MapReduce iterations can also be complicated. PLANET [29], for example, constructs a decision tree using MapReduce primitives but requires implementing a separate job control system to supervise the search for node splits and to reduce job setup and teardown overhead.

Algorithms that require a single distributed computation step—such as locally weighted linear regression [15], naïve Bayes [15], and Gaussian discriminate analysis [15]—can be a better fit for MapReduce. For example, Alham et al. [1] propose a distributed support vector machine (SVM) learning algorithm implemented with a single MapReduce job. The map step runs the standard SVM algorithm multiple times, in parallel, on disjoint partitions of the training data. The reduce stage forms a single SVM by taking the union of the support vectors learned from each partition. While this algorithm is approximate (the final SVM is different from an SVM trained serially on all the data), the distributed SVM is comparable in accuracy to a serial one and faster to train.

Gao et al. [21] build an ensemble of decision trees with purely random topologies. Leaf count statistics are collected in parallel and are based on the full data set. This approach outperforms an ensemble of purely random trees with leaf counts computed from one data partition (each).

2.2 Distributed Ensembles

Ensemble learning has long been used for large-scale distributed machine learning. Instead of converting a learning algorithm to be natively parallel, run the (unchanged) algorithm multiple times, in parallel, on each data partition [12, 18, 19, 14]. An aggregation strategy combines the set of learned models into an ensemble that is usually as accurate, if not more accurate, than a single model trained from all data would have been. For example, Chan and Stolfo [12] study different ways to aggregate decision tree classifiers trained from disjoint partitions. They find that voting the trees in an ensemble is sufficient if the partition size is big enough to produce accurate trees. They propose arbiter trees to intelligently combine and boost weaker trees to form accurate ensembles in spite of small partitions. Domingos [18] similarly learns sets of decision rules from partitioned data, but combines them using a simpler weighted vote. Yan et al. [34] train many randomized SVM models with a MapReduce job; a second job runs forward stepwise selection to choose a subset with good performance. The final ensemble aggregates predictions through a simple vote. In this work we use simple voting as our aggregation strategy because our data partitions are relatively large.

Our distributed learning strategy is inspired by Chawla et al.'s work on distributed IVoting [14]. They empirically compare IVoting applied to all training data to distributed IVoting (DIVoting) in which IVoting is run multiple times, independently, on disjoint partitions of the training data. Unlike the studies described above, they trained multiple models from each data partition. Their results show that DIVoting achieves comparable classification accuracy to IVoting with a faster running time, and better accuracy than distributed bagging that used the same sample sizes. Our work differs from theirs because a) we use MapReduce to implement DIVoting (they used MPI), b) our data is 190X larger, and c) we apply lazy ensemble evaluation to speed up predictions from large ensembles. The work of Wu et al. [33] is also closely related to ours. They also train a decision tree ensemble using MapReduce, but only train one decision tree per partition (we run IVoting on each parition), do not use lazy ensemble evaluation, and evaluate the ensemble on a single small data set with 699 records.

Moretti et al. [27] build a framework to support easy learning of distributed ensembles. The framework is like MapReduce but with builtin support for specialized data partitioning and the ability to specify data as test data (to avoid storing it in replicated file systems). They demonstrate that their framework scales on synthetic datasets as big as 54GB.

Ye et al. [35] take advantage of efficient internode communication in MPI to implement distributed boosted decision trees. Their algorithm partitions the data by feature so that each node contains a disjoint subset of the attributes. Nodes compute the goodness of splitting on different features in parallel and send a controller node the best split from their view of the data. The controller communicates the best global split to the worker nodes, which then begin the search for the next tree node split. While all the communications are small in this algorithm, the frequent communications would be very expensive on MapReduce.

2.3 Lazy Ensemble Evaluation

Lazy ensemble evaluation is the strategy of only evaluating as many ensemble members as needed to make a good prediction. Whereas much research has studied removing unnecessary models from an ensemble (called ensemble pruning) [32], only a few studies have used lazy ensemble evaluation to dynamically speed up prediction time in proportion to the ease or difficulty of each data point. Fan et al. [19] use a Gaussian confidence interval to decide if ensemble evaluation can stop early for a test point. Their method differs from the one described in Section 3.3 in that a) ensemble members are always evaluated from most to least accurate, and b) confidence intervals are based on where evaluation could have reliably stopped on validation data. A fixed ordering is not necessary in our work because the base models should have similar accuracy; this leads to a simpler Gaussian lazy ensemble evaluation rule.

Hernández-Lobato et al. [23] use Bayesian inference to decide when ensemble evaluation can be stopped early. We compare to this method in our experiments, and refer to it as the Madrid Lazy Ensemble Evaluation (MLEE). In MLEE, the distribution of vote frequencies for different classes is modeled as a multinomial distribution with a uniform Dirchlet prior. The posterior distribution of the class vote proportions is updated at each evaluation step to reflect the observed base model prediction. MLEE computes the probability that the final ensemble predicts class c by enumerating the possible prediction sequences for the as-yet unqueried ensemble members, based on the current posterior distribution. Ensemble evaluation stops when the probability of some class exceeds the specified confidence level or when all base models have voted. MLEE is exponential in the number of classes but is $O(m^2)$ for binary classification (m ensemble members) and approximations exist to make it tractable for multi-class problems [26].

Markatopoulou et al. [25] propose a more complicated runtime ensemble pruning, where the choice of which base models to evaluate is decided by a meta-model trained to choose the most reliable models for different regions of the input data space. Their method can achieve better accuracy than using the entire ensemble, but generally will not lead to faster ensemble predictions.

3. COMET RECIPE

COMET is a recipe for large-scale distributed ensemble learning and efficient ensemble evaluation. The recipe has three components:

1. **MapReduce:** We write our distributed learning algorithm using MapReduce to easily parallelize the learning task. The mapper tasks build classifiers on local data partitions ("blocks" in MapReduce nomenclature), and a single reducer can collect the output. If the learned ensemble is large and/or the number of data points to be evaluated is large, evaluation can also be parallelized using MapReduce.
2. **IVoting Random Forest:** Each mapper runs a variant on Random Forests that replaces bagging with IVoting. The mapper builds an ensemble based on its local block of data (assigned by MapReduce). IVoting has the advantage that it gives more weight to difficult examples. Unlike boosting [20], however, each model in the ensemble votes with equal weight, allowing us to trivially merge the ensembles from all mappers into a single large ensemble.
3. **Lazy Ensemble Evaluation:** Many inputs are "easy" and the vast majority of the ensemble members agree on the classification. For these cases, querying a small sample of the members is sufficient to determine the ensemble's prediction with high confidence. Lazy ensemble evaluation significantly lowers the prediction time for ensembles.

The rest of this section describes these three components in more detail.

3.1 Distributed Learning via MapReduce

We take a coarse-grained approach to distributed learning that minimizes communication and coordination between compute nodes. We assume that the training data is partitioned randomly into blocks in such a way that class distributions are roughly the same across all blocks. Such shuffling can be accomplished in a simple pre-processing step that maps each data item to a random block.

In the learning phase, each mapper independently learns a predictive model from an assigned data block. The learned models are aggregated together into a final ensemble model by a reducer. This is the only step that requires internode communication, and only the final models are transmitted (not the data). Thus, we only require a single MapReduce pass for training. We call this distributed learning strategy coarse-grained because the task of learning a model is broken into large subtasks that are computed in parallel.

We implement the above strategy in the MapReduce framework [16] because the framework's abstractions match our needs, although other parallel computing frameworks (e.g., MPI) could also be used. To use MapReduce, one loads the input data into the framework's distributed file system and defines map and reduce functions to process key-value pair data during Map and Reduce stages, respectively. Mappers execute a map function on an assigned data block (usually read from the node's local file system). The map function produces zero or more key-value pairs for each input; in our case, the values correspond to learned trees (with random keys). During the Reduce stage, all the pairs emitted during the Map stage are grouped by key and passed to reducer nodes that run the reduce function. The reduce function receives one key and all the associated values produced by the Map stage. Like the map function, the reduce function can emit any number of key-value pairs. Resulting pairs are written to the distributed file system. The MapReduce framework manages data partitioning, task scheduling, data replication, and handling failures. The reducer(s) can write all the learned trees to a single output file or to multiple files to be used later in a parallel evaluation MapReduce pass.

The map and reduce functions for distributed ensemble learning are straightforward. The map function trains an ensemble on its local data block and then emits the learned trees. In this algorithm, we give each tree a random key to partition the ensemble for a distributed evaluation phase. The reduce function combines the trees for the key assigned to it. Thus, if trees are emitted to a single partition ($p = 1$), all trees will be reduced to one output file. If $p > 1$, each reducer will receive approximately $1/p$ of the randomly assigned trees, and there will be p output files.

We describe the GLEE rule in Section 3.3, but here we discuss how it can be parallelized with MapReduce. Each mapper has a portion of the entire ensemble and executes the GLEE routine locally. For each data point, we get a tally of the votes for the different classes. In most cases, this will require only a small portion of the entire ensemble, and we will be able to output a decision. In the rare cases where this portion of the ensemble is not sufficient to determine the final evaluation, we can then process that data point further (e.g., via another MapReduce pass or serial computations). Note that further processing is never required if the ensemble is small enough for every mapper to hold in memory.

In contrast to PLANET [29], as discussed previously, we stress that the learning phase requires only one MapReduce pass (two if you need a pre-processing pass to randomly distribute the data). For PLANET, an entire MapReduce pass is required to learn each level in each decision tree.

3.2 IVoting Random Forest

The key step in distributed learning above is constructing an ensemble from the local data partition using IVoting. IVoting (Importance-sampled Voting) [6] builds an ensemble by repeatedly applying the base learning algorithm (e.g., decision tree induction [8, 30]) to small samples called *bites*. Unlike bagging [4], examples are sampled with non-uniform probability. Suppose that k IVoting iterations have been run, producing ensemble E_k comprised of k base classifiers. To form the $k+1^{\text{st}}$ bite, training examples (x, y) are drawn randomly. If E_k incorrectly classifies x, (x, y) is added to training set B_{k+1}. Otherwise (x, y) is added to B_{k+1} with probability $e(k)/(1-e(k))$, where $e(k)$ is the error rate of E_k. This process is repeated until $|B_{k+1}|$ reaches the specified bite size b. Out-of-bag (OOB) [5] predictions are used to get unbiased estimates of $e(k)$ and E_k's accuracy on sampled points x. The OOB prediction for x is made by voting only the ensemble members that did not see x during training, i.e., x was outside the base models' training sets.

IVoting's sequential and weighted sampling is reminiscent of boosting [20]. Indeed, IVoting is similar to boosting in terms of accuracy [6]. IVoting differs from boosting, however, in that each base model receives equal weight for deciding the ensemble's prediction. This property simplifies merging the multiple ensembles produced by independent IVoting runs on disjoint data blocks. Finally, the base learning algorithm constructs models more quickly from bites than from the samples used in bagging or boosting because the bites are generally a small subset of the available data.

Breiman [6] showed that IVoting sampling generates bites containing roughly half correct and half incorrect examples. Thus, we use a variation of IVoting which draws (with replacement) 50% of the bite from the examples E_k correctly classifies and 50% from the examples E_k incorrectly classifies (based on OOB predictions). This implementation avoids the possibility of drawing and rejecting large numbers of correct examples for ensembles with very high accuracy. Algorithm 1 summarizes the IVoting algorithm.

Algorithm 1: Importance-sampled Voting (IVoting)

```
Input: Dataset $D \in (\mathcal{X}, \mathcal{Y})^*$; Ensemble size $m$; Bite size
       $b \in \mathbb{N}$; Base learner $L : (\mathcal{X}, \mathcal{Y})^* \to (\mathcal{X} \to \mathcal{Y})$
Output: Ensemble $E$
Initialize $D_0^+ = D$, $D_0^- = D$, $V_{oob}[\cdot, \cdot] = 0$, $E = \emptyset$;
for $i \in [1, m]$ do
    // Create the bite to train on.
    $B_i^+ = b/2$ uniform random samples from $D_{i-1}^+$;
    $B_i^- = b/2$ uniform random samples from $D_{i-1}^-$;
    $B_i = B_i^+ + B_i^-$;
    // Train a new ensemble member.
    $T_i = L(B_i)$;
    Add $T_i$ to $E$;
    // Update running votes.
    for $(x_j, y_j) \notin B_i$ do
        $V_{oob}[j, T_i(x_j)]$ += 1;
    $D_i^+ = \{(x_j, y_j) \in D \mid y_j = \arg\max_z V_{oob}[j, z]\}$;
    $D_i^- = \{(x_j, y_j) \in D \mid y_j \neq \arg\max_z V_{oob}[j, z]\}$;
return $E$;
```

Any classification learning algorithm can be used for the base learner in IVoting. Our experiments use decision trees [30, 31] because they generally form accurate ensembles [3]. The trees are grown to full size (i.e., each leaf is pure or contains fewer than ten training examples) using information gain as the splitting criterion. We use full-sized trees because they generally yield slightly more accurate ensembles [3]. To increase the diversity of trees and reduce training time for data sets with large numbers of features, only a random subset of features are considered when choosing the test predicate for each tree node. This attribute subsampling is used in random forests and has been shown to improve performance and decrease training time [7]. We employ the random forest heuristic for choosing the attribute sample size $d' = \lfloor 1 + \log_2 d \rfloor$, where d is the total number of attributes. As a whole, the learning algorithm run on each data block is a variant of Random Forests in which IVoting generates the training samples instead of bagging.

3.3 Lazy Ensemble Evaluation

A major drawback to large ensembles is the cost of querying all ensemble members for their predictions. In practice, many data points are easy to classify: the vast majority of the ensemble members agree on the classification. For these cases, querying a small sample of the members is sufficient to determine the ensemble's prediction with high confidence.

We exploit this phenomena via lazy ensemble evaluation where we decide if ensemble voting can be stopped early, *on a case by case basis for each data point*, while guaranteeing with high probability that the "lazy" prediction is the same as the prediction would be by using the entire ensemble. The risk that lazy evaluation stops voting too early (i.e., the probability that the early prediction is different from what the full ensemble prediction would have been) is bounded by a user-specified parameter α. Algorithm 2 lists the lazy ensemble evaluation procedure. Let x be a data point to classify using ensemble E, with E containing m base models. Initially all m models are in the unqueried set U. In each step, a model T is randomly chosen and removed from U to vote on x; the vote is added to the running tallies of how many votes each class has received. Based on the accumulated tallies and how many ensemble members have not yet voted, the stopping criterion decides if it is safe to stop and return the classification receiving the most votes. If it is not safe, a new ensemble member is drawn and the process is repeated until it is safe to stop or all m ensemble members have been queried.

In binary categorization, the vote of each base model can be modeled as a Bernoulli random variable. Accordingly, the distribution of votes for the full ensemble follows a binomial distribution with proportion parameter p. Provided that the number of members queried by the i^{th} step in Algorithm 2 is sufficiently large, we can invoke the Central Limit Theorem and approximate the binomial distribution with a Gaussian distribution.

We propose Gaussian Lazy Ensemble Evaluation (GLEE), which uses the Gaussian distribution to infer a $(1-\alpha)$ confidence interval around the observed mean $\hat{p}$. The interval is used to test the hypothesis that the unobserved proportion of positive votes p falls on the same side of 0.5 as $\hat{p}$ (and consequently, that the current estimated classification agrees with the full ensemble's classification). If 0.5 falls outside the interval, GLEE rejects the null hypothesis that

Algorithm 2: Lazy Ensemble Evaluation

```
Input: Input $x \in \mathcal{X}$
Input: Ensemble $E$ with $m$ members of $f : \mathcal{X} \to \{1, ..., c\}$
Input: $\alpha$, max. disagreement freq. for lazy vs. full eval.
Input: Vote stopping criteria
       Stop : $(\mathbb{N}_0^c, \mathbb{N}_1, \mathbb{R} \in [0, 1]) \to \{\text{true}, \text{false}\}$
Output: Approximate prediction from $E$ for input $x$.
Set $U = E$, $V = [0, ..., 0]$, $|V| = c$;
for $i \in [1, m]$ do
    Sample $T$ uniformly from $U$;
    Remove $T$ from $U$;
    Evaluate $v_i = T(x)$;
    Increment $V[v_i]$;
    if Stop$(V, m, \alpha)$ then
        return $\arg\max_i V[i]$;
return $\arg\max_i V[i]$
```

p and $\hat{p}$ are on different sides of 0.5 and terminates voting early. Formally, denote the interval bounds as $\hat{p} \pm \rho\delta$, where

$$\delta = z_{\alpha/2} \frac{\sigma}{\sqrt{n}} = z_{\alpha/2} \frac{\sqrt{\hat{p}(1-\hat{p})}}{\sqrt{n}}$$

and

$$\rho = \begin{cases} \sqrt{\frac{m-n}{m-1}} & \text{if } n > 0.05m \\ 1 & \text{otherwise.} \end{cases}$$

The critical value $z_{\alpha/2}$ is the usual value from the standard normal distribution. The finite population correction (FPC) ρ accounts for the fact that base models are drawn from a finite population (the ensemble). Intuitively, uncertainty about p shrinks as the set U becomes small. To ensure the Gaussian approximation is reasonable, GLEE only stops evaluation if at least 15 models have voted. We found this (somewhat arbitrary) threshold gave reasonable results.

The above hypothesis test only requires the lower bound (if $\hat{p} > 0.5$) or the upper bound (if $\hat{p} < 0.5$). Consequently we can improve GLEE's statistical power by computing a one-sided interval; i.e., use z_α instead of $z_{\alpha/2}$. When the GLEE stopping criteria is invoked, the *leading class* (the class with the most votes so far) is treated as class 1, and the *runner-up class* is treated as class 0.[1] GLEE stops evaluation early if the lower bound $\hat{p} - \delta$ is greater than 0.5.[2]

4. EXPLORATION OF LAZY ENSEMBLE EVALUATION

This section explores the efficacy of the GLEE rule across a wide range of ensemble sizes and for varying confidence levels. We simulate votes from large ensembles to explore the rule's behavior and to compare it to the MLEE rule.

The stopping thresholds for both methods are pre-computed for each ensemble size and stored in a table that is indexed by the number of votes received by the leading class. Pre-computing and caching the thresholds provides a minor speed-up for GLEE, but is necessary to make MLEE practical for large ensembles. To avoid numerical overflows, we compute the factorials required for MLEE in log-space.

Ensemble votes are simulated as follows. A uniform random number $p \in [0, 1]$ is generated to be the proportion of ensemble members that vote for class 1. The correct label for the example is 1 if $p \geq 0.5$ and 0 otherwise. Each model in the ensemble votes by sampling from a Bernoulli random variable with probability $\Pr(x = 1) = p$. The ensemble is evaluated until the stopping criterion is satisfied or all m ensemble members have voted. The lazy prediction, under the different stopping rules, and the prediction from evaluating the full ensemble are compared to the correct label to determine their relative accuracies. This process is repeated 5000 times to simulate making predictions for 5000 data points.

We report the results in terms of *relative votes* and *relative accuracy. Relative votes* is the average fraction of ensemble members evaluated before lazy evaluation stopped. *Relative accuracy* is the accuracy of lazy evaluation (over the 5000 examples) divided by the accuracy of non-lazy evaluation.

Figure 1a compares five approaches to lazy ensemble evaluation. All five methods provide similar speed-ups, with G1-FPC and G2-FPC requiring slightly fewer votes than the others. More importantly, we see that there is a significant benefit to applying these methods for ensembles with as few as 100 members and that the benefit becomes greater as the ensemble size grows.

In Figures 1b and 1c, we fix the ensemble size at $m = 10000$ and vary α. As we might expect, larger values of α require evaluating a smaller subset of the ensemble. Figure 1c shows that we can still achieve 99% of the original accuracy with a relatively large value of $\alpha = 0.01$. This requires as little as 2% of the ensemble, on average (Figure 1b). Finally, the G1-FPC rule is as good or better than MLEE in terms of relative accuracy. In the rest of the paper we use G1-FPC for lazy evaluation and will refer to it as the GLEE rule. Section 5 presents results on real data.

5. EXPERIMENTS

To understand how well our COMET approach performs we ran a set of experiments on two large real-world datasets.

5.1 Datasets

The data sets are described in detail below; the characteristics are summarized in Table 1.

Table 1: Dataset Characteristics

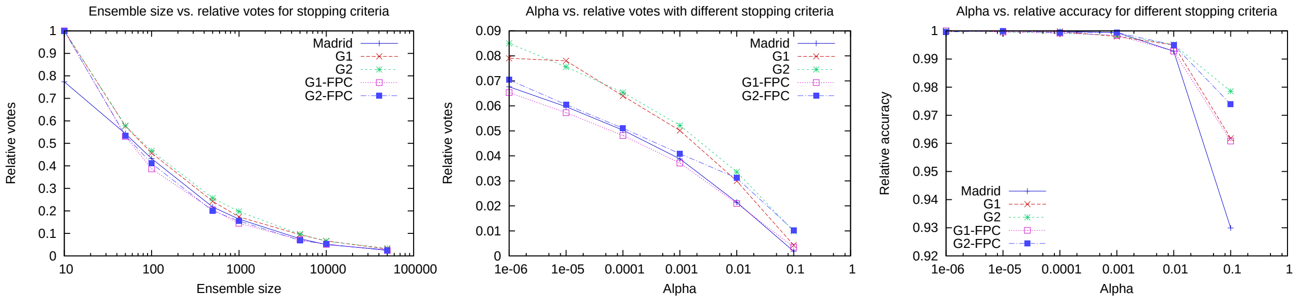
Name	Train	Test	Features	% Positive
ClueWeb	200M	1M	63	48.4%
eBird	1M	400K	1143	31.8%

5.1.1 ClueWeb09 Dataset

ClueWeb09 [9] is a web crawl of over 1 billion web pages (approximately 5TB compressed, 25TB uncompressed). For this dataset we use language categorization as the prediction task. Specifically, the task is to predict if a given web page's language is English or non-English. The features are counts of alpha-numeric characters ($0 - 9$, $a - z$, $A - Z$) plus one additional feature for any other character, for a total of 63 features. Counts are normalized to sum to 1.

[1] This class relabeling trick also enables direct application of GLEE to multiclass problems.

[2] This one-sided test is slightly biased because the procedure effectively chooses to compute a lower or upper bound after "peeking" at the data to determine which class is the current majority class. When $\hat{p}$ is close to 0.5 the bias results in slightly inaccurate confidence intervals that do not contain p with the specified $(1 - \alpha)$ frequency. On average, however, the impact of this bias on the accuracy of the lazy prediction is only noticeable for very small α (see Section 4).



(a) Relative Votes: Vary Ensemble Size (b) Relative Votes: Vary α (c) Relative Accuracy: Vary α

Figure 1: Relative number of votes and accuracy for different stopping criteria on simulated ensembles. The five stopping criteria are Madrid (MLEE), G1 (Gaussian one-tail), G2 (Gaussian two-tail), G1-FPC (G1 with finite population correction) and G2-FPC (G2 with finite population correction). In (a), $\alpha = 0.0001$. In (b) and (c), ensemble size is fixed to $m = 10000$.

We used MapReduce to extract features for each web page and randomly divide the data into blocks (by mapping each example to a random key). Preprocessing the full ClueWeb dataset took approximately 2 hours on our Hadoop cluster and created 1000 binary files totaling approximately 259 GB and containing nearly 1B examples. From this, we randomly extracted 200M training and 1M testing examples. The training data was divided into 200 blocks, each approximately 1/4GB in size and containing 1M examples.

5.1.2 *eBird*

The second dataset we use to evaluate COMET is the US48 eBird reference dataset [28]. Each record corresponds to a checklist collected by a bird watcher and contains counts of how many birds, broken down by species, were observed at a given location and time. In addition to the count data, each record includes attributes describing the environment in which the checklist was collected (e.g., climate, land cover), the time of year, and how much effort the observer spent. The eBird data tests how well COMET scales for problems with data having hundreds of attributes.

The prediction task in our experiment is to predict if an American Goldfinch (Carduelis tristis) will be observed at a given place and time based on the environmental and data collection attributes. We chose American Goldfinches because they are widespread throughout the United States (and thus, frequently observed) and exhibit complex migration patterns that vary from one region to another (making the prediction task hard). We used the data from 1970–2008 for training and the data from 2009 for testing. All non-zero counts were converted to 1 to create a binary prediction task. There is a total of 1143 features; specifically, we used all attributes except meta-data attributes intended for data filtering (COUNTRY, STATE_PROVINCE, SAMPLING_EVENT_ID, LATITUDE, LONGITUDE, OBSERVER_ID, SUBNATIONAL2_CODE).

After pre-processing, the data set contains 1.4M examples and requires 6.2GB of storage. We subdivided the data into 14 training and 6 testing blocks. Each block contains 70K examples and requires 1/4 GB of storage.

5.2 Implementation Details

For our experiments, we used the open-source Hadoop platform (version 0.21), which includes MapReduce and the Hadoop distributed filesystem (HDFS). We used the machine learning algorithm implementations provided by the Cognitive Foundry [2] (open-source software).

Table 2: Accuracy for Different Bite Sizes

Bite Size	ClueWeb Accuracy	eBird Accuracy
100	n/a	0.7265
500	n/a	0.7496
1K	0.8911	0.7614
5K	0.9089	0.7753
10K	0.9163	0.7755
50K	0.9316	0.7713
100K*	0.9359	0.7699
150K	0.9370	n/a
200K	0.9377	n/a

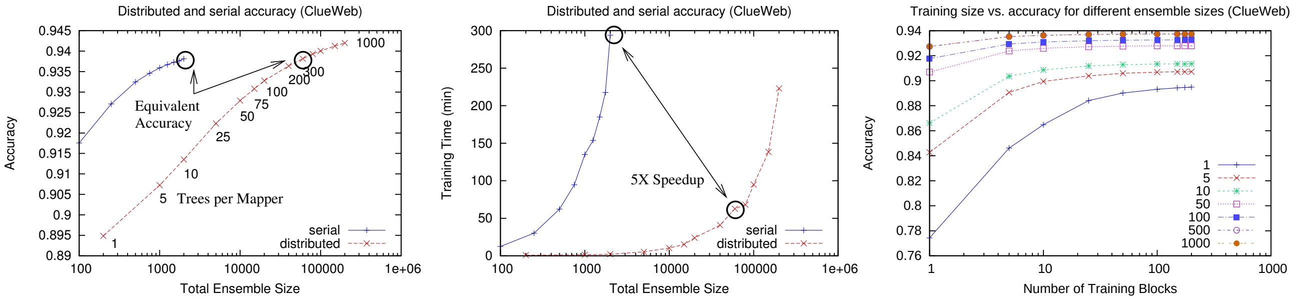
* eBird bite size was 70K (approx. data partition size).

All experiments were run on a cluster with 65 worker nodes. Each worker node has one quad-core Intel i-720 (2.66 Ghz) processor, 12 GB of memory, four 2 TB disk drives, and 1Gb Ethernet networking. Each worker node was configured to execute up to four map or reduce tasks concurrently. To make running times directly comparable, we ran the serial algorithm on an individual worker node with a copy of the training data sample on the local filesystem.

We loaded the data into HDFS with a big enough block size to ensure each file was contained one block (i.e., 256MB, vs. the default 64MB block size). Large block sizes improve accuracy by allowing IVoting to sample from more diverse examples, at the expense of spending more time per worker node. Since each mapper produces relatively few outputs, we reduced the size of Hadoop's internal buffers to maximize the memory available for the learning algorithm.

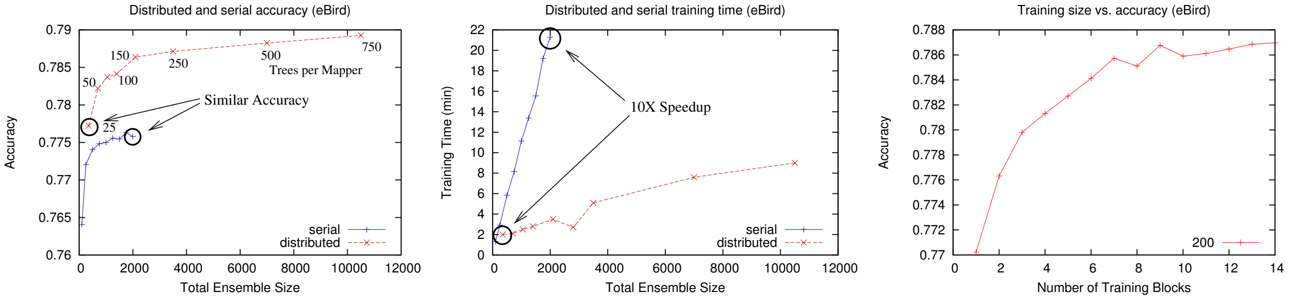
In GLEE, the straightforward way to sample models (without replacement) from the ensemble is to generate a new random number for each ensemble member that is evaluated. If the cost of generating a random number is relatively expensive, lazy evaluation may not provide enough of a speed-up and may even slow down ensemble evaluation. To avoid this, our GLEE implementation permutes the ensemble order once at load time. Each ensemble evaluation is started from a different random index in this order. Thus, only a single random number is generated per ensemble prediction.

In all experiments the bite size b was set to 100K for ClueWeb and 10K for eBird. These values were chosen by running IVoting for 1000 iterations on one data block



(a) Accuracy Comparison (b) Training Time Comparison (c) Vary Training Data & Ensemble Size

Figure 2: COMET results on ClueWeb data. Figures (a) and (b) compare distributed learning (200M examples split into 200 blocks) to serial learning (1M examples). Circles highlight the distributed ensemble with similar accuracy to the best serial ensemble. Figure (c) illustrates the effect of varying the number of training data blocks (1M examples per block). Different lines correspond to varying size of local ensemble (IVoting iterations). Lines for ensemble sizes 500 and 1000 are superimposed.



(a) Accuracy Comparison (b) Training Time Comparison (c) Vary Training Data

Figure 3: COMET results on eBird data. Figures (a) and (b) compare distributed learning (1M examples split into 14 blocks) to serial learning (70K examples). Circles highlight the distributed ensemble with similar accuracy to the best serial ensemble. Figure (c) illustrates the effect of varying the number of training data blocks (70K examples per block) with 200 ensemble members trained per block.

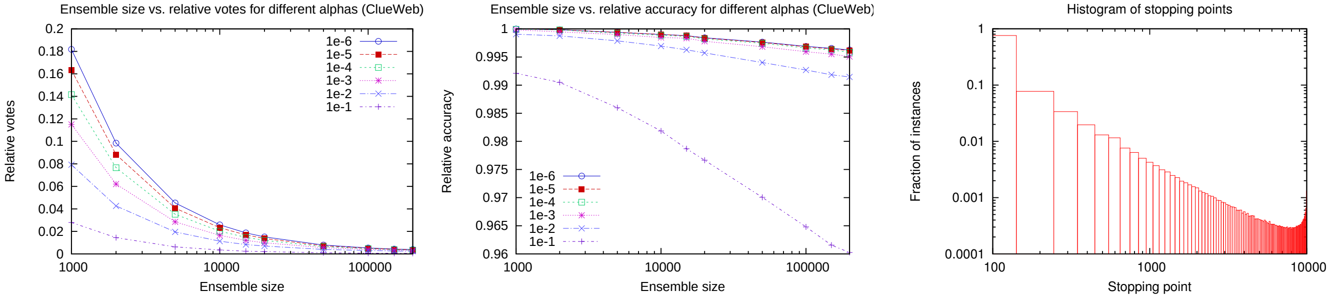
with different bite sizes and measuring the accuracy on the test data. For eBird, accuracy peaked at 10K (Table 2), possibly because larger bite sizes reduced the diversity of the base models. For ClueWeb, accuracy started to plateau around 100K (Table 2). While larger bite sizes yielded small improvements, they also resulted in trees with big enough memory footprints to significantly limit how many ensemble members could be trained per core.

5.3 Results

Figure 2 compares COMET (distributed) with serial IVoting Random Forests for the ClueWeb09 data with full ensemble evaluation (i.e., GLEE is not used). The serial code trains on a single block (1M examples) using 9 different ensemble sizes: 100, 250, 500, 750, 1000, 1250, 1500, 1750, 2000. The accuracy ranges from 91.8% (for the smallest ensemble) up to 93.8% (for the largest ensemble time). The training time ranges from 12min to 5hr. COMET trains on 200 blocks (200M examples), varying across 13 different values for the local ensemble size: 1, 5, 10, 25, 50, 75, 100, 200, 300, 400, 500, 750, 1000. The total ensemble size is 200 times the local ensemble size; thus, the largest total ensemble has 200K members. The accuracy ranges from 89.5% (corresponding to a local ensemble size of 1 and a total ensemble size of 200) to 94.2% (corresponding to a local ensemble size of 1000 and a total ensemble size of 200K) with time varying from less than 1min to 3hr, respectively. As a point of comparison, the distributed model achieves an accuracy of 93.8% (the same as the best serial model) in only 60min, corresponding to a total ensemble size of 60K (300 trees per block). Thus, we achieve a 5X speed-up in training time without sacrificing any accuracy.

Figure 3 compares COMET (distributed) with IVoting Random Forests for the eBird data (again, without GLEE). The serial code trains on a single block (70K examples) using the same 9 different ensemble sizes as used for the ClueWeb09 data. The accuracy ranges from 76.4% (for the smallest ensemble) up to 77.6% (for the largest ensemble time). The training time ranges from 1–20min. COMET trains on 14 blocks (1M examples), varying across 8 different values for the local ensemble size: 25, 50, 75, 100, 150, 250, 500, 750. The total ensemble size is 14 times the local ensemble size; thus, the largest total ensemble has 10,500 members. The accuracy ranges from 77.7% (better than the best serial accuracy) to 78.9% with time varying from less than 2-9min. The best accuracy achieved by the serial version is 77.5% with a total ensemble size of 2000 and a training time of 21min; the distributed version improves on this with an accuracy of 77.8% for a total ensemble size of only 350 (local size of 25) and a training time of 2min. Thus, we see a 10X speed-up in training time.

Figures 2c and 3c vary the number of data blocks used



(a) ClueWeb Relative Votes

(b) ClueWeb Relative Accuracy

(c) Histogram of ClueWeb Early Stopping Points

Figure 4: GLEE results on ClueWeb data using one-tailed Gaussian with finite-population correction (G1-FPC) with a minimum of 15 votes. (a) and (b) show the relative votes and accuracy for various ensemble sizes and values of α. Values are calculated relative to using the entire ensemble. (c) Histogram of number of evaluations made (x-axis) for an ensemble of size 10000 and $\alpha = 0.01$.

in the training. For ClueWeb, all parameters are the same as above except for the following. The number of blocks is varied from 1 to 200 (with 1M examples per block), and the local ensemble size is varied from 1 to 1000. We clearly see a flattening out as the number of blocks increases, essentially flat-lining at 40. Likewise, the gain for increasing the ensemble size becomes small (invisible in this graph) for a local ensemble size of more than 250. For eBird, all parameters are the same as above except that we fix the local ensemble size at 200 and vary the number of blocks between 1 and 14. The accuracy increases almost monotonically with the number of blocks used.

Figure 4 shows the results of using GLEE (G1-FPC) with ensembles of different sizes on the ClueWeb data (results on eBird data are similar and are omitted for space reasons). As expected, the results show that decreasing α increases the average number of votes (Figure 4a) and the relative accuracy for any size ensemble (Figure 4b). For all ensemble sizes and α values evaluated, using the early stopping criteria provides a significant speed-up over using the entire ensemble. This speed-up increases as the ensemble size is increased, even for small values of α. For the ClueWeb data, we can achieve greater than 99% relative accuracy for $\alpha = 0.01$. For an ensemble of size 1K, fewer that 10% of the ensemble needs to be evaluated, on average, and for an ensemble of size 100K, that drops to less than 0.1%. Thus, the cost of evaluating a large ensemble can be largely mitigated via GLEE. Figure 4c shows a histogram of the number of evaluations needed by GLEE with $\alpha = 0.01$, providing insight into why the stopping method works — the vast majority of instances require evaluating only a small proportion of the ensemble.

6. CONCLUSION

We have presented COMET, our recipe for supervised learning on massive, distributed data, which consists of using MapReduce for parallelization, using IVoting Random Forests for the learning scheme, and using GLEE for lazy ensemble evaluation. One of the key questions with large datasets is whether it is necessary to actually use all of the data or if a sample of the data would suffice. For both ClueWeb and eBird, we saw increases in accuracy by using more data. More significant, however, is the improvement in training time, with up to a 10X improvement. Some previous works have not shown accuracy results as compared to serial training [21, 29]; here we show that there is no detriment and perhaps even some improvement in accuracy via distributed learning.

The main issue with COMET is that we can easily create extremely large ensembles by training in parallel. However, our results have demonstrated the effectiveness of using lazy ensemble evaluation to efficiently make predictions with large and small ensembles. Depending on the ensemble size, the savings in evaluation cost can easily be 100X or better. Moreover, the relative error from using lazy evaluation may be lower than the jitter one would expect from different runs of a randomized learning algorithm.

In future work, we seek to remove the need for the preprocessing step of randomizing the distribution of the data before the algorithm is used and instead leave as much data in place as possible. Data shuffling can become a bottleneck because it requires copying the entire data; this is especially problematic if the data is spread across multiple data centers. One idea is to add additional MapReduce iterations to share ensemble members and/or important examples between the different blocks in the hope that this will result in less communication and data movement than moving all of the data as a pre-processing step.

A lot of large datasets also have highly skewed class distributions. We would like to extend this recipe to handle such cases, perhaps by replicating data for minority classes to make the data distribution in each block more even for training and by adopting different methodologies for the base classifiers for the ensemble to make them more robust to skew. While learning from skewed data may be more difficult, we expect that lazy ensemble evaluation will provide a greater speed-up on skewed data (large ensembles will likely be important for achieving high accuracy on skewed data).

Acknowledgments.

Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

7. REFERENCES

[1] N. K. Alham, M. Li, S. Hammoud, Y. Liu, and M. Ponraj. A distributed SVM for image annotation. In *FSKD'10*, vol. 6, pp. 2983–2987. IEEE, 2010.

[2] J. Basilico, Z. Benz, and K. R. Dixon. The cognitive foundry: A flexible platform for intelligent agent modeling. In *BRIMS'08*, 2008. Software available from `http://foundry.sandia.gov/`.

[3] E. Bauer and R. Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, 36(1-2):105–139, 1999.

[4] L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[5] L. Breiman. Out-of-bag estimation. `ftp://ftp.stat.berkeley.edu/pub/users/breiman/OOBestimation.ps`, 1996.

[6] L. Breiman. Pasting small votes for classification in large databases and on-line. *Machine Learning*, 36(1):85–103, 1999.

[7] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[8] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and Regression Trees.* Chapman & Hall/CRC, 1984.

[9] J. Callan, M. Hoy, C. Yoo, and L. Zhao. The ClueWeb09 dataset. `http://boston.lti.cs.cmu.edu/Data/clueweb09/`, 2009.

[10] R. Caruana, N. Karampatziakis, and A. Yessenalina. An empirical evaluation of supervised learning in high dimensions. In *ICML'08*, pp. 96–103, 2008.

[11] R. Caruana and A. Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *ICML'06*, pp. 161–168, 2006.

[12] P. K. Chan and S. J. Stolfo. Learning arbiter and combiner trees from partitioned data for scaling machine learning. In *KDD'95*, pp. 39–44, 1995.

[13] E. Y. Chang, K. Zhu, H. Wang, H. Bai, J. Li, Z. Qiu, and H. Cui. Parallelizing support vector machines on distributed computers. In *NIPS 20*, pp. 257–264, 2008.

[14] N. V. Chawla, L. O. Hall, K. W. Bowyer, and W. P. Kegelmeyer. Learning ensembles from bites: A scalable and accurate approach. *Journal of Machine Learning Research*, 5:421–451, 2004.

[15] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-Reduce for machine learning on multicore. In *NIPS 19*, pp. 281–288, 2007.

[16] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[17] M. Deodhar, C. Jones, and J. Ghosh. Parallel simultaneous co-clustering and learning with map-reduce. In *IEEE Intl. Conf. on Granular Computing*, pp. 149–154, 2010.

[18] P. Domingos. Using partitioning to speed up specific-to-general rule induction. In *AAAI-96 Workshop on Integrating Multiple Learned Models*, pp. 29–34. AAAI Press, Menlo Park, CA, USA, 1996.

[19] W. Fan, F. Chu, H. Wang, and P. S. Yu. Pruning and dynamic scheduling of cost-sensitive ensembles. In *AAAI'02*, pp. 146–151, 2002.

[20] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *ICML'96*, pp. 148–156, 1996.

[21] W. Gao, R. Grossman, P. Yu, and Y. Gu. Why naive ensembles do not work in cloud computing. In *ICDM'09 Workshops*, pp. 282–289, 2009.

[22] J. E. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. In *AISTATS'09*, pp. 177–184, 2009.

[23] D. Hernández-Lobato, G. Martínez-Muñoz, and A. Suárez. Statistical instance-based pruning in ensembles of independent classifiers. *IEEE Trans. on Pattern Anal. and Mach. Intell.*, 31(2):364–369, 2009.

[24] Z. Liu, H. Li, and G. Miao. MapReduce-based backpropagation neural network over large scale mobile data. In *ICNC'10*, pp. 1726–1730, 2010.

[25] F. Markatopoulou, G. Tsoumakas, and I. Vlahavas. Instance-based ensemble pruning via multi-label classification. In *ICTAI'10*, 2010.

[26] G. Martínez-Muñoz, D. Hernández-Lobato, and A. Suárez. Statistical instance-based ensemble pruning for multi-class problems. In *ICANN'09*, pp. 90–99, 2009.

[27] C. Moretti, K. Steinhaeuser, D. Thain, and N. V. Chawla. Scaling up classifiers to cloud computers. In *ICDM'08*, pp. 472–481, 2008.

[28] M. A. Munson, K. Webb, D. Sheldon, D. Fink, W. M. Hochachka, M. Iliff, M. Riedewald, D. Sorokina, B. Sullivan, C. Wood, and S. Kelling. *The eBird Reference Dataset, Version 2.0.* Cornell Lab of Ornithology and National Audubon Society, Ithaca, NY, 2010.

[29] B. Panda, J. Herbach, S. Basu, and R. Bayardo. PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce. *Proc. VLDB Endowment*, 2(2):1426–1437, 2009.

[30] J. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

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

[32] G. Tsoumakas, I. Partalas, and I. Vlahavas. An ensemble pruning primer. In *Applications of Supervised and Unsupervised Ensemble Methods*, pp. 1–13. Springer-Verlag, 2009.

[33] G. Wu, H. Li, X. Hu, Y. Bi, J. Zhang, and X. Wu. MReC4.5: C4.5 ensemble classification with MapReduce. In *ChinaGrid '09*, pp. 249–255, 2009.

[34] R. Yan, M.-O. Fleury, M. Merler, A. Natsev, and J. R. Smith. Large-scale multimedia semantic concept modeling using robust subspace bagging and MapReduce. In *LS-MMRM'09*, pp. 35–42, 2009.

[35] J. Ye, J.-H. Chow, J. Chen, and Z. Zheng. Stochastic gradient boosted distributed decision trees. In *CIKM'09*, pp. 2061–2064, 2009.