TELL ME SOMETHING NEW
A NEW APPOACH TO PARALLEL LEARNING AND ITS APPLICATION TO BOOSTING

Anonymous Authors[1]

ABSTRACT

This paper presents several methods for reducing the I/O and communication requirements of parallelized boosting.

The inner loop of the tree boosting algorithm is a search for splitting rules. This search is driven by estimates of expected loss. Traditionally, these estimates were based on *all available data*. Previous work () has shown that early stopping can be used to reduce the amount of data required.

In this work we extend the work on early stopping in two ways, both of which further increase the speed of the boosting algorithm. First, we show how early stopping can be used to parallelize the boosting algorithm. Second, we show how memory pressure can be reduced in when the examples have non-uniform weights.

We describe these improvement and show their mathematical properties. We implemented our methods for boosting and evaluated them on the splice-site prediction problem (**?**Agarwal et al., 2014). We compare our performance to that of SparkML () XGBoost (Chen & Guestrin, 2016), LightGBM (Ke et al., 2017). Our implementation is about ten times faster than the alternatives (fill in when we have details).

1 INTRODUCTION

A major bottleneck for machine learning from big data is the movement of data between storage units. This includes disk, memory and cache organization and communication between different computers. In this work we describe two new ways to reduce this I/O bottleneck and accelerate learning in the context of boosting trees.

Many machine learning algorithms, including XGBoost (**?**)nd lightGBM (), follow a similar pattern. At each iteration the complete training set is scanned. Then, based on statistics calculated from the training set, the model is updated to decrease the loss.

Scanning all of the data is obviously optimal from the point of view of the statistical estimates. However, the incremental improvemen in the model, given an increase from a fracton of the data to the full data, might be small. If it is sufficiently small, then stopping early and proceeding to the next iteration can result in significantly faster running time.

This idea was introduced by Wald(Wald, 1973) in the 1940s under the title "Sequential analysis" (SA). We describe SA precisely in Section **??**.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the Systems and Machine Learning (SysML) Conference. Do not distribute.

As an illustrative example, suppose that our goal is to find a classifier from $M_1, \ldots, M_k$ whose error rate is close to the minimum across that k models. The standard approach is to scan all of the traininqg examples, compute the average error of each model, and choose the model with the minimal error. The SA approach is to read the examples one by one, each time updating the error estimate for each model, and emply a specifically designed *stopping rule* to decide when to stop and which model to output. An example where SA will stop much before the standard approach is when one rule has an error rate of 0.1 while all of the other rules have error of 0.5. Using stopping rules to accelerate boosting has been studied before by Domingo and Watanabe (Domingo & Watanabe, 2000) and by Bradley and Schapire (Bradley & Schapire, 2007).

The first contribution of this paper is a new protocol for parallelizing boosting algorithms, based on stopping rules, which we call "tell me something new" or **TMSN**. Most parallelized algorithms, including Spark-ML, XGBoost and LightGBM use Bulk-Synchronization **??** (BS). On the other hand, **TMSN** is an asynchronous protocol with a weak notion of common state.

We now describe **TMSN** in the context of boosting. The main computation step of any boosting algorithm is the search for a weak rule that is slightly better than random guessing. Usually, the stated goal is to find the *best* weak rule. However, for adaboost to make progress, *any* rule whose error is significantly smaller than $1/2$ can be used.

This relaxation of the requirement makes it possible to use stopping rules.

As the standard goal of boosting algorithms is to find the *best weak rule* most implementation of boosting algorithms *scan all of the training data* at each iteration, which becomes very slow when the data is large. As mentioned above (Domingo & Watanabe, 2000; Bradley & Schapire, 2007) proposed to use early stopping in the non-parallelized context. The i Data paralel and feature paralel implementations of boosting () accelerate this computation, but still require reading each training point. **TMSN** parallelization is a variant of feature parallelism. As in bulk-synchronous feature-parallel boosting, each **TMSN** worker scans locally stored training examples to evaluate the error of a set of weak rules. However *unlike* bulk-synchronized, only a (typically small) fraction of the data is scanned. The workers all use a stopping rule to decide when to terminate the search. The stopping rule fires when it identifies a weak rule whose error is (with high probability) smaller than $1/2 - \gamma$. When this good weak rule has been identified, it interrupts the search, adds the found rule to the strong rule, and broadcasts this strong rule. The other workers, upon recieving the new strong rule, interrupt their own search and use the new rule.

This rough version of **TMSN** would work if the workers were synchronized and communication was instantanous. Both are unrealistic assumptions. To remove the need for these assumptions we add a *global measure of progress*, which is an upper bound on the expected loss of the strong hypothesis. When a worker broadcasts a new strong rule, it pairs it with this upper bound. When a worker recieves a strong rule, it acccepts it only if the newcomers upper bound is smaller than the current upper bound. The full details are in section **??**

We call this protocol "tell me something new" because the workers send out information only when they have "something new" which in our case means a significantly better strong rule. This protocol has several desirable properties:

Both XGBoost and LightGBM Bulk-Synchronization **??** model of parallel computation. In this model the workers are all synchronized at the start of each boosting iteration. This establishes a well defined *common state* at the synchronization boundary which simplifies the reasoning about the algorithm an it's interaction with the hardware and the OS.

The main contribution of this p"tell me something new" or **TMSN** *we do away with both synchronization and common state*.

1. **No head-node** Most distributed systems rely on a head-node that synchronizes the workers. The head node is a single-point-of-failure and a bottleneck, especially systems with a large number of workers. **TMSN** avoids these problems because it does not require a head-node.

2. **Reduced communication bandwidth:** In typical bulk-synchronous protocols all workers send and recieve information from the head node at each step. This can mean a lot of communication even when there is little progress in terms of reducing loss. The workers in **TMSN** remain mute when as long as they don't find anything useful.

3. **No blocking** The communication operations are all non-blocking, in other words, at no point does a worker wait for a response from another worker. This means that CPU utilzation is not reduce by wait times.

4. **Robustness** As a result of the fact that there is no synchronization and no head node, they system is very robust. If a single computer is slow or crashes, the effect on the other computers is small. Computers can be removed or added at any time and will quickly catch up to the current best model and start contributing their cycles to the search.

The second contribution of this paper is a method to improve the usage of main memory when training examples have non-uniform weights. Intuitively, when a large fraction of the examples have very low weight, they contribute little to the error estimates. We show how to quantify the effect of non-uniform weights in general and show how selective sampling can increase the accuracy of the estimates. We also propose a stopping rule when learning from non-uniform weights.

The rest of the paper is organized as follows.

The rest of the paper is divided into four sections. First we give a general description of **TMSN** in Section 2. Then we introduce a special application of our algorithm, namely **Sparrow**, in Section **??**. After that we describe in more details of the algorithms and the system design of **Sparrow** in Section 4. Finally, we present empirical results in Section 5.

2 Tell Me Something New

We start with a general description of **TMSN** in the context of the Statistical Query model (). In the following section we will describe **TMSN** for boosting.

Most learning algorithm can be described as a combination of statistical estimation steps and minimization steps. Examples include mini-batch SGD which estimate the gradient at each iteration, Decision tree algorithms that maximize a purity function and Boosting algorithms that minimize weighted error. This broad family of algorithms can be formalized as Statistical Query algorithms (SQ algorithms) **??**.

Te statistical query model defines an interface between the learning algorithm and the training data. Rather than giving the algorithm direct access to the examples, it is restricted to making queries of the form "What is the expected value of the random variable X?" For example, consider the query "what is the error rate of the classifier c?" this query can be written as

$$\text{What is} E\left[\mathbb{1}(c(\vec{x}) \neq y)\right] \;?$$

where $\mathbb{1}(c(\vec{x}) \neq y)$ is the error indicator function that is 1 when $c(\vec{x}) \neq y$ and 0 otherwise.

In general, a statistical query is defined by a function over the sample space (i.e. a random variable) $Q : X \to [0, 1]$ an error parameter $\epsilon > 0$ and an accuracy parameter $\delta > 0$ An answer to the query is an estimate of the expected value $E[Q]$ which we denote $\hat{E}[Q]$. The difference between the true expectation and the estimate shoud be smaller than ϵ with probability at least $1 - \delta$ over the random chice of the training set T.

$$P_{T\sim\mathcal{D}^n}\left[|E[Q] - \hat{E}[Q]| \geq \epsilon\right] \leq \delta$$

The most common estimator $\hat{E}[Q]$ is the unweighted average of the random variable over a training set of size n:

$$\hat{E}[Q] = \frac{1}{n}\sum_{i=1}^{n} Q(\vec{x}_i)$$

A standard bound (Hoeffding) on the error of the estimate is

$$P_{T\sim\mathcal{D}^n}\left[|E[Q] - \hat{E}[Q]| \geq \epsilon\right] \leq 2e^{-2\epsilon^2 n} \qquad (1)$$

Inverting the relation we find that $n = (1/2\epsilon^2)\log\frac{2}{\delta}$ is a sufficiently large sample. This relationship holds in the worst case, in other words, when the distribution of Q has maximal variance. I the variance of Q is smaller, than a smaller sample will suffice. However, without making additional assumptions on the distribution of Q, how can we take advantage of this gap?

Wald /cite proposed an ingenious answer to this question. samples (n above) is not chosen in advance. Instead, after each sample a so-called "stopping rule" is evaluated. The stopping rule has three possible outputs: "M_1 has smaller loss", "M_2 has smaller loss" or "Insufficient data, take another sample". The result is that if the gap is large, the number of samples used be the stopping rule is much smaller than n, reducing the memory load of the program.

3 TMSN FOR BOOSTING

Next, we describe how **TMSN** is applied to boosting. Boosting algorithms (Schapire & Freund, 2012) are iterative, they generate a sequence of *strong rules* of increasing accuracy.

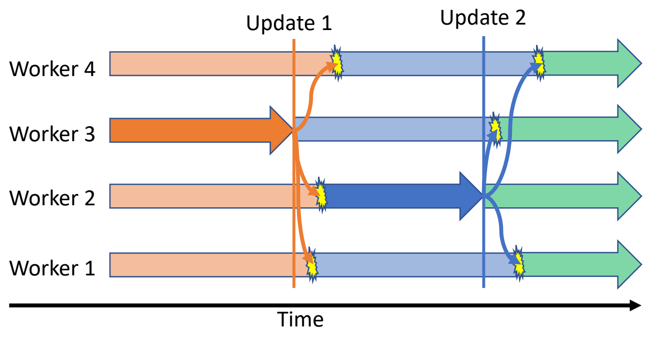

Figure 1. **Execution timeline of a TMSN system** System consists of four workers. The first update occurs when worker 3 identifies a better classifier H_1. It then replaces H_0 with H_1 and broadcasts (H_1, z_1) to the other workers. The other workers receive the message the at different times, depending on network congestion. At that time they interrupt the scanner (yellow explosions) and start using H_1. Next, worker 2 identifies an improved rule H_2 and the same process ensues.

The strong rule at iteration T is a weighted majority over T of the the weak rules in $\mathcal{W}$.

$$H_T(x) = \text{sign}\left(\sum_{t=1}^{T}\alpha_t h_t(x)\right)$$

For the purpose of **TMSN** we define $\mathcal{H}$ to be the set of strong rules combining any number of weak rules from $\mathcal{W}$.

Boosting algorithms can be interpreted as gradient descent algorithm (Mason et al., 1999). Specifically, if we define the *potential* of the strong rule H with respect to the training set $\mathcal{S}$ to be

$$Z_{\mathcal{S}}(H_T) \doteq \frac{1}{n}\sum_{i=1}^{n} e^{-y_i H_T(x_i)},$$

then AdaBoost is equivalent to coordinate-wise gradient descent, where the coordinates are the elements of $\mathcal{W}$. Suppose we have the strong rule H and consider changing it to $H + \alpha h$ for some $h \in \mathcal{W}$ and for some small α. The derivative of the potential wrt α is:

$$\left.\frac{\partial}{\partial\alpha}\right|_{\alpha=0} Z_{\mathcal{S}}(H+\alpha h) = \frac{1}{n}\sum_{i=1}^{n}\left.\frac{\partial}{\partial\alpha}\right|_{\alpha=0} e^{-y_i(H(x_i)+\alpha h(x_i))} = \frac{1}{n}\sum_{i=1}^{n} -y_i h($$

Our goal is to minimize the average potential $Z_{\mathcal{S}}(H_{T+1})$, therefor our goal is to find a weak rule h that makes the gradient negative. Another way of expressing this goal is to find a weak rule with a large empirical *edge*:

$$\hat{\gamma}(h) \doteq \sum_{i=1}^{n} w_i y_i h(x_i) \text{ where } w_i = \frac{1}{Z}e^{-y_i H(x_i)}; Z = \sum_{i=1}^{n} e^{-y_i H(x_i)} \qquad (2)$$

w_i defines a distribution over the training examples, with respect to which we are measuring the correlation between $h(x_i)$ and y_i. This is the original view of boosting, which is the process of finding weak rules with significant edges with respect to different distributions. We distinguish between the empirical edge $\hat{\gamma}(h)$, which depends on the sample, and the **true** edge, which depends on the underlying distribution:

$$\gamma(h) \doteq E_{(x,y)\sim\mathcal{D}}(w(x,y)yh(x)) \text{ where } w(x,y) = \frac{1}{Z}\mathcal{D}(x,y)e^{-yH(x)} \tag{3}$$

and Z is the normalization factor with respect the the true distribution $\mathcal{D}$.

A small but important observation is that boosting does not require finding the weak rule with the **largest** edge at each iteration. Rather, it is enough to find a rule for which we are sure that it has a significant (but not necessarily maximal) advantage. More precisely, we want to know that, with high probability over the choice of $\mathcal{S} \sim \mathcal{D}^n$ the rule h has a significant *true* edge $\gamma(h)$.

Sequential Analysis and Early Stopping The standard approach when looking for the best weak rule is to compute the error of candidate rules using all available data, and then select the rule h that maximizes the empirical edge $\hat{\gamma}(h)$. However, as described above, this can be over-kill. Observe that if the true edge $\gamma(h)$ is large it can be identified as such using a small number of examples.

Bradley and Schapire (Bradley & Schapire, 2007) and Domingo and Watanabe (Domingo & Watanabe, 2000) proposed using early stopping to take advantage of such situations. The idea is simple: instead of scanning through all of the training examples when searching for the next weak rule, a *stopping rule* is checked for each $h \in \mathcal{W}$ after each training example, and if this stopping rule "fires" then the scan is terminated and the h that caused the rule to fire is added to the strong rule. We use early stopping in our algorithm.

For reasons that will be explained in the next section, we use a different stopping rule than (Bradley & Schapire, 2007) or (Domingo & Watanabe, 2000). We use a stopping rule proposed in (Balsubramani, 2014) for which they prove the following

Theorem 1 (based on (Balsubramani, 2014) Theorem 4) *Let M_t be a martingale $M_t = \sum_i^t X_i$, and suppose there are constants $\{c_k\}_{k\geq 1}$ such that for all $i \geq 1$, $|X_i| \leq c_i$ w.p. 1. For $\forall \sigma > 0$, with probability at least $1-\sigma$ we have*

$$\forall t : |M_t| \leq C\sqrt{\left(\sum_{i=1}^t c_i^2\right)\left(\log\log\left(\frac{\sum_{i=1}^t c_i^2}{|M_t|}\right) + \log\frac{1}{\sigma}\right)},$$

where C is a universal constant.

Effective Sample Size Equation 3 defines $\hat{\gamma}(h)$, which is an estimate of $\gamma(h)$. How accurate is this estimate? Our initial gut reaction is that if $\mathcal{S}$ contains n examples the error should be about $1/\sqrt{n}$. However, when the examples are weighted this is clearly wrong. Suppose, for example that k out of the n examples have weight one and the rest have weight zero. Obviously in this case we cannot hope for an error smaller than $1/\sqrt{k}$.

A more quantitative analysis follows. Suppose that the weights of the examples in the training set $\mathcal{S} = \{(x_1,y_1),\ldots,(x_n,y_n)\}$ are $w_1 = w(x_1,y_1),\ldots,w_n = w(x_n,y_n)$. Thinking of finding a good weak rule in terms of hypothesis testing, the null hypothesis is that the weak rule h has no edge. Finding a rule that is significantly better than random corresponds to rejecting the hypothesis that $\gamma(h) = 0$. Assuming the null hypothesis, $y_i h(x_i)$ is $+1$ with probability 1/2 and -1 with probability 1/2. From central limit theorem and assuming n is larger than 100, we get that the null distribution for $\hat{\gamma}(h) = \sum_{i=1}^n w_i y_i h(x_i)$ is normal with zero mean and standard deviation $\sum_{i=1}^n w_i^2$. The statistical test one would use in this case is the **Z**-test for

$$\mathbf{Z} = \frac{\hat{\gamma}(h)}{\sqrt{\sum_{i=1}^n w_i^2}} = \frac{\sum_{i=1}^n w_i y_i h(x_i)}{\sqrt{\sum_{i=1}^n w_i^2}} \tag{4}$$

As should be expected, the value of **Z** remains the same whether or not $\sum_{i=1}^n w_i = 1$. Based on Equation 4 we define the *effective number of examples* corresponding to the un-normalized weights $w_1,\ldots,w_n$ as:

$$n_{\text{eff}} \doteq \frac{\left(\sum_{i=1}^n w_i\right)^2}{\sum_{i=1}^n w_i^2} \tag{5}$$

Owen (Owen, 2013) used a different line of argument to arrived at a similar measure of the effective samples size for a weighted sample.

The quantity n_{eff} plays a similar role in large deviation bounds such as the Hoeffding bound (Hoeffding, 1963) (details ommitted). It also plays a central role in Theorem 1 and thus in the stopping rule that we use.

To understand the important role that n_{eff} plays in our algorithm, supppose the training set is of size n and that only $m \ll n$ examples can fit in memory. Our approach is to start by placing a random subset of size m into memory and then run multiple boosting iterations using this subset. As the strong rule improves, n_{eff} decreases and as a result the stopping rule based on Theorem 1 requires increasingly more examples before it is triggered. When n_{eff}/m crosses a pre-specified threshold the algorithm flushes out the training examples currently in memory and samples a new set of m examples using acceptance probability proportional to their weights. The new examples have uniform weights and therefor after sampling $n_{\text{eff}} = m$.

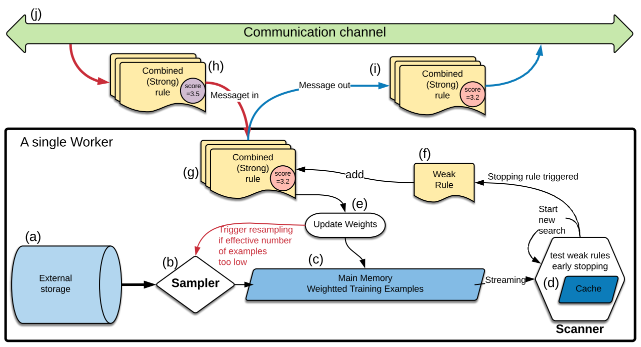

Figure 2. The **Sparrow** system architecture.

Intuitively, weighted sampling utilizes the computer's memory better than uniform sampling because it places in memory more difficult examples and fewer easy examples. The result is better estimates of the edges of specialist[1] weak rules that make predictions on high-weight difficult examples.

Another concern is the fraction of the examples that are selected. In the method described here the expected fraction is $(\frac{1}{n}\sum_{i=1}^{n} w_i)/(\max_i w_i)$.

4 System Architecture and Algorithms

The **Sparrow** distributed system consists of a collection of independent workers connected through a shared communication channel. There is no synchronization between the workers and no identified "head node" to coordinate the workers. The result is a highly resilient system in which there is no single point of failure and the overall slowdown resulting from machine slowness or failure is proportional to the fraction of faulty machines.

Each worker is responsible for a finite (small) set of weak rules. This is a type of feature-based parallelization (Caragea et al., 2004). The worker's task is to identify a weak rule, based on one of the features in the set, that has a significant edge.

We assume each worker stores *all* of the training examples on it's local disk (element **(a)** in Figure 2)[2].

Our description of **Sparrow** is in two parts. First, we describe the design of a *single* **Sparrow** worker. Following that, we describe how concurrent workers use the **TMSN** protocol to update each other. Figure 2 depicts the architecture of a single computer and its interaction with the communication channel. Pseudocode with additional detail is provided in the supplementary material to this paper.

4.1 A single Sparrow worker

As was said above, each worker is responsible for a set of the weak rules. The worker's task is to identify a rule that has a significant edge (Equation 3). The worker consists of two subroutines that can execute in parallel: a **Scanner (d)** and a **Sampler (b)**. We describe each subroutine in turn.

The Scanner (element **(d)** in Figure 2) The Scanner's task is to read training examples sequentially and stop when it has identified one of the rules to be a *good* rule. More specifically, at any time point the Scanner stores the current strong rule H_t, a set of candidate weak rules $\mathcal{W}$ (which define the candidate strong rules of **TMSN**) and a target edge γ_t. The scanner scans the training examples stored in memory sequentially, one at a time. It computes the weight of the examples using H_t and then updates a running estimate of the edge of each weak rule $h \in \mathcal{W}$.

The scan stops when the stopping rule determine that the true edge of a particular weak rule $\gamma(h_t)$ is, with high probability, larger than a threshold γ. The worker then adds the identified weak rule h_t **(f)** to the current strong rule H_t to create a new strong rule H_{t+1} **(g)**.

The worker computes a "performance score" z_{t+1} which is an upper bound on the Z-score the strong rule by adding the weak rule to it. The pair (H_{t+1}, z_{t+1}) is broadcast to the other workers **(i)**. The worker then resumes it's search using the strong rule H_{t+1}.

The Sampler Our assumption is that the entire training dataset does not fit into main memory and is therefore stored in external storage **(a)**. As boosting progresses, the weights of the examples become increasingly skewed, making the dataset in memory effectively smaller. To counteract that skew, the **Sampler** prepares a *new* training set, in which all of the examples have equal weight, by using selective sampling. When the effective number of examples associated with the old training set becomes too small, the scanner stops using the old training set and starts using the new one.[3]

The sampler uses selective sampling by which we mean that the probability that an example (x, y) is added to the sample is proportional to $w(x, y)$. Each added example is assigned

[1]Specialist weak rules and their advantages are described in Section sec:Algorithm

[2]In other words, the training data it replicated across all of the computers. This choice is made to maximize accuracy. If the data is too large to fit into the disk of a single worker, then it can be randomly partitioned between the computers. The cost is a potential increase in the difference between training error and test error

[3]The sampler and scanner can run in parallel on separate cores. However in our current implementation the worker alternates between Scanning and sampling.

an initial **weight** of 1. [4]

Incremental Updates: Our experience shows that the most time consuming part of our algorithms is the computation of the predictions of the strong rules H_t. A natural way to reduce this computation is to perform it incrementally. In our case this is slightly more complex than in XGBoost or LightGBM, because **Scanner** scans only a fraction of the examples at each iteration. To implement incremental update we store for each example, whether it is on disk or in memory, the results of the latest update. Specifically, we store for each training example the tuple (x, y, w_s, w_l, H_l), Where x, y are the feature vector and the label, H_l is the strong rule last used to calculate the weight of the example. w_l is the weight last calculated, and w_s is example's weight when it was last sampled by the sampler. In this way **Scanner** and **Sampler** share the burden of computing the weights, a cost that turns out to be the lion's share of the total run time for our system.

4.2 Communication between workers

Communication between the workers is based on the **TMSN** protocol. As explained in Section 4.1, when a worker identifies a new strong rule, it broadcasts (H_{t+1}, z_{t+1}) to all of the other workers. Where H_{t+1} is the new strong rule and z_{t+1} is an upper bound on the true Z-value of H_{t+1}. One can think of z_{t+1} as a "certificate of quality" for H_{t+1}.

When a worker receives a message of the form (H, z), it either accepts or rejects it. Suppose that the worker's current strong rule is H_t whose performance score is z_t. If $z_t < z$ then the worker interrupts the Scanner and restarts it with $(H_t, z_t) \leftarrow (H, z)$. If $z_t \geq z$ then (H, z) is discarded and the scanner continues running uninterrupted.

5 Experiments

In this section we describe the results of experiments comparing the run time of **Sparrow** with those of two leading implementations of boosted trees: XGBoost and LightGBM.

Setup We use a large dataset that was used in other studies of large scale learning on detecting human acceptor splice site (Sonnenburg & Franc, 2010; Agarwal et al., 2014). The learning task is binary classification. We use the same training dataset of 50 M samples as in the other work, and validate the model on the testing data set of 4.6 M samples. The training dataset on disk takes over 27 GB in size.

[4]There are several known algorithms for selective sampling. The best known one is rejection sampling where a biased coin is flipped for each example. We use a method known as "minimal variance sampling" (Kitagawa, 1996) because it produces less variation in the sampled set.

Algorithm	Instance	Instance Memory	Training
XGBoost, in-memory	`x1e.xlarge`	122 GB	41
XGBoost, off-memory	`r3.xlarge`	30.5 GB	156
LightGBM, in-memory	`x1e.xlarge`	122 GB	34
LightGBM, off-memory	`r3.xlarge`	30.5 GB	44
TMSN, sample 10%	`c3.xlarge`	7.5 GB	57.4 (1 17.7 (10

Table 1. Experiments on the Splice Site Detection Task

As the code is not fully developed yet, we restrict our trees to one level so-called "decision stumps". We plan to perform comparisons using multi-level trees and more than two labels. We expect similar runtime performance there. To generate comparable models, we also train decision stumps in XGBoost and LightGBM (by setting the maximum tree depth to 1).

Both XGBoost and LightGBM are highly optimized, and support multiple tree construction algorithms. For XGBoost, we selected approximate greedy algorithm for the efficiency purpose. LightGBM supports using sampling in the training, which they called *Gradient-based One-Side Sampling (GOSS)*. GOSS keeps a fixed percentage of examples with large gradients, and then randomly sample from remaining examples with small gradients. We selected GOSS as the tree construction algorithm for LightGBM.

All algorithms in comparison optimize the exponential loss as defined in AdaBoost. We also evaluated the final model by calculating its area under precision-recall curve (AUPRC) on the testing dataset.

Finally, the experiments are all conducted on EC2 instances from Amazon Web Services. Since XGBoost requires 106 GB memory space for training this dataset in memory, we used instances with 120 GB memory for such setting. Detailed description of the setup is listed in Table 5.

Evaluation Performance of each of the algorithm in terms of the exponential loss as a function of time on the testing dataset is given in Figure 3. Observe that all algorithms achieve similar final loss, but it takes them different amount of time to reach that final loss. We summarize these differences in Table 5 by using the convergence time to an almost optimal loss of 0.061. Observe XGBoost off-memory is about 27 times slower than a single **Sparrow** worker which is also off-memory. That time improves by another factor of 3.2 by using 10 machines instead of 1.

In Figure 4 we perform the comparison in terms of AUPRC. The results are similar in terms of speed. However, in this case XGBoost and LightGBM ultimately achieve a slightly better AUPRC. This is baffling, because all algorithms work by minimizing exponential loss.

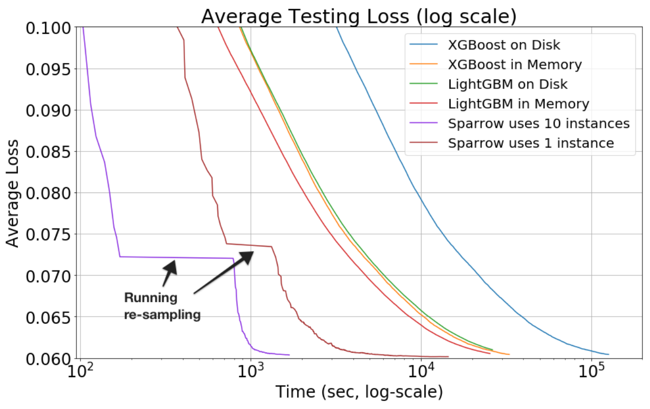

Figure 3. Comparing the average loss on the testing data using **Sparrow**, XGBoost, and LightGBM, lower is better. The period of time that the loss is constant for **Sparrow** is when the algorithm is generating a new sample set.

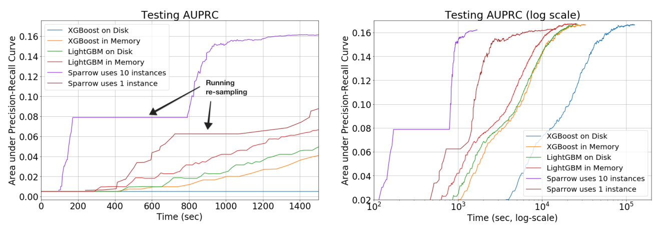

Figure 4. Comparing the area under the precision-recall curve (AUPRC) on the testing data using **Sparrow**, XGBoost, and LightGBM, higher is better. (left) Normal scale, clipped on right. (right) Log scale, clipped on left. The period of time that the AUPRC is constant for **Sparrow** is when the algorithm is generating a new sample set.

Conclusions While the results are exciting plenty of work remains. We plan to extend the algorithm to boosting full trees as well as other types of classifiers. In addition, we observe that run time is now dominated by the time it takes to create new samples, we have some ideas for how to significantly reduce the sampling time.

References

Agarwal, A., Chapelle, O., Dudk, M., and Langford, J. A Reliable Effective Terascale Linear Learning System. *Journal of Machine Learning Research*, 15:1111–1133, 2014. URL http://jmlr.org/papers/v15/agarwal14a.html.

Balsubramani, A. Sharp Finite-Time Iterated-Logarithm Martingale Concentration. *arXiv:1405.2639 [cs, math, stat]*, May 2014. URL http://arxiv.org/abs/1405.2639. arXiv: 1405.2639.

Bradley, J. K. and Schapire, R. E. FilterBoost: Regression and Classification on Large Datasets. In *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, pp. 185–192, USA, 2007. Curran Associates Inc. ISBN 978-1-60560-352-0. URL http://dl.acm.org/citation.cfm?id=2981562.2981586.

Caragea, D., Silvescu, A., and Honavar, V. 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, January 2004. ISSN 1448-5869. doi: 10.3233/HIS-2004-11-210. URL https://content.iospress.com/articles/international-journal-of-hybrid-intelligent-syste his010.

Chen, T. and Guestrin, C. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pp. 785–794, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2939785. URL http://doi.acm.org/10.1145/2939672.2939785.

Domingo, C. and Watanabe, O. Scaling Up a Boosting-Based Learner via Adaptive Sampling. In *Knowledge Discovery and Data Mining. Current Issues and New Applications*, Lecture Notes in Computer Science, pp. 317–328. Springer, Berlin, Heidelberg, April 2000. ISBN 978-3-540-67382-8 978-3-540-45571-4. doi: 10.1007/3-540-45571-X_37. URL https://link.springer.com/chapter/10.1007/3-540-45571-X_37.

Hoeffding, W. Probability Inequalities for Sums of Bounded Random Variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963. ISSN 0162-1459. doi: 10.2307/2282952. URL http://www.jstor.org/stable/2282952.

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T.-Y. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 3146–3154. Curran Associates, Inc., 2017. URL http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boostin pdf.

Kitagawa, G. Monte Carlo Filter and Smoother for Non-Gaussian Nonlinear State Space Models. *Journal of Computational and Graphical Statistics*, 5(1):1–25, 1996. doi: 10.1080/10618600.1996.10474692. URL https://www.tandfonline.com/doi/abs/10.1080/10618600.1996.10474692.

Mason, L., Baxter, J., Bartlett, P., and Frean, M. Boosting Algorithms As Gradient Descent. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pp. 512–518, Cambridge, MA, USA, 1999. MIT Press. URL http://dl.acm.org/citation.cfm?id=3009657.3009730.

Owen, A. B. *Monte Carlo Theory, Methods and Examples*. 2013. URL http://statweb.stanford.edu/~owen/mc/.

Schapire, R. E. and Freund, Y. *Boosting: Foundations and Algorithms*. MIT Press, 2012. ISBN 978-0-262-01718-3. Google-Books-ID: blSReLACtToC.

Sonnenburg, S. and Franc, V. COFFIN: A Computational Framework for Linear SVMs. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, pp. 999–1006, USA, 2010. Omnipress. ISBN 978-1-60558-907-7. URL http://dl.acm.org/citation.cfm?id=3104322.3104449.

Wald, A. *Sequential Analysis*. Courier Corporation, 1973. ISBN 978-0-486-61579-0. Google-Books-ID: zXq-PAQAAQBAJ.