# Accelerating boosting using early stopping and weighted sampling

**Anonymous Authors**[1]

## Abstract

We present a new boosting trees algorithm that improves the use of the memory hierarchy over the current best two implementations: XGBoost and LightGBM.

We present two methods for accelerating boosting: early stopping, and weighted sampling. We describe these improvement and show their mathematical properties. We implemented our methods for boosting and evaluated them on the splice-site prediction problem (Sonnenburg & Franc, 2010; Agarwal et al., 2014). We compare our performance to that of XGBoost (Chen & Guestrin, 2016) and LightGBM (Ke et al., 2017). Our results show that our method is significantly faster when the training set is too large to fit in main memory.

## 1 Introduction

A major bottleneck for learning algorithms that use big data is the transfer of the training data up the memory hierarchy from disk, through main memory and the caches to the CPU cores. This probem becomes severe when the training data does not fit in main memory and examples have to be processed multiple times.

The leading implementations of boosted trees: XGBoost (Chen & Guestrin, 2016) and LightGBM (Ke et al., 2017) suffer from this problem. LightGBM will not run if memory is not sufficiently large to hold all of the training data. XGBoost will run in that case, but is very slow.

The main contribution of this paper is a boosting tree algorithm we call **Sparrow**, which runs much faster than XGBoost and LightGBM, especially when memory is too small to store all of the training data.

Our initial implementation, presented here, is for the special case of boosting "stumps", i.e. trees of depth one. Plans for extending the work to full-size trees are described at the end of the paper.

A trivial way to deal with the problem of limited memory is to read into memory as much data as would fit, and learn only from that data. The problem with that approach is increased over-fitting. One approach to this problem is presented in random-forests (), where trees are trained on separate subsets of the data and then combined by a majority vote. Random forests significantly reduce the variance of the trees, but they do not decrease the bias. Boosting can reduce the variance as well as the bias (**?**). We therefor want a memory-efficient version of boosting.

**Sparrow** is based on three ideas from statistics: *early stopping*, the *effective* number of examples and *weighted sampling*. Each of these ideas appears in the literature, but their combination into an algorithm is new. We now describe how these three ideas are combined.

At a high level, **Sparrow** works as follows. It repeatedly draws weighted samples from disk into memory. Each sample is used for multiple boosting iterations. As boosting progresses the weights of the examples become increasingly skewed. A skewed sample is effectively smaller. Examples with very small weight are essentially a waste of memory space. We quantify this by a quantity we call "the effective number of examples'. When the "the effective number of examples" decreases below a threshold **Sparrow** flushes the sample and selects a new sample. The new sample have uniform weights and therefor the effective sample size is equal to the size of the sample.

As for reading examples from memory into the CPU, *early stopping* is used to stop the reading of examples as soon as a good enough weak rule has been identified.

We implemented **Sparrow** using the Rust programming language. We compared it's performace to the performance of XGBoost and LightGBM on 50 Million examples (the human acceptor splice site dataset (**?**Agarwal et al., 2014)). We show that on a 16GB machine **Sparrow**is about 200 times faster than XGBoost. On 144GB machine all of the data can be loaded into memory and the performance is .....

The rest of the paper is organized as follows. In Section 3 we describe the statistical tools used in the design of **Spar-**

[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.

**row**. In Section 4 we describe **Sparrow**. In Section 5 we describe the experiments comparing **Sparrow** to XGBoost and LightGBM. We conclude in Section 6 with future work.

## 2 Related Work

The use of stopping rules to accelerate boosting has been stedied before by Domingo and Watanabe (Domingo & Watanabe, 2000) and by Bradley and Schapire (Bradley & Schapire, 2007).

Our work here is based on the original formulation of boosting (**????**). XGBoost and LightGBM are based on an alternative formulation of boosting propsed by Freedman Hastie and Tibshirani ().

## 3 Theory

We start with a brief description of the confidence-rated boosting algorithm (Algorithm 9.1 on the page 274 of (Schapire & Freund, 2012)).

Let $\vec{x} \in X$ be the feature vectors and let the output be $y \in Y = \{-1, +1\}$. The target is defined as a joint distribution $\mathcal{D}$ over $X \times Y$, our goal is to find a classifier $c : X \to Y$ with small error:

$$\text{err}_{\mathcal{D}}(c) \doteq P_{(\vec{x},y)\sim\mathcal{D}}\left[c(\vec{x}) \neq y\right]$$

We are given a set $\mathcal{H}$ of base classifiers $h : X \to [-1, +1]$. The Score function generated by Adaboost is a weighted sum of $T$ rules from $\mathcal{H}$

$$S_T(\vec{x}) = \left(\sum_{t=1}^{T} \alpha_t h_t(\vec{x})\right)$$

The strong classifier is the sign of the score function: $H_T = \text{sign}(S_T)$.

Adaboost is a gradient descent algorithm. It operates by iteratively decreasing the value of an exponential potential function (**??**) We start with the *true* potential, i.e. the potential in the limit where the size of the training set increases to infinity. We will then consider the realistic case, where the size of the training set is finite. The *true* potential of the score function $S_t$ is

$$\Phi(S_t) = E_{(\vec{x},y)\sim\mathcal{D}}\left[e^{-S_t(\vec{x})y}\right]$$

Consider adding a single base rule $h_t$ to the score function $S_{t-1}$: $S_t = S_{t-1} + \alpha_t h_t$ and taking the partial derivative of the potential with respect to $\alpha_t$ we get:

$$\left.\frac{\partial}{\partial \alpha_t}\right|_{\alpha_t=0} \Phi(S_{t-1} + \alpha_t h) = E_{(\vec{x},y)\sim\mathcal{D}_{t-1}}\left[h(\vec{x})y\right] \quad (1)$$

Where

$$\mathcal{D}_{t-1} = \frac{\mathcal{D}}{Z_{t-1}} \exp\left(-S_{t-1}(\vec{x})y\right) \quad (2)$$

and $Z_{t-1}$ is a normalization factor that makes $\mathcal{D}_{t-1}$ a distribution.

Adaboost performs coordinate-wise gradient descent where each coordinate corresponds to one base rule. Using equation 1 we can express the gradient with respect to base rule $h$ as a correlation, which we also call the *true edge*:

$$\gamma_t(h) \doteq \text{corr}_{\mathcal{D}_{t-1}}(h) \doteq E_{(\vec{x},y)\sim\mathcal{D}_{t-1}}\left[h(\vec{x})y\right] \quad (3)$$

The goal of a single boosting step is to find a base rule with significant edge. [1]

Up to this point, our discussion regarded the true expected value or an inifinitely large training set. Real training sets are finite, therefore **Sparrow** has to *estimate* the edge of $h$. We use the standard unbiased estimate

$$\hat{\gamma}_t(h) \doteq \widehat{\text{corr}}_{\mathcal{D}_{t-1}}(h) \doteq \sum_{i=1}^{n} \frac{w_i}{Z_{t-1}} h(\vec{x}_i) y_i \quad (4)$$

where $w_i = e^{-S_{t-1}(\vec{x}_i)}$ and $Z_{t-1} = \sum_{i=1}^{n} w_i$

The novelty of **Sparrow** is in the way it uses samples of the training data to to identify rules whose true edge is significant. Several statistical techniques are used to minimize the number of examples needed to compute the estimates.

### 3.1 Effective Sample Size

Equation **??** defines $\hat{\gamma}(h)$, which is an unbiased estimate of $\gamma(h)$. How accurate is this estimate? A standard quantifier is the variance of the estimator:

$$\text{Var}(\hat{\gamma}) = \frac{\sum_{i=1}^{n} w_i^2}{\left(\sum_{i=1}^{n} w_i\right)^2} \quad (5)$$

If all of the weights are equal $\text{Var}(\hat{\gamma}) = 1/n$ which corresponds to a standard deviation of $1/\sqrt{n}$ which is the expected relation between the sample size and the error.

If the weights are not equal then the variance is larger and the estimate is less accurate. We define the *effective sample size* $n_{\text{eff}}$ to be

$$n_{\text{eff}} \doteq \frac{\left(\sum_{i=1}^{n} w_i\right)^2}{\sum_{i=1}^{n} w_i^2} \quad (6)$$

So that $\text{Var}(\hat{\gamma}) = 1/n_{\text{eff}}$.

To see that the name “effective sample size” makes sense, consider $n$ weights where $w_1 = \cdots, w_k = 1/k$ and

[1] Some times the goal is stated as finding the rule with the *maximal* edge. Here we require only that $h_t$ has a significant edge.

$w_{k+1} = \cdots = w_n = 0$. It is easy to verify that in this case $n_{\text{eff}} = k$ which agrees with our intuition that examples with zero weight have no effect on the estimate.

**Sparrow** samples examples from disk to memory using *weighted sampling*. This is a sequential algrithm that reads from disk one example $(\vec{x}, y)$ at a time, calculates the weight for that example $w_i$ and the flips a biased coin to accept the example with probability $w_i/M$, where $M \geq \max w_i$. Accepted examples are stored in main memory with initial weight of 1.

Suppose $n$ initial memory-resident training examples consist of 99% negative and 1% positive examples. In that case the first base rule will be "predict negative everywhere". After reweighting the total weights of the positive and negative examples will be equal. This in turn means that the the effective size of about $n_{\text{eff}} \approx n/25$. In other words, the variance of estimates of the edge has increased by a factor of 25 relative to the initial uniform weights.

When **Sparrow** detects that $n_{\text{eff}}$ is small. It clears the memory and samples a new set from disk. However, as the sampling uses the weights as acceptance probabilities, half of the training data will be positive and half of them will be negative and $n_{\text{eff}}$ is back to $n$.

This process continues as long as Adaboost is making progress and the weights are becoming increasingly skewed. When the skew is large, $n_{\text{eff}}$ is small and **Sparrow** resamples a new sample with uniform weights.

The net effect is that even though the memory-resident samples are small, they contain the important examples, whose weight is large.

### 3.2 Sequential analysis

**Sparrow** achieves Disk-to-Memory efficiency by using weighted resampling that is triggered when $n_{\text{eff}}$ is too small.

**Sparrow** achieve Memory-to-CPU efficiency by reading from memory the minimal number of examples that are necessary to establish that a particular weak rule has a significant edge. This is done using Sequential Analysis and Early Stopping.

Sequential analysis (SA) was introduced by Wald (Wald, 1973) in the 1940s. Here we give a short illustration. Suppose we want to estimate the expected loss of a model. In the standard large deviation analysis we assume that the loss is bounded in some range, say $[-M, +M]$ and that the size of the training set is $n$. This implies that the standard deviation of the training loss is at most $M/\sqrt{n}$. In order to let this standard deviation be smaller than some $\epsilon > 0$ we need that $n > (M/\epsilon)^2$. While this analysis is optimal in the worst case, it can be improved if we have additional information about the standard deviation. We can glean such information from the observed losses by using the following sequential analysis method. Instead of choosing $n$ ahead of time, the algorithm computes the loss of one example at a time. A *stopping rule* is used to decide whether, conditioned on the sequence of losses seen so far, there is very small probability that the difference between the average loss and the true loss is larger than $\epsilon > 0$. The result is that when the standard deviation is significantly smaller than $M$ the number of examples that need to be used in the estimate is much smaller than $n = (M/\epsilon)^2$.

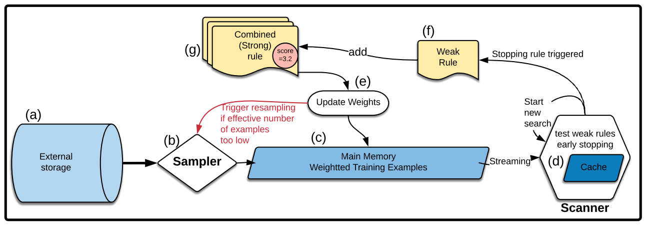

*Figure 1.* The **Sparrow** system architecture.

### 3.3 Sparrow's stopping rule

Using stopping rules for Adabost was proposed in (Domingo & Watanabe, 2000; Bradley & Schapire, 2007). Each use a different stopping rule. Unlike those works, we are interested in a rule that will take into account the fact that examples have different weights which impacts the the variance of the estimator. We want a stopping rule that is sensitive to $n_{\text{eff}}$.

Note that the choice of stopping rule has a direct impact on the performance of the algorithm. In other words, we want to use a stopping rule that is as tight as possible, including constants.

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.*

As we care about constants, we did a series of experiments to find the best constant $C$ which we use in **Sparrow**.

## 4 System Design and Algorithms

The main procedure of **Sparrow** is for identifing a rule that has a significant edge (Equation **??**). There are two subroutines that can execute in parallel in the process: a **Scanner (d)** and a **Sampler (b)**. We describe each subroutine in turn.

### Scanner

The task of a scanner (element **(d)** in Figure 1) 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), and a target edge $\gamma_t$. 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 $\gamma$. 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 weight of the added rule is calculated assuming that its edge is equal to $\gamma$.

The scanner falls into the *Failed* status if after exhausting all examples in the current sample set, no weak rule with an advantage larger than the threshold $\gamma$ is detected. When it happens, **Sparrow** shrinks the value of the target advantage threshold $\gamma$ and restart the scanner. In practice, **Sparrow** keeps track of the empirical edges $\hat{\gamma}(h)$ of all weak rules $h$. When the failure status happens, it reset the threshold $\gamma$ to be just below the value of the current maximum empirical edge of all weak rules (Figure 2).

### 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, **Sampler** prepares a *new* training set, in which all of the examples have equal weight, by using selective sampling. When the effective sample size associated with the old training set becomes too small, the scanner stops using the old training set and starts using the new one.[2]

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 an initial **weight** of 1. 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.

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

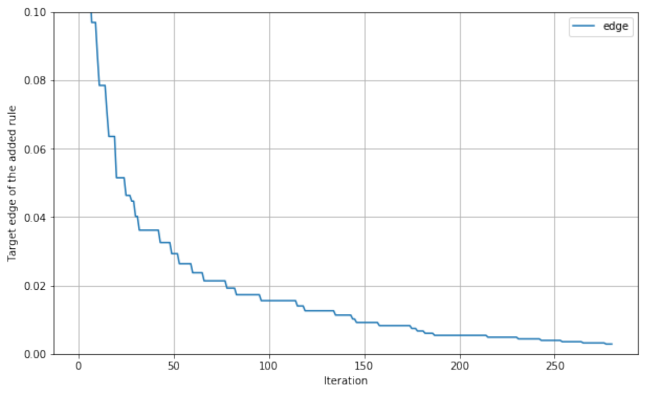

*Figure 2.* The edge of the rules being added to the strong rule (trained on the splice site dataset). New rules are being added to the ensemble with a weight calculated using the value of the threshold $\gamma$ at the time of their detection until no rule with an advantage over $\gamma$ can be detected. At that time **Sparrow** shrinks the value of the target edge $\gamma$, and restarts the scanner.

**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.

## 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.

| Memory | **Sparrow** | XGBoost | LightGBM |
|---|---|---|---|
| 16 GB | 18.5 (d) | 3224.7 (d) | (crash) |
| 32 GB | 17.9 (d) | 1705.8 (d) | (crash) |
| 72 GB | 21.9 (d) | 1497.7 (d) | 108.1 (m) |
| 144 GB | 19.6 (d) | 241.4 (m) | 109.4 (m) |
| # Rules | 189 | 400 | 400 |

*Table 1.* Comparison of the total training time on the splice site detection task (minutes) and the number of rules in the final ensemble when it converges. The (m) suffix denotes the package loads all training data to memory. The (d) suffix denotes the package does not load all training data to memory and uses disk as the external memory.

| Memory | **Sparrow** | XGBoost | LightGBM |
|---|---|---|---|
| 16 GB | 5.7 (d) | 483.7 (d) | (crash) |
| 32 GB | 5.6 (d) | 255.9 (d) | (crash) |
| 72 GB | 6.8 (d) | 224.7 (d) | 16.2 (d) |
| 144 GB | 6.6 (d) | 36.2 (m) | 16.4 (d) |

*Table 2.* Comparison of the per-tree training time on the splice site detection task (seconds). The (m) suffix denotes the package loads all training data to memory. The (d) suffix denotes the package does not load all training data to memory and uses disk as the external memory.

### 5.1 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.

In current implementation of **Sparrow**, 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 chose approximate greedy algorithm which is its fastest method. 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. We ran the evaluations on four different instance types with increasing memory capacities, specifically 16 GB (`c5d.2xlarge`), 32 GB (`c5d.2xlarge`), 72 GB (`c5d.9xlarge`), and 144 GB (`c5d.18xlarge`). The memory requirement of **Sparrow** is decided by the sample size, which is a configurable parameter. XGBoost supports external memory training when the memory is too small to fit the training dataset. Unlike XGBoost, LightGBM does not support external memory execution.

### 5.2 Evaluation

We summarize the experiments results in Table 5 by using the convergence time to an almost optimal loss of 0.061. Table 5.1 lists the per-tree training time of each packages in different memory settings. The in-memory version of XGBoost runs out of memory on the instances with the memory sizes of 16-72 GB. We trained the model using the external memory version on those instances. For LightGBM, it runs out of memory on the instances with 16 GB and 32 GB memory.

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. In the most limited memory setting (16 GB), XGBoost external memory is about 174x slower than **Sparrow** which also uses external memory. Even comparing to the in-memory versions of LightGBM and XGBoost, **Sparrow** is 6x and 13x faster, respectively.

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.

## 6 Future Work

Our preliminary results show that early stopping and selective sampling can dramatically speed up boosting algorithms on large real-world datasets.

The source code for sparrow is available for

We have several directions for future work.

First, **Sparrow** is currently limited boosting stumps and to binary classification. We plan to extend **Sparrow** to deep trees and to allow more than two labels. We will then run it on many more data sets.

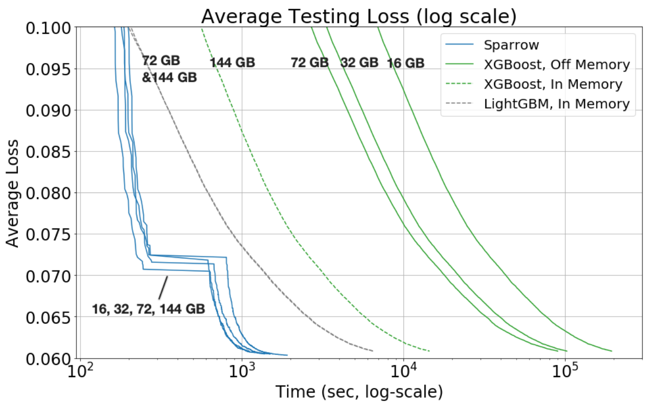

*Figure 3.* Comparing the average loss on the testing data using **Sparrow**, XGBoost, and LightGBM, lower is better. The period of time that the loss is constant for **Sparrow** is when the algorithm is generating a new sample set.

Second, our work shows that sampling from memory can take an inordinate amount of time when the training set is large and the weights are highly skewed. We have developed a stratified sampling algrithm that significantly reduces this problem.

Third, we are working on a parallelized version of **Sparrow** which uses a novel type of asynchronous communication protocol.

Fourth, our current implementation of **Sparrow** does not take advantage of multi-core machines as much as LightGBM does. We plan to address that problem.

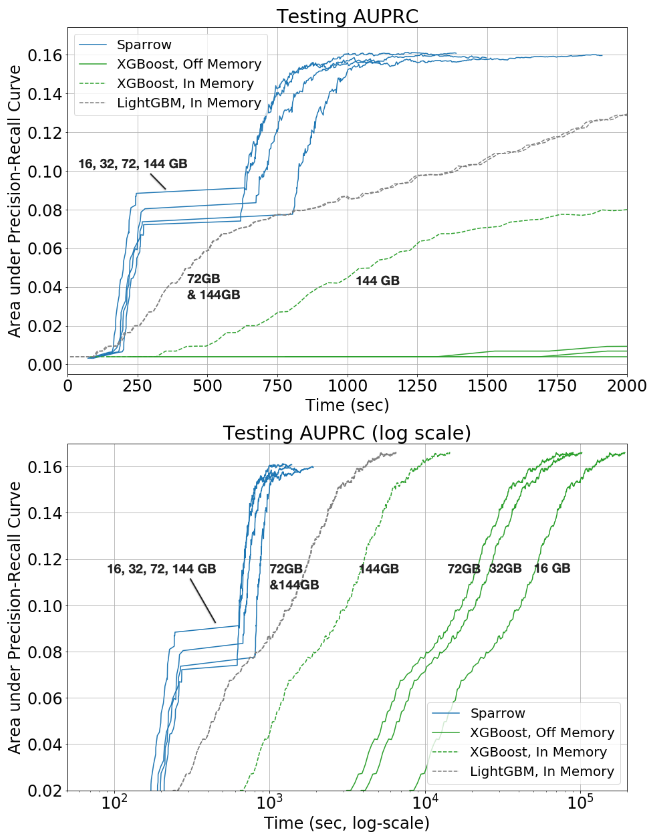

*Figure 4.* Comparing the area under the precision-recall curve (AUPRC) on the testing data using **Sparrow**, XGBoost, and LightGBM, higher is better. (left) Normal scale, clipped on right. (right) Log scale, clipped on left. The period of time that the AUPRC is constant for **Sparrow** is when the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

## References

Agarwal, A., Chapelle, O., Dudík, M., and Langford, J. A Reliable Effective Terascale Linear Learning System. *Journal of Machine Learning Research*, 15:1111–1133, 2014.

Balsubramani, A. Sharp Finite-Time Iterated-Logarithm Martingale Concentration. *arXiv:1405.2639 [cs, math, stat]*, May 2014.

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.

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.

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.

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.

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.

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

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.

Wald, A. *Sequential Analysis*. Courier Corporation, 1973. ISBN 978-0-486-61579-0.

## APPENDIX: PSEUDOCODE FOR Sparrow

**Algorithm 1** Main procedure of **Sparrow**

```
**Initialize** $H = 0, L = 0$
**Create initial sample** $S$ by calling SAMPLE
**for** $k := 1 \dots K$ **do**
    $Ret \leftarrow$ SCANNER$(\gamma_0, M, i, H, \mathcal{W})$
    **if** $Ret$ is *Fail* **then**
        Call SAMPLE to Get a New Sample.
    **else**
        $i', h, \gamma \leftarrow Ret$
        $i \leftarrow i'$
        $H \leftarrow H + \frac{1}{2} \log \frac{1/2+\gamma}{1/2-\gamma} h$
        Update $L$
```

**Algorithm 2** Procedure the Scanner

```
In-memory sampled set $S$ is defined globally

**function** SCANNER$(\gamma_0, M, i_0, H, \mathcal{W})$
    $\gamma \leftarrow \gamma_0, m \leftarrow 0, i \leftarrow i_0$
    $V \leftarrow 0, W \leftarrow 0$
    $\forall h \in \mathcal{W} : m[h] = 0$
    **while True do**
        $(x, y, w_s, w_l, H_l) \leftarrow S[i]$
        $i \leftarrow (i+1) \bmod |S|$
        **if** $i = i_0$ **then**
            **return** *Fail*
        $m \leftarrow m + 1$
        **if** $m > M$ **then**
            $\gamma \leftarrow \gamma/2, m \leftarrow 0$
        $w \leftarrow$ UPDATEWEIGHT(
                $x, y, w_l, H_l, w_s, H, i$)
        $V \leftarrow V + w^2, W \leftarrow W + |w|$
        **for all** $h \in \mathcal{W}$ **do**
            $m[h] \leftarrow m[h] + wyh(x)$
            $ret \leftarrow$ STOPPINGRULE(
                        $W, V, m[h], \gamma$)
            **if** $ret$ = **True then**
                **return** $i, h, \gamma$

**function** UPDATEWEIGHT$(x, y, w_l, H_l, w_s, H, i)$
    Calculate score update $s \leftarrow H(x) - H_l(x)$
    Calculate new weight $w \leftarrow w_l \exp(ys)$
    Update Sample: $S[i] = (x, y, w_s, w, H)$
    **return** $w/w_s$

**function** STOPPINGRULE$(W, V, m, \gamma)$
    $C, \delta$ are global parameters.
    $M \leftarrow |m - 2\gamma W|$
    **return** $M > C\sqrt{V(\log\log \frac{V}{M_0} + \log \frac{1}{\delta}}$
```

**Algorithm 3** Procedure of the Sampler

```
**function** SAMPLE()
    **Input:** Randomly permuted, disk-resident training-set.
    **Input** Current model $H$
    $S \leftarrow \{\}$
    **for all** available training data $(x, y)$ **do**
        $w_s \leftarrow \exp(-yH(x))$
        With the probability proportional to $w_s$,
            $S \leftarrow S + \{(x, y, w_s, w_s, H)\}$.
    **return** S
```