Accelerating boosting through early stopping and reweighting

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 machine learning from 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 problem is particularly hard when the training data does not fit in memory.

One solution to this problem, used in SGD, is to use a streaming algorithm which updates the model parameters after each example. Even though each update step is noisy, averaging using a small learning rate smoothes out the update trajectory.

Algorithms such as decision trees and boosting update the classifier only infrequently. But each update is very likely to reduce the true error of the classifier. In most implementations of these algorithms the whole training set is scanned at each iteration. This can take a long time. An alternative is to read into memory as many training examples as possible and then treat that set as the training set. This is much faster, but limiting the size of the training set can lead to overfitting.

In this paper we describe a third approach where memory is replenished with new samples regularly, reducing I/O load while keeping the algorithm from overfitting.

Two statistical methods are combined to achieve this result: sequential analysis and weighted sampling.

Using sequential to accelerate boosting has been suggested before by Domingo and Watanabe (Domingo & Watanabe, 2000) and by Bradley and Schapire (Bradley & Schapire, 2007).

There are two main ways of using the example weights in boosting:

- **Boosting by filtering**: Examples arrive in a stream and are selected with probability proportional to their weight. The weighted training error is calculated using an *unweighted average* over the selected examples.
- **Boosting by weighting**: Examples are stored in memory, along with their weight. The weighted training error is calculated by taking the weighted average over all of the examples.

The algorithm described in (Domingo & Watanabe, 2000; Bradley & Schapire, 2007) are both based on *boosting by filtering*. This means that only one example needs to be stored in memory at any time. That extremely small memory footprint comes at the cost of under-utilizing the memory and putting undue burden on disk-to-memory communication.

Our approach is to improve memory utilization by combining boosting by filtering and boosting by weighting, which we call **Sparrow**. It periodically samples batches on examples using boosting-by-filtering. It then performs several boosting-by-weighting iterations using the memory-resident batches.

The weights of the memory resident batch are initially uniform. These weights change into non-uniform weights as boosting-by-weighting progresses. This decreases the "effective size" of the batch. In Section **??** we show that a good measure of the effective size of a set of examples with weights $w_1, \dots, w_n$ is $(\sum_i w_i)^2/(\sum_i w_i^2)$. Having a small effective size implies a large error in the estimates of the weighted error. When the effective size falls below a set thresholds, the batch is discarded and a new batch is filtered

[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.

to replace it. The new batch will always have effective size n. By reusing the data in memory and sampling a new batch only when needed, **Sparrow** achieves significantly faster performance than other algorithm.

Additional speedup is achieve by using a stopping rule when running boosting on the memory-resident batch. This stopping rule is similar to the stopping rules used (Domingo & Watanabe, 2000; Bradley & Schapire, 2007) with the additional refinement that it takes in weights and the effective sample size into account. We describe this stopping rule in Section **??**.

The rest of the paper is organized as follows. In Section **??** we describe the mathematical tools used to design of **Sparrow**İn Section **??** we describe **Sparrow**İn Section **??** we describe the experiments comparing **Sparrow** to XGBoost and LightGBM. We conclude in Section **??** with some conclusions and future work.

2 Theory

The analysis given in this section applies to the original boosting algorithm (Freund & Schapire, 1997; Schapire & Singer, 1999; Freund & Mason, 1999) and to **Sparrow**. It does not apply to XGBoost (Chen & Guestrin, 2016) or to LightGBM (Ke et al., 2017) which are both implementations of Gradient-Boosted-Trees (Friedman, 2001), which does not lend itself to this type of analysis.

The inner loop of the original AdaBoost algorithm is a search for the weak rule with the smallest weighted error. This rule is found by computing the weighted error of each weak rule using the whole training set. When the training set is large, this scan can take a long time.

Sparrow reduces the required time by sampling subsets of the data into memory and by reading just enough data to find a good (rather than the best) weak rule. These techniques are based on sequential analysis and on selective sampling, which we now describe.

2.1 Sequential analysis

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 $n = (M/\epsilon)^2$.

2.1.1 *Sequential Analysis for Boosting*

(define $\mathcal{D}$, γ and $\hat{\gamma}$)

Boosting algorithms are iterative. At the iteration T, they generate a strong rule $H : X \to Y$ which is a weighted majority over T weak rules $h \in$. In this paper, we consider binary classification task. We can define the edge of a classifier h as

$$\gamma(h) \doteq E_{(x,y)\sim\mathcal{D}}\left[w(x,y)yh(x)\right] \qquad (1)$$

where $\mathcal{D}$ is the distribution over $X \times Y$, and $w(x,y) = \frac{1}{Z}\mathcal{D}(x,y)e^{-yH(x)}$.

A small but important observation is that AdaBoost does not require finding the weak rule with the **smallest** weighted error 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 over random guessing.

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 $\gamma(h)$.

Domingo and Watanabe (Domingo & Watanabe, 2000) and Bradley and Schapire (Bradley & Schapire, 2007) 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.

note that if a few rules are good and most are bad, this gives a significant reduction in sample size.

(Bradley & Schapire, 2007) and (Domingo & Watanabe, 2000) define use stopping rules that apply to equally weighted sampled data. For reasons that will be explained in the next section, we use a different We use a different stopping rule that will be explained in Section 2.3.

2.2 Effective Sample Size

Equation **??** 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 $\mathbf{Z}$-test for

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

As should be expected, the value of $\mathbf{Z}$ remains the same whether or not $\sum_{i=1}^n w_i = 1$. Based on Equation 2 we define the *effective number of examples* corresponding to the un-normalized weights $w_1, \ldots, w_n$ as:

$$n_{\text{eff}} \doteq \frac{\left(\sum_{i=1}^n w_i\right)^2}{\sum_{i=1}^n w_i^2} \tag{3}$$

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 omitted). 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, suppose 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 $n_{\text{eff}} = m$.

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

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

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 $(\frac{1}{n}\sum_{i=1}^n w_i)/(\max_i w_i)$.

2.3 Sparrow's stopping rule

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.

3 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 1)[2].

Our description of **Sparrow** is in two parts. First, we describe the design of a *single* **Sparrow** worker. Following

[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

that, we describe how concurrent workers use the **TMSN** protocol to update each other. Figure 1 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.

3.1 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 **??**). 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 1) 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 an initial **weight** of 1. [4]

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

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.

3.2 Communication between workers

Communication between the workers is based on the **TMSN** protocol. As explained in Section 3.1, 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.

4 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.

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

[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.

Setup We use a large dataset that was used in other studies of large scale learning on detecting human acceptor splice site (**?**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.

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 4.

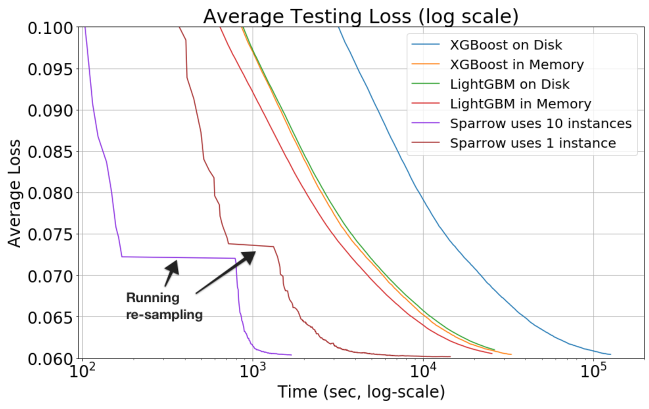

Figure 2. Comparing the average loss on the testing data using **Sparrow**, XGBoost, and LightGBM, lower is better. The period of time that the loss is constant for **Sparrow** is when the algorithm is generating a new sample set.

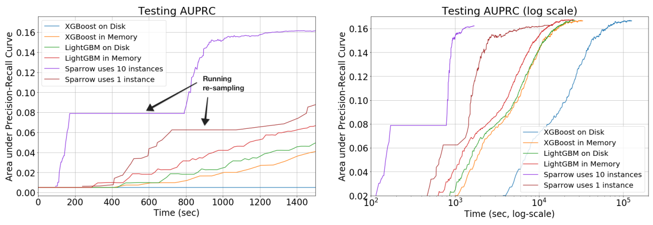

Figure 3. Comparing the area under the precision-recall curve (AUPRC) on the testing data using **Sparrow**, XGBoost, and LightGBM, higher is better. (left) Normal scale, clipped on right. (right) Log scale, clipped on left. The period of time that the AUPRC is constant for **Sparrow** is when the algorithm is generating a new sample set.

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 2. 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 4 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 3 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.

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., 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. 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.

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.

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.

Freund, Y. and Mason, L. The Alternating Decision Tree Learning Algorithm. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, pp. 124–133, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 978-1-55860-612-8.

Freund, Y. and Schapire, R. E. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. *Journal of Computer and System Sciences*, 55(1):119–139, August 1997. ISSN 0022-0000. doi: 10.1006/jcss.1997.1504.

Friedman, J. H. Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics*, 29 (5):1189–1232, 2001. ISSN 0090-5364. URL https://www.jstor.org/stable/2699986.

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.

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.

Owen, A. B. *Monte Carlo Theory, Methods and Examples*. 2013.

Schapire, R. E. and Singer, Y. Improved Boosting Algorithms Using Confidence-rated Predictions. *Machine Learning*, 37(3):297–336, December 1999. ISSN 1573-0565. doi: 10.1023/A:1007614523901.

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. Google-Books-ID: zXq-PAQAAQBAJ.