BoostVHT: Boosting Distributed Streaming Decision Trees

Theodore Vasiloudis
RISE SICS
tvas@sics.se

Foteini Beligianni
Royal Institute of Technology, KTH
fayb@kth.se

Gianmarco De Francisci Morales
Qatar Computing Research Institute
gdfm@acm.org

ABSTRACT

Online boosting improves the accuracy of classifiers for unbounded streams of data by chaining them into an ensemble. Due to its sequential nature, boosting has proven hard to parallelize, even more so in the online setting. This paper introduces BoostVHT, a technique to parallelize online boosting algorithms. Our proposal leverages a recently-developed model-parallel learning algorithm for streaming decision trees as a base learner. This design allows to neatly separate the model boosting from its training. As a result, BoostVHT provides a flexible learning framework which can employ any existing online boosting algorithm, while at the same time it can leverage the computing power of modern parallel and distributed cluster environments. We implement our technique on Apache SAMOA, an open-source platform for mining big data streams that can be run on several distributed execution engines, and demonstrate order of magnitude speedups compared to the state-of-the-art.

KEYWORDS

Online learning; Boosting; Decision Trees; Distributed Systems

1 INTRODUCTION

With the explosive growth of connected devices and mobile applications, massive amounts of streaming data are now available. Extracting knowledge from these massive data streams can generate substantial value for individuals and companies alike. For instance, a bank could monitor the transactions of its clients to detect frauds in real-time, so to be able to intervene.

Creating models by learning from massive streaming data is necessary to enable real-time predictions that can be used for decision-making. For this purpose, developing algorithms for continuous learning from large streams is of paramount importance. Having access to such algorithms allows us to apply learning techniques in domains where waiting for hours or even minutes for a batch model to be retrained is unacceptable, such as autonomous driving or real-time intrusion detection systems.

Learning in the context described above presents two major challenges. First, the sheer volume of data precludes processing and learning from billions of streaming data sources on a single machine. Nowadays, most data analytics pipelines employ a cluster of servers just to ingest the data, thus making learning in a distributed environment a compelling option. Second, the streaming nature of the data implies that the algorithm does not have access to the complete dataset at any point during the training. The presence of concept drift, where the function we are trying to learn may change during training, only exacerbates this problem.

Ideally, we would like a solution that is able to run on a cluster of servers, and at the same time is able to incorporate data in the model as it becomes available. In addition, the algorithm should be competitive with the state-of-the-art in terms of prediction accuracy. Meta-learning algorithms, or ensembles, where a group of weak learners are combined to provide highly accurate predictions, are a natural direction to explore. Some algorithms, such as bagging, are relatively easy to parallelize. However, one of the most successful meta-learning algorithms, *boosting*, is much more challenging, due to its sequential nature. In this paper, we provide a design for an online boosting algorithm that is able to run on modern distributed streaming engines.

The core idea of boosting is to train a chain of weak classifiers such that each successive one is trained on the mistakes made by its predecessors in the chain. As a result, the ensemble can learn more complex models than the weak learners it is composed of. Clearly, boosting is an inherently sequential algorithm, and as such has proven challenging to parallelize. While there exist a number of approaches that perform online boosting [1, 7, 16] or parallel boosting via approximations [11, 15, 17], designing an online *and* parallel boosting algorithm has received little attention in the literature. Our goal is to bring the accuracy of boosting to the distributed and online setting that is now common to many application domains.

In this work we present a new algorithm that combines two previous approaches that together tackle the issue of distributed online boosting. Previous distributed boosting approaches utilize data-parallel boosting, which commonly breaks the assumptions that most online boosting algorithms are based upon. Instead, our method leverages a recently introduced model-parallel online learning algorithm based on decision trees, the Vertical Hoeffding Tree (VHT) [14]. This algorithm allows us to employ parallel computation to speed up the learning process without breaking the order of the boosting stages. In addition, this design is compatible with any online boosting algorithm, thereby maintaining their theoretical accuracy guarantees.

Our proposal, BoostVHT, is implemented on top of Apache SAMOA,[1] an open-source platform for mining big data streams. As a result, we are able to deploy BoostVHT on top of several distributed stream processing engines supported by SAMOA. In

CIKM'17 , November 6–10, 2017, Singapore, Singapore

ACM ISBN 978-1-4503-4918-5/17/11…$15.00
https://doi.org/10.1145/3132847.3132974

[1] https://samoa.incubator.apache.org

particular, our experiments on Apache Storm show the almost-ideal scalability of BoostVHT, which is achieved without sacrificing prediction accuracy compared to sequential online boosting.

In short, our contributions are as follows:

- we present BoostVHT, a generalized method for online boosting which uses a model-parallel base learner, allows for parallel, online, highly-accurate classification, and is compatible with any specific online boosting algorithm;
- we provide an open-source implementation[2] of the algorithm in the Apache SAMOA online learning framework, which allows deploying the algorithm on top of several distributed streaming engines, including Apache Storm and Apache Flink;
- we report on an extensive set of experiments on the effectiveness of BoostVHT in terms of prediction accuracy, which show that our algorithm improves substantially over the base learner, and is able to match the performance of the sequential online boosting algorithm it is based on;
- we show that BoostVHT scales almost ideally to very large datasets, which would be prohibitive to handle on a single machine.

2 PRELIMINARIES

2.1 Boosting

Boosting refers to a general and provably effective method of producing a very accurate prediction rule by combining rough and moderately inaccurate learners. It is an ensemble method, where a group of "weak" learners are combined to create an arbitrarily accurate "strong" learning algorithm. It was originally proposed by Schapire [19], and later refined by Freund [12].

Possibly the most famous instance of Boosting is *AdaBoost* [13]. The algorithm takes as input a training set $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, where each instance x_i belongs to some domain X , and each label y_i is in some label set Y. The label domain Y can be binary, in the simpler case, or multi-class. AdaBoost works through a sequence of calls to a *base learning algorithm* in steps $t = 1, \ldots, s$ in a *chain*. The final hypothesis is simply a linear combination of all the hypotheses generated during the learning process

$$H(x) = \sum_{t=1}^{s} \alpha_t h_t(x),$$

where α_t is a weight that encodes the confidence we have in the corresponding hypothesis, and is derived from the training error.

The main idea of AdaBoost is to maintain a distribution over the training set $D_t(i)$ for each instance i. This distribution, or weight, captures the difficulty presented by each instance. Initially, all weights are set equally. At each step t, the distribution for the training set is adjusted by increasing the weights of incorrectly classified instances by the preceding learners up to $t - 1$. For each hypothesis $h_t, t \neq 1$, half of the weight is given to training instances which have been misclassified by the previous hypothesis h_{t-1}, and the other half to the rest. This procedure forces the weak learner to focus on the hard-to-classify instances in the training set.

[2]https://issues.apache.org/jira/browse/SAMOA-72

2.2 Online Boosting

Several adaptations of Boosting to the online setting have been proposed in the literature. OzaBoost [16] is the first one and the simplest, so we focus our explanation on it. However, our proposal easily applies to recently-proposed state-of-the-art algorithms such as AdaBoost.OL and Online BBM [1].

The core idea of OzaBoost is to mimic the weighting scheme of AdaBoost by using a Poisson distribution with a parameter λ. The weights for each instance and step $D_t(i)$ are drawn from this distribution. This parameter is increased when the instance is misclassified, and reduced otherwise.

2.3 Hoeffding Tree

A decision tree consists of a tree structure, where each internal node corresponds to a test on an attribute. The node splits into a branch for each attribute value (for discrete attributes), or a set of branches according to ranges of the value (for continuous attributes). Leaves contain classification predictors, usually majority class classifiers, i.e., each leaf predicts the class belonging to the majority of the instances that reach the leaf.

Decision tree models are very easy to interpret and visualize. The class predicted by a tree can be explained in terms of a sequence of tests on its attributes. Each attribute contributes to the final decision, and it's easy to understand the importance of each attribute.

The Hoeffding tree or VFDT is a very fast decision tree for streaming data [10]. Its main characteristic is that rather than reusing instances recursively down the tree, it uses them only once.

At the beginning of the learning phase, it creates a tree with only a single node. First, the algorithm sorts the instance into a leaf l (line 1). This leaf is a *learning leaf*, and the algorithm updates the sufficient statistic in l.

A single instance usually does not change the distribution significantly enough, therefore the algorithm tries to grow the tree only after a certain number of instances n_{min} has been sorted to the leaf. In addition, the algorithm does not grow the tree if all the instances that reached l belong to the same class.

To grow the tree, the algorithm iterates through each attribute and calculates the corresponding splitting criterion $\overline{G}_l(x_i)$, which is an information-theoretic function, such as entropy or information gain, The algorithm also computes the criterion for the scenario where no split takes places ($x_\emptyset$). Domingos and Hulten [10] refer to this inclusion of a no-split scenario with the term *pre-pruning*.

The algorithm then chooses the best (x_a) and the second best (x_b) attributes based on the criterion.By using these chosen attributes, the algorithm computes the Hoeffding bound ϵ to determine whether the leaf needs to be split or not.

If the best attribute is the no-split scenario ($x_\emptyset$), the algorithm does not perform any split. The algorithm also uses a tie-breaking τ mechanism to handle the case where the difference in splitting criterion between x_a and x_b is very small.

If the algorithm splits the node, it replaces the leaf l with an internal node. It also creates branches based on the best attribute that lead to newly created leaves and initializes these leaves.

2.4 SAMOA

Apache SAMOA[3] is an open-source distributed stream mining platform [8, 9]. It allows for easy implementation and deployment of distributed streaming machine learning algorithms on supported distributed stream processing engines (DSPEs) [21]. Additionally, it provides the ability to integrate new DSPEs into the framework and leverage their scalability to perform big data mining [2].

An algorithm in SAMOA is represented by a directed graph of operators that communicate via messages along streams which connect pairs of nodes. This graph is called a *Topology*. Each node in a Topology is a *Processor* that sends messages through a *Stream*. A Processor is a container for the code that implements the algorithm. At runtime, several parallel replicas of a Processor are instantiated by the framework. Replicas work in parallel, with each receiving and processing a portion of the input stream. These replicas can be instantiated on the same or different physical computing resources, according to the DSPE used. A Stream can have a single source but several destinations (akin to a pub-sub system).

A Processor receives *Content Events* via a Stream. Algorithm developers instantiate a Stream by associating it with exactly one source Processor. When the destination Processor wants to connect to a Stream, it needs to specify the *grouping* mechanism which determines how the Stream partitions and routes the transported Content Events. Currently there are three grouping mechanisms in SAMOA:

- *Shuffle grouping*, which routes the Content Events in a round-robin way among the corresponding Processor replicas. This grouping ensures that each Processor replica receives the same number of Content Events from the stream.
- *Key grouping*, which routes the Content Events based on their *key*, i.e., the Content Events with the same key are always routed by the Stream to the same Processor replica.
- *All grouping*, which replicates the Content Events and broadcasts them to all downstream Processor replicas.

2.5 VHT

The Vertical Hoeffding Tree (VHT) [14] is a model-parallel distributed version of the Hoeffding tree. Recall from Section 2.3 that there are two main parts to the Hoeffding tree algorithm: *sorting* the instances through the current model, and accumulating *statistics* of the stream at each leaf node. This separation offers a neat cut point to modularize the algorithm into two components. The first component is called *model aggregator*, and the second one *local statistics*. Figure 1 presents an illustration of the algorithm, specifically its components and of how the data flows between them.

The model aggregator maintains the current model (the tree). Its main duty is to receive the incoming instances and sort them to the correct leaf. If the instance is unlabeled, the model predicts the label at the leaf and sends it downstream (e.g., for evaluation). Otherwise, if the instance is labeled it is used as training data. The VHT decomposes the instance into its constituent attributes, attaches the class label to each, and sends them independently to the following stage, the *local statistics*. Algorithm 1 shows the pseudocode for the model aggregator.

[3]https://samoa.incubator.apache.org

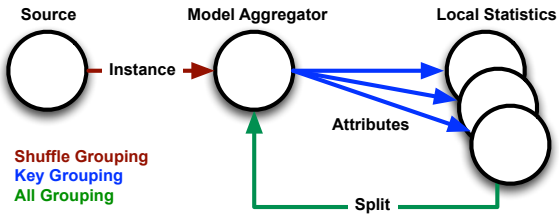


Figure 1: High level diagram of the VHT topology.

Algorithm 1: Model Aggregator

input : VHT, the current decision tree in the model aggregator; (x, y), a labeled training instance wrapped in $instance$ content event from source; $local\_result$, a local-result content event from the local statistics.

1 **if** *incoming content event is an* $instance$ **then**
2 Use VHT to sort x into a leaf l
3 Send $attribute$ content events to local statistics
4 Increment n_l, the number of instances seen at l
5 **if** $n_l \mod n_{min} = 0$ **and not** *all instances seen at l belong to the same class y* **then**
6 Add l into the list of splitting leaves
7 Send $compute$ content event with the ID of leaf l to all local statistics
8 **else** `// incoming local_result`
9 Get correct leaf l from the list of splitting leaves
10 Update x_a and x_b in l from with x_a^{local} and x_b^{local} from $local\_result$
11 **if** $local\_results$ *from all local statistics received* **or** *time out reached* **then**
12 Compute Hoeffding bound $\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n_l}}$
13 **if** $x_a \neq x_\emptyset$ **and** $(\overline{G}_l(x_a) - \overline{G}_l(x_b) > \epsilon$ **or** $\epsilon < \tau)$ **then**
14 Replace l with an internal node branching on x_a
15 **forall** *branches of the split on x_a* **do**
16 Add a new leaf with derived sufficient statistics from the split node
17 Send *drop* content event with ID of leaf l to all local statistics

The local statistics contain the sufficient statistics n_{ijk} for a set of attribute-value-class triples. Conceptually, the local statistics Processor can be viewed as a large distributed table, indexed by leaf ID (row), and attribute ID (column). The value of the cell represents a set of counters, one for each pair of attribute value and class. The local statistics simply accumulate statistics on the data sent to it by the model aggregator. SAMOA implements vertical parallelism by connecting the model to the statistics via key grouping. It uses a composite key made by the leaf ID and the attribute ID. Pseudocode for the local statistics is given in Algorithm 2.

Leaf splitting. Periodically, the model aggregator will try to see if the model needs to evolve by splitting a leaf. When a sufficient number of instances have been sorted through a leaf, it sends a broadcast message to the statistics, asking to compute the split criterion for the given leaf ID. The statistics get the table corresponding to the

Algorithm 2: Local Statistic

input : $attribute$, an *attribute* content event; $compute$, a *compute* content event; $local\_statistic$, the local statistics could be implemented as $Table < leaf\_id, attribute\_id >$

1 **if** *incoming content event is an* $attribute$ **then**
2 Update $local\_statistic$ with data in $attribute$: attribute value, class value and instance weights
3 **else if** *incoming content event is a* $compute$ **then**
4 Get ID of leaf l from $compute$ content event
5 For each attribute i of leaf l in local statistic, compute $\overline{G}_l(x_i)$
6 Find x_a^{local}, the attribute with the highest $\overline{G}_l$
7 Find x_b^{local}, the attribute with the second highest $\overline{G}_l$
8 Send x_a^{local} and x_b^{local} using $local\_result$ content event to model aggregator

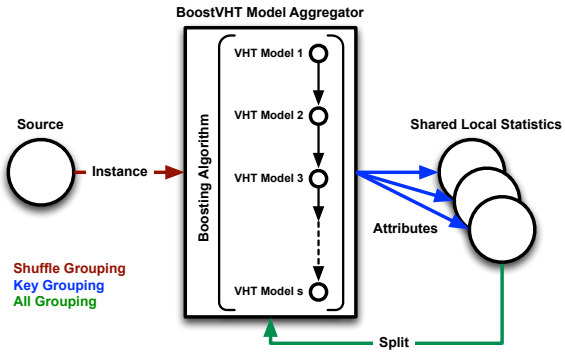


Figure 2: Topology of BoostVHT.

leaf, and for each attribute compute the splitting criterion in parallel (e.g., information gain or entropy). Each local statistics Processor then sends back to the model the top two attributes according to the chosen criterion, together with their scores. The model aggregator simply needs to compute the overall top two attributes, apply the Hoeffding bound, and see whether the leaf needs to be split.

Two cases can arise: the leaf needs splitting, or it does not. In the latter case, the algorithm simply continues without taking any action. In the former case instead, the model modifies the tree by splitting the leaf on the selected attribute, and generating one new leaf for each possible value of the branch. Then, it broadcasts a `drop` message containing the former leaf ID to the local statistics. This message is needed to release the resources held by the leaf and make space for the newly created leaves. Subsequently, the tree can resume sorting instances to the new leaves. The local statistics will create a new table for the new leaves lazily, whenever they first receive a previously unseen leaf ID. In its simplest version, while the tree adjustment is performed, the algorithm drops the new incoming instances.

3 ALGORITHM

There are two specific drawbacks in the current state-of-the-art boosting algorithms that we aim to address with our proposal. On the one hand, current online boosting algorithms are unable to take advantage of modern parallel and distributed architectures. On the other hand, most parallel boosting approaches make concessions in terms of the correctness of the algorithm.

One of the main challenges in parallel and online boosting is normalizing the weights of the samples in such way that they form "smooth" probability distributions. Online boosting algorithms such as OzaBoost approximate these weights so that the algorithm will converge to the offline version of AdaBoost asymptotically. To do so, they change the weight of each instance as it moves down the boosting chain, thus requiring a strict sequential boosting order. This requirement makes data-parallel approaches extremely expensive, as each stage of the boosting chain would require network communication. Conversely, data-parallel boosting algorithms train multiple examples in parallel. After having iterated through the complete training set, they adjust their weights to form the desired distributions. Clearly, this latter technique cannot be applied in an online setting where examples arrive sequentially. In short, the assumptions made by the online boosting algorithms break the ones made by the parallel boosting algorithms, and vice-versa.

To solve this conundrum, we take a different approach that allows us to maintain the theoretical guarantees of the online boosting algorithms, while at the same time taking advantage of the parallelism available in modern computer clusters. We borrow the same approach used by VHT: rather than trying to parallelize the training in terms of instances, we instead parallelize over the attributes of each instance, thus resulting in a model-parallel algorithm. To do so, we modify the design of the VHT algorithm to allow for efficient training of boosted models.

We call our approach BoostVHT. Our boosting chain consists of a sequence of VHT models, each of which is trained in parallel and asynchronously. As an instance passes through the chain of models, the order of the boosters is maintained, thus guaranteeing the sequential training assumptions of the online boosting algorithms.

Figure 2 presents a visualization of the algorithm's topology, its components, and the connections between them. The main difference from VHT is in the model aggregator. At a high level, the BoostVHT model aggregator consists of two independent components: the specific boosting algorithm, and the chain of VHT models, i.e., the ensemble. The first component is modular, i.e., we can employ the logic of any existing online boosting algorithm. In Section 4 we report results obtained with OzaBoost [16], however we have also implemented AdaBoost.OL [1], OSBoost [7], and SAMME [22]. Algorithm 3 shows the pseudocode for BoostVHT model aggregator implemented with OzaBoost.

The second component is tightly linked to the specific choice of weak learner, VHT. The design of VHT neatly decomposes the prediction phase from the learning phase. The former is sequential, and happens by sorting an instance through the tree at the model aggregator, which is local to a single Processor. The latter is parallel and distributed, and happens asynchronously at the local statistics by leveraging the cluster.

Recall that the distribution of weight instances in online boosting depends on the (prequential) prediction error of each member of the ensemble. Given that in VHT this error can be computed locally, the whole ensemble chain can be kept locally inside the BoostVHT model aggregator. The main advantage of this design is that no communication is required to forward an instance to the next stage

Algorithm 3: BoostVHT Model Aggregator($\mathbf{h}$, VHT_t, (x, y))

init : $\lambda_t^c \leftarrow \lambda_t^w \leftarrow 0, \quad \forall t \in [1, s]$ `// cumulative weight of instances with correct and wrong predictions`

input : $\mathbf{h}$, the ensemble, a set of s hypotheses h_t; VHT_t, VHT model t; (x, y), a labeled training instance.

output : prediction $\widehat{y}$.

`// prequential evaluation: first test...`

1 $\widehat{y} = \arg\max_{\dot{y} \in Y} \sum_{t=1}^{s} \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right) I\left(h_t(x) = \dot{y}\right)$

`// ...then train`

2 $\lambda \leftarrow 1$

3 **foreach** $h_t \in \mathbf{h}$ **do** `// in order` $t \in [1, s]$

4 k $\leftarrow$ Poisson(λ)

5 **if** $k > 0$ **then** `// give weight` k `to the instance`

6 $h_t \leftarrow VHT_t(h_t, (x, y))$

7 **if** $y = h_t(x)$ **then** `// correct prediction`

8 $\lambda_t^c \leftarrow \lambda_t^c + \lambda$

9 $\epsilon_t \leftarrow \frac{\lambda_t^w}{\lambda_t^c + \lambda_t^w}$

10 $\lambda \leftarrow \lambda\left(\frac{1}{2(1-\epsilon_t)}\right)$

11 **else** `// wrong prediction`

12 $\lambda_t^w \leftarrow \lambda_t^w + \lambda$

13 $\epsilon_t \leftarrow \frac{\lambda_t^w}{\lambda_t^c + \lambda_t^w}$

14 $\lambda \leftarrow \lambda\left(\frac{1}{2\epsilon_t}\right)$

15 **return** $\widehat{y}$

of the boosting chain. From the point of view of the online boosting algorithm, the whole process happens locally, and thus does not require any modification to the logic of the boosting algorithm. However, the training of the ensemble members happens in parallel on the cluster.

This design choice gives us two distinct advantages. First, it decouples the achievable parallelism from the number of VHT models (boosting stages) being used. Second, it allows for communication-efficient training. In a data-parallel boosting algorithm such as POCA [18], the maximum amount of achievable parallelism is limited to the number of boosting stages used. In many cases, this number can be smaller than the number of cores available in a modern data center. In contrast, our design allows to use as many boosting stages as the user sees fit. The parallelism is only limited by the dimensionality of the data.

Finally, with the model being local to one Processor, no communication is required to make predictions. This choice does however create a trade-off between communication and prediction speed. As the model is local, the predictions are computed sequentially by a single Processor. Since our technique is aimed at a distributed environment we chose to avoid additional communication steps as it is usually more expensive than computation in a distributed setting.

3.1 Optimizations

We now describe two optimizations that reduce the computational, memory, and communication costs of BoostVHT.

Shared Local Statistics. A naïve implementation of BoostVHT would use a set of local statistics for each VHT member in the ensemble. However, the local statistics Processors can be shared among all the trees. To enable this sharing, the only requirement is that the leaf IDs across all the trees are unique. This property ensures that there is no collision among statistics of different trees. Given that the trees are all kept locally in the BoostVHT model aggregator, ensuring unique IDs is straightforward.

This optimization makes it possible to fine-tune the level of parallelism independently from the number of members of the ensemble. That is, it fully separates non-functional concerns (the parallelism), which affect the speed of the algorithm, from the functional ones (number of trees), which affect its accuracy. In addition, consolidating the local statistics reduces the overhead due to running a large number of Processors, both in terms of memory and computation.

Aggregated Messages. For each VHT model in the chain, the training of the algorithm is performed in parallel and asynchronously. In order to achieve efficient distribution of the attributes, we collocate the statistics of a specific range of attributes for all VHT models in a specific parallel replica of the local statistics. This structure can be thought of as a distributed hash table, where each attribute range is given a unique ID that is common across different replicas of the VHT models, and all the statistics for the same attribute range hash to the same parallel replica of the local statistics Processor.

The way this design choice achieves communication efficiency is twofold. Trough the collocation, we bound the communication needed for each example to p (the chosen parallelism level) messages per instance. Whereas, if each individual attribute were to be hashed to a local statistics Processor, m (the number of attributes) messages per instance would be needed. In typical application scenarios we expect $m \gg p$. In addition, BoostVHT only needs to communicate split messages from the local statistics to the model aggregator. In contrast, parallel boosting algorithms such as AdaBoost.PL [17] send the complete models, and thus have a much larger communication overhead.

Finally, the fact that the local statistics are shared between the VHT models in the chain allows us to only send each attribute slice event only once to each local statistics Processor, and re-use it for every VHT in the chain. After the first VHT has sent the attribute data, the remaining VHT models in the boosting chain can send just the adjusted weight. Compared to the naïve approach of each VHT sending attribute messages individually, we send only p attribute slice messages instead of $s \times p$ for each instance, where p is the number of local statistics Processors and s the number of boosting stages. When we consider that each attribute message can contain hundreds or thousands of attributes (m/p), this optimization can produce significant communication savings.

4 EXPERIMENTS

This section presents an experimental evaluation of our proposed approach. We first compare our algorithm against two baselines in terms of prediction accuracy and running time. Then we focus on the scalability properties of BoostVHT.

4.1 Experimental Setup

We run the single-threaded and parallel experiments on a server with 16 cores 128 GiB of main memory. For the distributed experiments of Section 4.7 we use a cluster of 8 virtual machines in a cloud environment, each with 8 vCPUs and 32 GiB of memory. We implement BoostVHT by using SAMOA v0.5.0, and use MOA v2016.10 for the baseline. We run the parallel and distributed experiments on top of Apache Storm v0.9.4, and use 12 executors (Storm's processing slots). For each dataset and parameter setting, we run the experiments five times and report the average measure. Unless otherwise specified, we use an ensemble size of $s = 10$.

4.2 Data

We use both synthetic and real datasets to test different aspects of our approach.

Synthetic Datasets. For the experiments we use three different data generators: a hyperplane generator, a random tree generator for dense instances, and a tweet generator for sparse instances. Each generated dataset has 1M instances and represents a binary classification problem.

- **Random tree generator:** generates instances using a tree-like structure for the attributes and a binary class. We create three datasets with different numbers of attributes: 20, 200, and 2000. In the results, we refer to these datasets as *rtg_attributeNum*.
- **Text generator:** generates random tweets, simulating a sentiment analysis task. We create four sparse datasets with different number of attributes: 50, 100, 150, and 500. TextGenerator attributes represent the appearance of words in tweets from a predefined bag-of-words, and produces tweets with an average length of 15 words. Each word is drawn from the bag from a Zipf distribution with skew parameter $z = 1.5$. We refer to these datasets as *textGen_attributeNum*.
- **Hyperplane**: generates a problem of predicting the class of points separated by a rotating hyperplane. We create three different dense datasets with 50, 100, and 150 attributes, respectively. We refer to these datasets as *hyper_attributeNum*.

Real Datasets.

- *CovertypeNorm* is a multi-class dataset with 581 012 instances and 54 attributes, which describes the forest cover type for 30x30 meter cells collected by the US Forest Service. It is commonly used in online learning due the concept drift present in the class distribution.
- *elecNormNew* is a binary classification problem with 45 312 instances and 8 attributes. It records electricity prices collected from the Australian New South Wales Electricity Market, and also contains concept drift as the prices fluctuate according to the supply and demand in the market.
- *Airlines*. This dataset is an adaptation of a regression dataset to a binary classification problem with 539 383 instances and 7 attributes. The task is to predict whether a flight will be delayed given information about its scheduled departure.
- *Diabetes* is a dataset used during 1994 AAAI Spring Symposium on Artificial Intelligence in Medicine, and has since been used in numerous machine learning studies. It is a binary classification problem with 768 instances and 8 attributes, and the task is to predict whether a patient is diabetic based on a number of measurements taken from automatic devices at regular intervals and paper records.

4.3 Baselines

- **MOA:** We use the implementation of OzaBoost available in MOA [4] as our baseline, using a sequential Hoeffding Tree as a base learner. This implementation is single-threaded and provides an indication of the achievable accuracy when using the OzaBoost algorithm. Ideally, the accuracy of our algorithm should match the one of this baseline, while outperforming it in terms of running time.
- **VHT:** We use the non-boosted version of the VHT algorithm to show how the boosting employed by BoostVHT improves the prediction accuracy over the base learner.

We also implemented POCA in SAMOA but were not able to achieve accuracy comparable to the other methods with it, or obtain the original code from the authors, so we omit the comparisons.

4.4 Metrics

We use the following metrics to compare the performance in terms of running time and accuracy:

- *Speedup* of execution over MOA OzaBoost, defined as:

$$Speedup = \frac{ExecutionTime_{MOA}}{ExecutionTime_{SAMOA}}$$

- *Kappa Statistic* is a robust classification accuracy metric that takes into consideration the probability of agreement by chance, indicating an improvement over a majority class classifier [3].

We use a prequential evaluation strategy for our accuracy measurements: for each instance in the dataset, we first make a prediction with the current model, update our metric based on the prediction error, and only then reveal the instance label and use it to train the model. When showing the evolution of the accuracy over time we report the results using a sliding window containing 1000 instances.

4.5 Accuracy

In this section we evaluate the accuracy of BoostVHT compared to OzaBoost in MOA, and the VHT algorithm which we use as a base learner. This evaluation can be considered a sanity check to confirm that the boosting algorithm works as intended, and that the approximation introduced by VHT to facilitate parallel training does not affect the accuracy adversely.

The VHT algorithm is itself a strong learning algorithm, however there exist datasets where it can struggle with accuracy, for example in high-dimensional problems. In those cases BoostVHT can offer improved accuracy, as we can see in Figure 3 for the text generator datasets. As we increase the dimensionality of the data, the performance of VHT degrades, while BoostVHT remains unaffected, and matches the performance of MOA OzaBoost. Figure 4 presents another example, this time for the Hyperplane datasets, where VHT is consistently less accurate than BoostVHT, which again is able to match OzaBoost as implemented in MOA.

Table 1: Kappa statistic (percentage) measuring the prediction accuracy of OzaBoost in MOA and BoostVHT when using the Local and Storm execution engines.

Dataset	MOA	BoostVHT	
		Local	Storm
rtg_20	43.63	43.25	43.14
rtg_200	32.21	31.20	29.73
rtg_2000	N/A	52.10	52.25
textGen_50	87.08	87.01	86.82
textGen_100	87.03	87.15	86.49
textGen_150	88.70	88.46	87.87
textGen_500	89.54	89.33	88.63
hyper_50	46.60	47.29	46.72
hyper_100	40.02	40.63	39.85
hyper_150	36.42	36.43	36.21
covertypeNorm	44.30	45.00	39.20
elecNormNew	50.91	51.61	47.90
airlines	18.13	17.62	17.73
diabetes	2.72	1.91	6.21
Average	**51.3**	**51.3**	**50.6**

To provide an overall comparison of BoostVHT and MOA OzaBoost, we list the accuracy of the algorithm over the different datasets in Table 1. We report the accuracy achieved by running the algorithm locally, and in parallel on the Storm execution engine. It is clear that BoostVHT is able to closely match the accuracy achieved by MOA OzaBoost, both when using the sequential and the parallel execution engine. The performance is stable on the synthetic datasets as well as on the real ones. We note that MOA did not complete the random tree generator experiment with 2000 attributes after 12 hours, so we stopped the experiment.

4.6 Speedup

In the previous section we saw that BoostVHT does indeed boost the accuracy of a single VHT learner, and is able to closely follow the accuracy of the single-threaded OzaBoost with Hoeffding trees. In this section we show that BoostVHT can be orders of magnitude faster than MOA OzaBoost, while maintaining high accuracy.

The speedup results are summarized in Table 2. Even the local version of BoostVHT is able to dramatically outperform MOA OzaBoost, with speedups reaching two orders of magnitude for some datasets. Note that the speedup achieved when using the Storm execution engine is smaller than the local version. While this result may seem counter-intuitive, the choice of execution engine can have adverse effects in the performance of the algorithm. The local version does not perform any serialization and deserialization of messages as they pass through the various SAMOA Processors, granting it a major performance benefit. Additionally, Storm includes mechanisms for fault-tolerance, delivery acknowledgments, and worker coordination through the master. All these mechanisms create significant overhead that lead to the degradation in performance compared to the local execution engine. However, by using Storm we are able to scale-out the algorithm to much larger data sizes, as we show in Section 4.7.

Table 2: Average Speedup of BoostVHT over OzaBoost in MOA.

Dataset	Local	Storm
rtg_20	39.3	4.3
rtg_200	49.3	33.1
textGen_50	75.5	11.8
textGen_100	84.7	20.1
textGen_150	100.4	30
textGen_500	91	68.6
hyper_50	16.6	4.6
hyper_100	17.8	7.8
hyper_150	18.3	11
covertypeNorm	18.6	2.2
elecNormNew	2.7	0.1
airlines	116.3	7
diabetes	1	0.4
Average	**45.2**	**14.4**

4.7 Scalability

To examine the scalability of BoostVHT, we run experiments in a parallel and a distributed setting. We assess its *strong scaling* characteristics, i.e., the speedup achieved by increasing the computing resources while keeping the problem size fixed, and its *weak scaling* characteristics, where we increase the computing resources while increasing the size of the problem by the same factor.

For strong scaling, we wish to achieve *linear speedup*: growing the computational resources by a factor of C should lead to a reduction of the training time by the same factor of C. For weak scaling, we wish to achieve *linear scale-out*: growing both the computing resources and the size of the problem by a factor of C should not affect the training time adversely. We measure the mean time to train 1000 instances generated by the text generator, and report the average time over 30 000 instances, in milliseconds. For all the experiments we use the same ensemble size of 10, and vary the number of local statistics Processors.[4]

4.8 Weak Scaling

For the weak scaling experiments, we double the number of attributes along with the computing resources for each run. For the parallel experiments we vary the number of attributes from 500 to 4000, with the number of local statistics Processors ranging from 2 to 16. The results are reported in Figure 5. For the parallel implementation the training time per 1000 instances remains relatively stable until scale 4 (4000 attributes with 16 Processors). The decrease in performance can be attributed to the fact that the server only has 16 cores available, which are occupied by the 16 local statistics Processors. In this case, other SAMOA Processors, such

[4]In this context, Processor refers to SAMOA *Processors* (see Section 2.4), and we use the term *cores* when referring to CPUs.

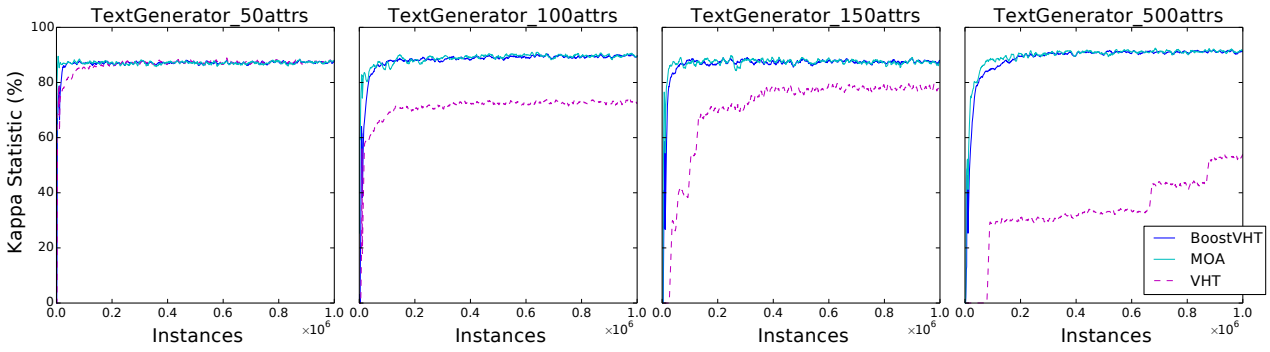


Figure 3: Kappa statistic (accuracy) as a function of arriving instances over time for text generator datasets with an increasing number of attributes.

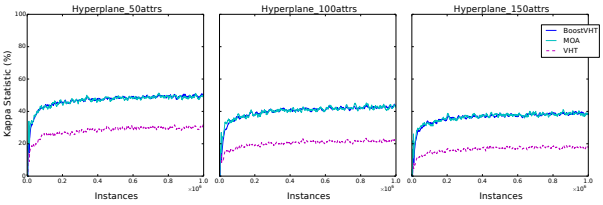


Figure 4: Kappa statistic (accuracy) as a function of arriving instances over time for Hyperplane datasets with an increasing number of attributes.

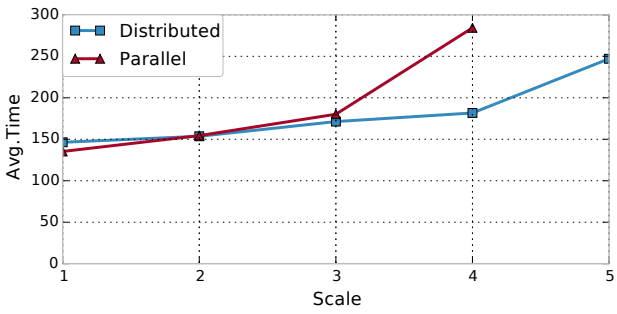


Figure 5: Weak scaling experiments, time in milliseconds. Scale 1x on the x-axis refers to 500 attributes with 2 Processors, and we double both Processors and attributes for each scale increment (up to 8,000 attributes with 32 Processors).

as the input and model aggregator, and the Storm daemons, have to share computing resources, thus causing scheduling delays.

For the distributed experiments we vary the number of attributes from 500 to 8000, and the number of local statistics Processors from 2 to 32. As can be seen in the blue rectangle line in Figure 5, the training time per 1000 instances is more stable than in the parallel experiments. Especially for scale 4 (4000 attributes and 16 Processors), given that the cluster has additional processing slots, no SAMOA Processor has to share resources with another. We do observe an increase in training time at scale 5 however, which can be attributed to the increased scheduling and communication costs when using 4 workers instead of 2.

Overall, both the distributed and parallel experiments indicate good weak scaling characteristics for the algorithm, with the training time not being adversely affected by the increase in scale.

4.9 Strong Scaling

To test the strong scaling of the algorithm, we use the text generator to create a stream of instances with 1000 attributes, and vary the number of local statistics Processors used. We present the results in Figure 6a for parallel execution and in Figure 6b for distributed execution. For the parallel execution we run five experiments varying the number of local statistics Processors from 1 to 16. The algorithm achieves near-linear speedup as we increase the available computing resources. Similarly to what we observed previously, at larger scales the scaling deviates slightly from the ideal linear speedup, because SAMOA Processors have to share cores.

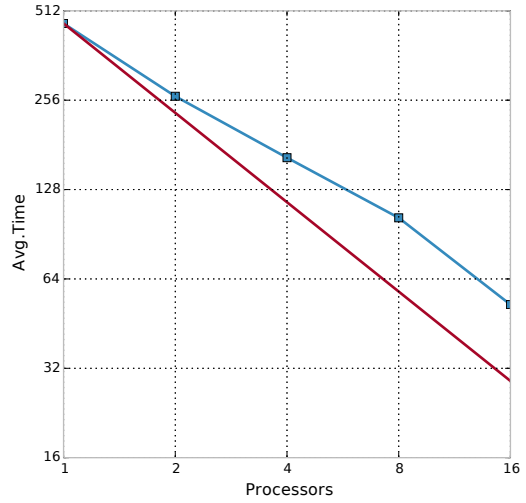
In the distributed setting we perform seven experiments, varying the number of local statistics Processors from 1 to 64. Again we observe near-linear speed-up, which seems to taper off at 32 Processors, due to the increased coordination cost.

Overall, both experiments show an almost ideal strong scaling for BoostVHT.

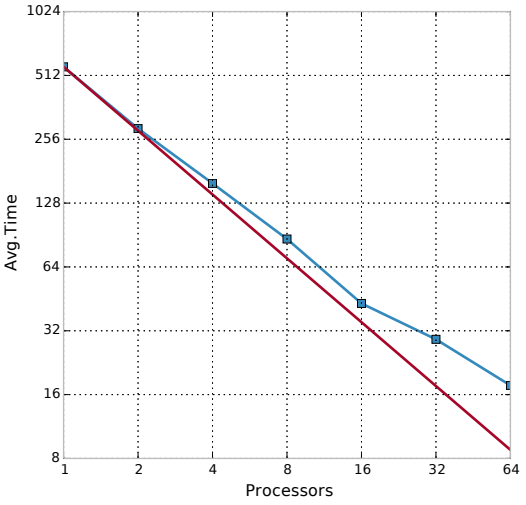
5 RELATED WORK

Due to its excellent performance, boosting [12, 19] has been studied extensively, both to investigate its theoretical aspects and to create extensions to the original framework. This section provides an overview of related work, focusing mostly on online and parallel extensions to boosting.

Online boosting. One of the first online boosting algorithms was proposed by Oza and Russell [16]. The algorithm, called OzaBoost, aims to approximate the behavior of the batch AdaBoost [13] algorithm, whereby the misclassified instances from the preceding model are given half the training weight for the following model, and the correctly classified ones the remaining half. To allow this re-weighting process to happen online, instances weights are drawn



(a) Parallel experiments.



(b) Distributed experiments

Figure 6: Strong scaling in the parallel (top) and distributed (bottom) setting. The time reported is the average time to train 1,000 instances, each with 1,000 attributes, in milliseconds.

from a Poisson distribution whose λ parameter is decreased or increased according to whether the instances were classified correctly or incorrectly respectively.

The first theoretically grounded approach was presented by Chen et al. [7]. The paper re-examines the base assumption that the weak learners perform better than random, and narrows it down to apply only when the sum of the instance weights is large enough. The result is an adaptation of the SmoothBoost algorithm [20] to the online setting. The authors also propose a new weighting scheme for combining the weak learner decisions, and provide theoretical guarantees on the error rate of the algorithm.

Beygelzimer et al. [1] improved upon the results of Chen et al. by relaxing the weak learner assumptions. They describe an algorithm that is optimal in terms of error rate, and another parameter-free adaptive algorithm that out-performs previous state-of-the-art algorithms. The authors present a new definition of weak online learning, and use it to develop an online extension to the boost-by-majority algorithm [12] that is optimal in the number of weak learners and has near-optimal sample complexity. They then use the online loss minimization framework to develop their adaptive algorithm, which uses a logistic loss function and online gradient descent to determine the instance and weak learner weights.

Parallel and distributed boosting. Due to the inherently sequential nature of boosting, parallelizing the algorithm is not trivial. One of the first parallel and distributed versions of the algorithm was proposed by Fan et al. [11]. In their algorithm, classifiers are trained either on random samples of the complete dataset, or on disjoint partitions, named r (random) and d (disjoint) sampling respectively. In r-sampling a fixed number of instances is randomly picked from the training dataset at each boosting round. After training the weak classifier, the instance weights are re-adjusted according to the success of the weak learner before sampling again and beginning the next boosting round. In d-sampling a classifier is learned over each partition of the data. Again, the weights of the instances are re-adjusted according to their difficulty between each boosting round. This work also proposed a non-parallel but incremental version of AdaBoost, which feeds the classifiers small partitions of the complete data, and keeps a history of the k most recent classifiers, which are re-used to train an additional classifier on the new data.

This process is similar to the one proposed by Breiman [5], who describe two algorithms aimed at datasets that cannot fit into memory, and include an online variation. The proposed algorithms, Rvote and Ivote take small pieces of the complete dataset, train classifiers on them, and paste together their predictions for the final decision. Rvote selects data randomly via bootstrap sampling, while Ivote uses importance weighting, preferring instances that are classified incorrectly. In the online version, events that are misclassified are always accepted for training, while correctly classified ones are accepted in proportion to the current out-of-bag error. A bounded history of k classifiers is used again to limit the memory use of the algorithm. While these algorithms can deal with large datasets due to their incremental nature, they don't make use of the parallel and distributed capabilities of modern systems.

Lazarevic and Obradovic [15] proposed algorithms for parallel and distributed boosting. In the parallel setting, k weak learners are trained in parallel for each boosting round $t \in [1, s]$, each trained on a disjoint subset of the training data. After each weak learner is trained, they compete for each data point, and the best performing learner is selected as the weak learner for the current boosting round, t. The final predictions are made by combining the predictions of the best classifier at each boosting round. The goal of this approach is to decrease the number of boosting iterations needed to achieve the highest accuracy, compared to a the sequential case. The distributed algorithm builds separate learners at each data

partition, and combines them at the end of each boosting round. This combination uses a local instance weight distribution at each worker which, when concatenated, should give the same distribution as AdaBoost. This approach has the disadvantage that the learners for each partition are prone to overfit to their local dataset. In addition, the synchronization at the end of each round can incur large communication costs.

Chawla et al. [6] described distributed versions of the sequential Ivote and Rvote algorithms. The algorithm builds classifiers incrementally on "bites" of the data, similarly to what Breiman [5] proposed. However, it does so on each partition of the data independently. The learning stops either when the out-of-bag error plateaus, or after a predefined number of iterations. The final predictions are made by combining the individual predictions via majority voting. Unlike the proposal by Lazarevic and Obradovic [15], this algorithm requires no communication between processors, but it is again prone to overfitting each data partition.

Most of the aforementioned approaches use parallelization and distribution schemes to scale up the possible data sizes that can be tackled by a boosting algorithm, but do not provide a parallel speed-up. More recently, Palit and Reddy [17] tackle this challenge with a distributed boosting algorithm which relies on the MapReduce programming paradigm. Their algorithm, AdaBoost.PL, runs independent versions of a boosting algorithm on each of p workers, and sorts the k local weak learners according to their weight. It then aggregates the models in a final ensemble by merging together the local weak learners with matching performance levels, i.e., those which have the same rank when locally sorted by weight. The final classifier is a combination of the $p \times k$ weak learners learned.

To the best of our knowledge, the only work on online *and* parallel boosting was presented by Reichler et al. [18]. The proposed algorithm, POCA (Parallel Online Continuous Arcing), maintains a number of weak learners which are linked in a "virtual chain". Each instance is delivered to all learners and used to train them. The chain structure ensures that the $k-1$ preceding learners influence the learning rate of weak learner k, by propagating their errors down the chain and increasing the weight of instances that are misclassified as they move down the chain. The necessary weight re-normalization is achieved by keeping track of the highest weight each weak learner has observed recently, and using it to normalize the newly generated weights. This procedure ensures that very small weights do not slow down the learning process. While the reported performance of POCA in terms of accuracy is similar to that of traditional boosting algorithms, the paper never presents a parallel speedup, noting that "on a single CPU it can run orders of magnitude slower" than traditional boosting algorithms, and its parallelism is limited to the ensemble size.

6 CONCLUSIONS

In this paper we presented a novel technique that combines state-of-the-art online boosting algorithms with model-parallel decision trees to achieve fast and accurate training over unbounded streams.

We showed that the proposed technique is able to achieve accuracy comparable to single-threaded versions of the algorithm, while at the same time achieving order of magnitude speed-ups over a state of the art online learning framework. The approach is scalable and can be used in a parallel and distributed setting, while the open-source implementation allows us to execute it over many different distributed engines. The technique is also algorithm-agnostic, allowing any online boosting algorithm to be used, without breaking its assumptions and maintaining its theoretical guarantees.

One immediate research thread we aim to investigate is overcoming the limitation of maintaining a single predictive model, which leads to a linear relationship between the number of features and the prediction time. By having a replicated model we should be able to parallelize prediction as we do training currently, at the cost of increased communication. In the future we aim to investigate efficient data-parallel online boosting algorithms, that trade-off accuracy for runtime performance, and ultimately combine the two approaches to create truly scalable, model-and-data parallel online boosting algorithms.

REFERENCES

[1] Alina Beygelzimer, Satyen Kale, and Haipeng Luo. 2015. Optimal and Adaptive Algorithms for Online Boosting. In *ICML*, Vol. 37. 2323–2331.

[2] Albert Bifet and Gianmarco De Francisci Morales. 2014. Big Data Stream Learning with SAMOA. In *ICDM*. 1199–1202.

[3] Albert Bifet, Gianmarco De Francisci Morales, Jesse Read, Geoff Holmes, and Bernhard Pfahringer. 2015. Efficient Online Evaluation of Big Data Stream Classifiers. In *KDD*. 59–68.

[4] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. 2010. MOA: Massive online analysis. *JMLR* 11 (2010), 1601–1604.

[5] Leo Breiman. 1999. Pasting Small Votes for Classification in Large Databases and On-Line. *Machine Learning* 36, 1-2 (1999), 85–103.

[6] Nitesh V. Chawla, Lawrence O. Hall, Kevin W. Bowyer, and W. Philip Kegelmeyer. 2004. Learning Ensembles from Bites: A Scalable and Accurate Approach. *JMLR* 5 (2004), 421–451.

[7] Shang-Tse Chen, Hsuan-Tien Lin, and Chi-Jen Lu. 2012. An Online Boosting Algorithm with Theoretical Justifications. In *ICML*. 1873–1880.

[8] Gianmarco De Francisci Morales. 2013. SAMOA: A Platform for Mining Big Data Streams. In *RAMSS Workshop @WWW'13*. 777–778.

[9] Gianmarco De Francisci Morales and Albert Bifet. 2015. SAMOA: Scalable Advanced Massive Online Analysis. *JMLR* 16 (2015), 149–153.

[10] Pedro Domingos and Geoff Hulten. 2000. Mining high-speed data streams. In *KDD*. 71–80.

[11] Wei Fan, Salvatore J. Stolfo, and Junxin Zhang. 1999. The Application of AdaBoost for Distributed, Scalable and On-line Learning. In *KDD*. 362–366.

[12] Yoav Freund. 1995. Boosting a Weak Learning Algorithm by Majority. *Information and Computation* 121, 2 (1995), 256–285.

[13] Yoav Freund and Robert E Schapire. 1995. A desicion-theoretic generalization of on-line learning and an application to boosting. In *EuroCOLT*. 23–37.

[14] Nicolas Kourtellis, Gianmarco De Francisci Morales, Albert Bifet, and Arinto Murdopo. 2016. VHT: Vertical Hoeffding Tree. In *BigData*. 915–922.

[15] Aleksandar Lazarevic and Zoran Obradovic. 2002. Boosting Algorithms for Parallel and Distributed Learning. *Distributed and Parallel Databases* 11, 2 (2002), 203–229.

[16] Nikunj C. Oza and Stuart Russell. 2001. Online Bagging and Boosting. In *Artificial Intelligence and Statistics*. 105–112.

[17] Indranil Palit and Chandan K. Reddy. 2012. Scalable and Parallel Boosting with MapReduce. *TKDE* 24, 10 (2012), 1904–1916.

[18] Jesse A. Reichler, Harlan D. Harris, and Michael A. Savchenko. 2004. Online Parallel Boosting. In *AAAI*. 366–371.

[19] Robert E Schapire. 1990. The strength of weak learnability. *Machine learning* 5, 2 (1990), 197–227.

[20] Rocco A. Servedio. 2003. Smooth Boosting and Learning with Malicious Noise. *JMLR* 4 (2003), 633–648.

[21] Anh Thu Vu, Gianmarco De Francisci Morales, João Gama, and Albert Bifet. 2014. Distributed Adaptive Model Rules for Mining Big Data Streams. In *BigData*. 345–353.

[22] Ji Zhu, Hui Zou, Saharon Rosset, and Trevor Hastie. 2009. Multi-class AdaBoost. *Statistics and its Interface* 2, 3 (2009), 349–360.