Kevin Deforth, Marc Desgroseilliers, Nicolas Gama, Mariya Georgieva, Dimitar Jetchev, and Marius Vuille

`XORBoost`: Tree Boosting in the Multiparty Computation Setting

Abstract: We present a novel protocol `XORBoost` for both training gradient boosted tree models and for using these models for inference in the multiparty computation (MPC) setting. Our protocol supports training for generically split datasets (vertical and horizontal splitting, or combination of those) while keeping all the information about features, thresholds, and evaluation paths private; only tree depth and the number of the binary trees are public parameters of the model. By using novel optimization techniques that reduce the number of oblivious permutation evaluations as well as sorting operations, we further speedup the algorithm. The protocol is agnostic to the underlying MPC framework or implementation.

Keywords: Privacy-preserving machine learning, multiparty computation, xgboosting

DOI 10.56553/popets-2022-0099
Received 2022-02-28; revised 2022-06-15; accepted 2022-06-16.

1 Introduction

Gradient boosting is a machine learning technique for regression and classification problems that yields a prediction model in the form of an ensemble of weak prediction models, typically decision trees [22]. `XGBoost` [2, 11], is currently one of the most popular open-source libraries supporting gradient boosting for various programming environments and architectures.

Kevin Deforth: Inpher, Switzerland, E-mail: kevin.deforth@inpher.io
Marc Desgroseilliers: Inpher, Switzerland, E-mail: marc@inpher.io
Nicolas Gama: Inpher, Switzerland, E-mail: nicolas@inpher.io
Mariya Georgieva: Inpher, Switzerland, E-mail: mariya@inpher.io
Dimitar Jetchev: Inpher, Switzerland, E-mail: dimitar@inpher.io
Marius Vuille: Inpher, Switzerland, E-mail: marius@inpher.io

Multiparty computation (MPC) is a method for cryptographic computing allowing several parties holding private data to evaluate a public function on their aggregate data while revealing only the output of the function and nothing else. Recent advances in the area make these protocols practical and suitable for real-world applications such as machine and statistical learning [6, 7, 9, 16, 19, 23–25, 32, 33, 36].

1.1 Our contributions

1.1.1 Setting and threat model

We consider a data distribution setup where input data comes from two or more private data sources. The data is either horizontally split among the owners (i.e., every owner has different samples/rows sharing the same features/columns), vertically split (every owner has complete set of features for all the samples/rows) or any combination of the two. The goal is to train and evaluate a gradient boosted tree model based on [11] in the MPC setting using the fully stacked dataset and without revealing private information. We will often assume that the data owners play the role of the compute parties (or players) that train and evaluate the boosting model, though this is not strictly necessary (i.e., our framework will support a scenario where private data owners secret share data among a set of compute parties that is not necessarily the same as the set of data owners). The compute parties operate in the semi-honest security model (players execute the exact steps of the algorithm) with full-threshold security (if all players except one decide to collude, the data of the non-colluding player is still protected).

1.1.2 Contributions

We present a novel protocol, `XORBoost`, for gradient boosted tree model training and prediction in the multiparty computation (MPC) setting. The computing parties in our protocol only learn the shape of the model, i.e., the tree depth and the number of trees and noth-

Table 1. Prior work most closely related to XORBoost. (Clas.:Classification, Reg.:Regression, Num.:Numerical, Cat.:Categorical, Vert.:Vertical, Horz.:Horizontal).

	Task		Data		Split			
Reference	**Clas.**	**Reg.**	**Num.**	**Cat.**	**Vert.**	**Horz.**	**Leakage**	**Learning Algorithm**
[29]	○	○	○		○		1st/2nd order stats	Federated XGBoost
[12]	○	○	○		○		1st/2nd order stats and all instance vectors	Federated XGBoost
[30]	○	○	○			○	leaf weights, thresholds	Federated XGboost
[18]	○			○	○	○	tree depth	ID3
[3]	○	○	○	○	○	○	tree depth	C4.5
[4]	○		○	○	○	○	tree depth/number of trees	Random Forest [ID3]
This work	○	○	○	○	○	○	tree depth/number of trees	XGBoost

ing else about the private input data. The feature indices and the threshold values associated to the non-leaf nodes as well as the weights (or prediction values) associated to the leaf nodes are all secret-shared throughout the computation. We also ensure that during training, no information about the relevant first- and second-order partial derivatives is revealed. Finally, the path taken by any sample in a decision tree remains secret.

The improvements presented in this paper allow to efficiently train a gradient boosted tree model of moderate depth (see Section 8). These include the use of a very efficient sorting algorithms (such as the oblivious quicksort from [9]), the precomputation of generator vectors to apply inverse permutations (Section 3.2), the use of compressed instance vectors (Section 4.2) and the storage of permuted instance vectors to reduce the number of times the permutation function is called. The overall number of oblivious permutations needed after an initial oblivious bucketing (or preprocessing) phase is $(2+D)k$ per tree where D is the tree depth and k the number of features.

The proposed algorithm is agnostic to the choice of the underlying MPC method or framework. We used the `Manticore` MPC framework for several reasons: 1) it provides access to boolean arithmetic as well as arithmetic with real numbers represented using modular integers [9] or the prior floating-point numbers framework [7]; 2) instantiating with `Manticore`'s representation of real numbers via modular integers, our protocol is information-theoretically secure; 3) we made use of `Manticore`'s oblivious sorting and oblivious permutations functionality. However, all MPC libraries that support Boolean and real number arithmetic such as `SCALE-MAMBA` [1], `SecureML` [33], `ABY` [32, 34], `SPDZ-2K` [15] can be used. In the `Manticore` framework, the computations are split into an offline (providing random precomputed data without yet accessing the private data) and an online phase (the actual computation with the private data). The offline phase can be performed interactively by the same computing parties that run the online phase using techniques such as oblivious transfer similarly to the methods proposed in [8, 15, 33] or by an independent party (different from the computing parties) called trusted dealer. In the former case, the MPC protocol is slower but the security model is stronger. In the latter case, the offline phase is significantly accelerated but in order to protect the privacy of the input data, the trusted dealer must not collude with any of the compute parties. In addition, to ensure security against malicious external adversaries, all communications between the trusted dealer as well as all communication between the players during the online phase is end-to-end encrypted.

On a training dataset of 25,000 samples and 300 features in the 2-player setting, we train a model consisting of 10 regression trees of depth 4, using histograms of 128 buckets, in less than 1.5 minutes per tree (total end-to-end compute time). In our benchmarks, we split the data-independent offline phase (which differs from one security model to the next) and the data-dependent phase (that is similar across most MPC protocols). We only report the latter.

1.2 Related work

Various attempts have been made to adapt classical boosting methods to privacy-preserving settings, both for training and inference [3, 4, 12, 18, 21, 26, 27, 29–31] (see also [10] for a recent survey of the literature).

In [12, 21, 29] frameworks based on federated learning and homomorphic encryption are proposed that allow for training a boosting model on vertically split datasets, that is, datasets where for each feature, the entire data for that feature (or feature column) belongs to a single party[1]. While this is suitable for applications where external private features can be added to enhance the model performance, it does not cover horizontally split data, that is, cases where private datasets with the same features can be concatenated to build a larger dataset (a classical federated learning/edge computing scenario). In comparison, data distribution in our `XORBoost` framework is as general as it can be (horizontally split, vertically split or a combination of the two), thus avoiding such limitations.

The framework proposed in [29] computes splits using secret sharing but avoids computing gains explicitly as it requires expensive private division operations. The security implication, compared to `XORBoost`, is that players learn information about the first- and second-order partial derivatives of the intermediate loss functions (what we call the 1st and 2nd order statistics), thus, potentially leaking information about the underlying training data.

A different method for training that does not require vertical splitting is proposed in [30]. The protocol is based on federated learning, secret sharing and homomorphic encryption and, in contrast to `XORBoost`, for every internal node, the feature and threshold leading to the maximum reduction of the loss function are revealed. This can potentially leak sensitive information about the original input training data. Even more information about the underlying input data and the trained model is revealed in [12]. For more regarding the training of tree-based ensemble models in the federated learning setting, see [20, 28, 38–40].

Recent work on `XGBoost` inference via additive Somewhat Homomorphic Encryption (SHE) is studied in [31].

Solutions using secure enclaves are proposed in [26, 27]. The security model is significantly different from the one considered here: our approach is based on information-theoretic security as opposed to hardware security. Even if measures are taken to obfuscate memory access and thus limit side channel attacks these approaches remain vulnerable to attacks targeting the secure enclave.

Finally, there have been multiple efforts to perform tree-based learning in the MPC setting. The protocol [3] allows for training and evaluating classification trees on data split horizontally or vertically (or any mixture thereof) and is based on the `MP-SPDZ` framework [23]. The major observation of [3] is that the output of the chosen tree learning algorithm (C4.5 trees) only depends on the relative order of the input; this allows to map the input data to the integer domain in an arbitrary but order-preserving manner. This skillfully avoids costly privacy-preserving operations on fixed- or floating-point numbers. The method of [18] trains a different decision tree algorithm (ID3) using a stopping mechanism that does not reveal information. The subsequent work of [4] builds on top of this and implements the bagging technique (in the MPC setting) leading to `Random Forest` model. These works use different learning algorithms, leading to fundamentally different approaches and tradeoffs in the implementations. `XORBoost` is based on XGBoost [11] and is inherently different from `CART` or `Random Forest`. Whereas `Random Forest` trains many trees independently on subsampled datasets, `XGBoost` iteratively adds trees to the ensemble in order to maximize the loss reduction. Furthermore, our protocol leverages fixed-point arithmetic. This allows to compute prediction weights accurately and hence, train regression trees instead of being limited to classification trees with categorical response variables.

To facilitate the comparisons with prior art, we refer to Table 1 where we outline the closest related works to `XORBoost` together with their relevant properties, features and leakage.

2 Background and preliminaries

For a detailed review of `XGBoost`, we refer the reader to [11]. Consider a dataset X of size $N \times k$ and a response variable y (a vector of size N). We use $X^{(j)}$ to refer to column j of X and X_i to refer to the ith row of X. Thus, $X_i^{(j)}$ denotes the ith element of the jth column.

2.1 Binary decision trees

A binary decision tree of depth D on the feature space $\mathbb{R}^k$ consists of $2^D - 1$ inner nodes (referred to as non-leaf nodes or split nodes) and 2^D outer nodes (referred to as leaf nodes). We denote by $\mathcal{N}$ the set of nodes. Associated to each inner node n is a pair $(j_n, \mathtt{t}_n)$ of a feature index $j_n \in \{1, \ldots, k\}$ and a threshold $\mathtt{t}_n$ (a real number). Associated to each leaf node ℓ is a weight

1 The implementation of [12] has been extended to allow for horizontally split datasets as well. However, we have not found any accompanying publications or preprints on `Homo SecureBoost` [37].

value $\mathtt{w}_\ell$ (a real number). We thus represent a tree as

$$\mathtt{Tree} = (\mathtt{TreeStructure},\ \mathtt{TreeWeights}),$$

where

$$\mathtt{TreeStructure} = ((j_1, \mathtt{t}_1), \dots, (j_{2^D-1}, \mathtt{t}_{2^D-1}))$$

is the list of pairs associated to the (list) of inner nodes and

$$\mathtt{TreeWeights} = (\mathtt{w}_1, \dots, \mathtt{w}_{2^D})$$

is the list of weights. It is further assumed that the TreeStructure is split into D layers at depth $0, 1, \dots, D-1$ and of sizes $2^0, 2^1, \dots, 2^{D-1}$ respectively, denoted by $L_0, \dots, L_{D-1}$. Thus, the nodes in the layer at depth d are indexed (from left to right) by $\{2^d, \dots, 2^{d+1}-1\}$. We denote by L_D the layer containing the leaf nodes. The following recursive procedure evaluates the subtree rooted at a given node n on a given sample $x = (x_1, \dots, x_k)$:

```
eval($x, n$) =
    if $n$ is a leaf of weight $\mathtt{w}$ : return $\mathtt{w}$
    else $n$ is an inner node $(j, \mathtt{t})$ :
        if $x_j < \mathtt{t}$ return eval($x, n_{\mathtt{left}}$)
        else return eval($x, n_{\mathtt{right}}$),
```

where $n_{\mathtt{left}}$ and $n_{\mathtt{right}}$ denote the left, respectively the right child of n.

We also define $\mathtt{eval}(x, \mathtt{Tree})$ to be the eval procedure called on x and the root node of Tree. In a prediction scenario when the tree is fixed and the sample varies, we often abbreviate the notation as $\mathtt{Tree}(x)$ seen as a piecewise constant function $\mathtt{Tree}\colon \mathbb{R}^k \to \mathbb{R}$, and on training scenario where x is fixed and the tree varies, we use $\mathtt{eval}_x(\mathtt{Tree})$, which, for a fixed TreeStructure, is continuously differentiable over the TreeWeights.

If there are many samples, we write $\mathtt{Tree}(X) \in \mathbb{R}^N$, to mean Tree evaluated at each row X_i of X. We let $\hat{y}$ be the estimate for the response variable y. Given a tree ensemble $\{\mathtt{Tree}^{(1)}, \dots, \mathtt{Tree}^{(T)}\}$ and a learning rate parameter η, one defines the predictions on X recursively as

$$\widehat{y}^{(t)} = \widehat{y}^{(t-1)} + \eta\, \mathtt{Tree}^{(t)}(X) \in \mathbb{R}^N. \tag{1}$$

The reason for the learning rate η is to dampen the contribution of the new tree added to the current model. Often, one takes $\eta = 1$ to obtain the total prediction on X as

$$\widehat{y}^{(T)} = \sum_{t=1}^{T} \mathtt{Tree}^{(t)}(X) \in \mathbb{R}^N. \tag{2}$$

2.2 Objective function

Gradient tree boosting is an iterative process using a current prediction $\widehat{y}^{(T)}$ on T trees to greedily (see [22] for a definition) grow a new $(T+1)$th tree that most reduces a certain objective function. For a given function $\mathtt{loss}\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ (e.g., mean-squared error or logistic loss) and a fixed training set (X, y), consider the function

$$\mathcal{L}_{\widehat{y}^{(T)}}(\mathtt{Tree}^{(T+1)}) = \sum_{i=1}^{N} \mathtt{loss}(y_i, \widehat{y_i}^{(T)} + \mathtt{eval}_{X_i}(\mathtt{Tree}^{(T+1)})) + \mathtt{Reg}(\mathtt{Tree}^{(T+1)}). \tag{3}$$

We add the superscript $\widehat{y}^{(T)} = (\widehat{y}_1^{(T)}, \dots, \widehat{y}_N^{(T)})$ to indicate the dependency on the predictions of the model at time t - note that these will be the hyper-parameters for the optimization problem that calculates the new tree $\mathtt{Tree}^{(T+1)}$ at time $T+1$.

Here, a regularization function Reg is used to reduce overfitting by penalizing large parameter values (similarly to Ridge and Lasso regression models). We use L_2-regularization on the leaf weights

$$\mathtt{Reg}(\mathtt{Tree}^{(t)}) = \gamma |L| + \frac{\lambda}{2} \sum_{\ell \in L} \mathtt{w}_\ell^2,$$

where λ and γ are fixed hyperparameters, L is the set of leaves of $\mathtt{Tree}^{(t)}$ and $\mathtt{w}_\ell$ is the leaf weight associated to the leaf $\ell \in L$.

The goal is to perform a greedy optimization, that is, given the ensemble $\{\mathtt{Tree}^{(1)}, \dots, \mathtt{Tree}^{(T)}\}$ and the corresponding vector $\widehat{y}^{(T)}$ of predictions of the ensemble on the training data at time T, find a tree $\mathtt{Tree}^{(T+1)}$ that minimizes the objective function $\mathcal{L}_{\widehat{y}^{(T)}}$. Note that for a fixed TreeStructure the restriction of the objective function $\mathcal{L}_{\widehat{y}^{(T)}}$ on the TreeWeights space is differentiable, convex in the case of logistic loss, and even quadratic in the case of mean-square loss. The basic idea of the greedy XGBoost algorithm is to recursively take a tree (initially a single leaf), replace one of its leaves by a fixed number of splits, and for each of these potential tree structures, retain the one that shows the maximal reduction for $\mathcal{L}$, and repeat the process until the tree is a full binary decision tree of depth D.

Since TreeStructure has both discrete parameters (the feature indices) and continuous parameters (the thresholds), one can discretize the latter by preprocessing/bucketing, to obtain a finite search space for the tree structure at each step of the above recursive splitting procedure. For a fixed tree structure, we define the *score function* to measure the reduction of the loss function for a given set of tree weights. Under the assumption that $\mathcal{L}_{\widehat{y}^{(T)}}$ is equal or well-approximated by its second-order expansion at zero, we define the score function as

$$\begin{aligned}&\texttt{score}(\texttt{TreeStructure}) = \\ &= \frac{1}{2}\texttt{grad}(\mathcal{L}_{\widehat{y}^{(T)}})^t \cdot \texttt{Hess}^{-1}(\mathcal{L}_{\widehat{y}^{(T)}}) \cdot \texttt{grad}(\mathcal{L}_{\widehat{y}^{(T)}}), \quad (4)\end{aligned}$$

with gradient and hessian over the TreeWeights space, all evaluated at zero.

Remark. *To justify why the score defined in* (4) *is the relevant one, consider a (differentiable) real-valued function $f(x_1, \ldots, x_n)$ on n variables and a fixed point $x^{(0)} := (x_1^{(0)}, \ldots, x_n^{(0)}) \in \mathbb{R}^n$. Letting δ be a vector in a small neighborhood of 0, the second-order approximation of $f(x^{(0)} + \delta)$ around $x^{(0)}$ is given by*

$$f(x^{(0)}+\delta) \sim f(x^{(0)}) + \texttt{grad}(f)^t|_{x^{(0)}} \cdot \delta + \frac{1}{2}\delta^t \cdot \texttt{Hess}(f)|_{x^{(0)}} \cdot \delta.$$

The value of $\delta \in \mathbb{R}^n$ that minimizes the above approximation is

$$\delta_{\min} = -\texttt{Hess}(f)^{-1}|_{x^{(0)}} \cdot \texttt{grad}(f)|_{x^{(0)}},$$

and

$$\begin{aligned}&f(x^{(0)} + \delta_{\min}) = f(x^{(0)}) - \\ &\quad -\frac{1}{2}\texttt{grad}(f)|_{x^{(0)}}^t \cdot \texttt{Hess}(f)|_{x^{(0)}}^{-1} \cdot \texttt{grad}(f)|_{x^{(0)}}.\end{aligned}$$

Applying this to $f(\cdot) = \mathcal{L}_{\widehat{y}^{(T)}}(\texttt{TreeStructure}, \cdot)$ justifies the definition of score.

The score formula simplifies via the following lemma to the formula in the original XGBoost paper [11, eq.(6)]. A proof of this equivalence is given in Appendix A:

Lemma 2.1. *Let $\partial_b\,\texttt{loss}$ and $\partial_b^2\,\texttt{loss}$ be the first and second order partial derivatives of the function loss with respect to the second variable and let $g_i = \partial_b\,\texttt{loss}(y_i, \widehat{y_i}^{(T)})$ and $h_i = \partial_b^2\,\texttt{loss}(y_i, \widehat{y_i}^{(T)})$ for $1 \le i \le N$. One has (see [11]):*

$$\texttt{score}(\texttt{TreeStructure}) = \sum_{\texttt{leaves}\,\ell} \frac{G_\ell^2}{2(H_\ell + \lambda)}, \quad (5)$$

where

$$G_\ell = \sum_{i \in \ell} g_i \text{ and } H_\ell = \sum_{i \in \ell} h_i, \quad (6)$$

where $i \in \ell$ denotes summing over all instances that visit node ℓ.

Between each step, we update the tree structure by picking the split (feature and threshold value) that maximizes the score: since we split only one node at a time, we only need to account for the contribution of the new left and right leaves in the above score, the rest of the leaves remaining unchanged.

Consider now splitting a node n, i.e. attaching two children nodes n_{left} and n_{right} to n. The gain associated to this split is the difference in the objective resulting from attaching these two children nodes:

$$\texttt{gain} = \frac{G_{n_{\text{left}}}^2}{2(H_{n_{\text{left}}} + \lambda)} + \frac{G_{n_{\text{right}}}^2}{2(H_{n_{\text{right}}} + \lambda)} - \frac{G_n^2}{2(H_n + \lambda)} - \gamma \quad (7)$$

Note that the gain needs to take into account $-\gamma$, since splitting a node results in an increment of the total number of nodes.

2.3 MPC representation of a tree

Our multiparty computation (MPC) protocol is based on additive secret sharing as defined in [9]. An element x is said to be secret shared among the p players $P_1, \ldots, P_p$, if every player P_i holds an x_i, such that $x_1 + x_2 + \ldots + x_p = x$. We use $[\![x]\!]$ to denote a p-tuple of secret shares $(x_1, \ldots, x_p)$. According to the context, these secret shares can be either binary, modular integers, floating-point numbers, etc.. We apply a classical approach based on Beaver triples [5] for the evaluation of linear combinations and multiplications.

The TreeStructure is secret shared. For every internal node n, the feature index j_n is kept secret and encoded as an additively secret shared vector $\mathtt{e}_n$ of size k. Here $\mathtt{e}_n$ is an indicator vector with single one and the other values set to 0. The set of all these secret shared vectors is denoted $[\![\mathbf{E}]\!]$.

To access the j_nth column, we simply compute the matrix multiplication $[\![X]\!] \cdot [\![\mathtt{e}_n]\!]$, thus, keeping the tree structure private. The thresholds $\mathtt{t}_n$ and the weights $\mathtt{w}_\ell$ are also kept secret and additively secret shared; we use the notation $[\![\mathtt{t}_n]\!]$ and $[\![\mathtt{w}_\ell]\!]$ to indicate the secret shared values and $[\![\mathbf{T}]\!]$ and $[\![\mathbf{W}]\!]$ to denote the set of secret shared thresholds and weights respectively. The TreeStructure and TreeWeights are secret shared throughout training and evaluation.

Several MPC protocols and libraries in the literature provide a practical method for combining arithmetic and Boolean shares [9, 19, 23, 32, 33]. We implement XORBoost leveraging the Manticore efficient conversions between fixed-point number representation and boolean representation [9], but our algorithm is agnostic to the choice of MPC framework. We optimize memory and runtime with the following choice: the real-valued `TreeWeights` and thresholds are secret shared as real numbers in fixed-point representation while the feature indices corresponding to the inner node splits are secret shared as Boolean vectors of length k. Using the `Manticore` framework, we achieve full-threshold, information-theoretic security in semi-honest security model with an offline trusted dealer.

3 Data preprocessing phase

A major practical challenge for splitting a node into a left child and a right child in the `xgboost` algorithm is that, *a priori*, the maximal gain is computed over one discrete variable (the feature index) and one continuous variable (the threshold value corresponding to that feature). To discretize the search for the threshold, one uses histograms for the feature values. Computing these histograms in a privacy-preserving manner is challenging - it requires to obliviously sort the feature vectors and extract the (secret) sorting permutations, as explained in Section 3.1.

There are multiple ways to secret share a permutation - for instance, one secret sharing method (but not the only one) used in, e.g., the `Manticore` framework, is based on secret sharing of the (binary) states of the switches of a Benes network. A permutation of N elements can thus be represented by a Boolean matrix of dimensions $N(2\lfloor\log_2 N\rfloor+1)/2$. We use the notation $[\![\pi]\!]$ to indicate a secret sharing of a permutation π.

The secret shared sorting permutations are needed later for the training procedure (see Algorithm 6). Obliviously applying a secret shared permutation is expensive: for an input vector of size N it requires $2\lfloor\log_2 N\rfloor + 1$ element-wise multiplications of two vectors of length $N/2$. In Section 3.2 we introduce a novel algorithm that reduces the number of times these permutations are applied, leading to a significant gain in efficiency ($\mathcal{O}(\log_2 B)$ instead of $\mathcal{O}(B)$ per feature, where B is the number of buckets in the histogram).

3.1 Oblivious sorting of feature vectors and oblivious histograms

State-of-the-art MPC algorithms and framework enable for efficient sorting of numerical vectors. For instance, the `Manticore` framework achieves this by applying a precomputed, secret shared, uniformly random permutation to the target vector on which a variant of the quicksort algorithm is applied [9, §5.2]. Throughout the computation, the input vector remains secret shared and nothing about the underlying data is revealed. The output contains the sorted vector and / or the secret shared sorting permutation and its inverse. While the `Manticore` algorithm supports oblivious sorting of vectors with repeated values, this method is not suited for categorical feature vectors. For these, we refer the reader to Section 6.

We now introduce some notation used throughout this paper: given a permutation σ on N elements and a vector $v = (v_1, \ldots, v_N) \in \mathbb{R}^N$, σ acts on v by permuting the coordinates, i.e.,

$$\sigma(v) := (v_{\sigma(i)})_{i=1}^N.$$

We define the function `obliv_perm`, that takes as input a secret shared vector $[\![v]\!]$ of size N together with a list of $r > 0$ secret shared permutations $[\![\sigma_1]\!], \ldots, [\![\sigma_r]\!]$ and returns a secret shared $N \times r$ matrix, whose jth column corresponds to the vector $\sigma_j(v)$.

Finally, let π_j, $j = 1, \ldots, k$, be the sorting permutation of the feature column $X^{(j)}$ and π_j^{-1} its inverse permutation. For a vector v of size N we use the notation Π_v to denote the $N \times k$ matrix $[\pi_1(v)|\ldots|\pi_k(v)]$.

3.1.1 Bucket vectors

As mentioned earlier, we use histograms to discretize the search space for the thresholds. More specifically, given the number of buckets B in the histogram and a feature vector $X^{(j)}$, we only consider the $B-1$ values

$$\mathtt{t}_b^{(j)} := \pi_j\left(X^{(j)}\right)_{b\lfloor N/B\rfloor+1}, \quad b = 1, \ldots, B-1, \tag{8}$$

as possible threshold candidates for the given feature (note that $\mathtt{t}_0^{(j)}$ corresponds to an empty split).

Recall that an inner node $(j, \mathtt{t})$ is evaluated for sample x using the predicate $x_j < \mathtt{t}$ where j is the feature index and $\mathtt{t}$ is the threshold value.

We assume that all elements in a feature column $X^{(j)}$ are unique. We have observed empirically that repeated values do not have a large impact on the training or testing loss achieved (5% repetition rate). As such, we do not take extra steps to mitigate the effect of repetitions. It would be possible to use the technique found in [3, Section 4.2] to do so.

We introduce the selector vectors, a set of special binary vectors. For each bucket index $b = 1, \ldots, B-1$

and each feature index $j = 1, \dots, k$, define the binary *selector vector* (of size N)

$$s_b^{(j)} := (X^{(j)} < \mathtt{t}_b^{(j)}). \qquad (9)$$

This is the characteristic vector of the elements in $X^{(j)}$ that belong to the first b buckets and it is crucial in the computation of the node splittings in the training procedure (see Section 4.3).

Note the following equality of binary vectors of size N:

$$s_b^{(j)} = \pi_j^{-1}(\mathtt{BV}_b),\ \forall\, b = 1, \dots, B-1, \qquad (10)$$

where

$$(\mathtt{BV}_b)_i := \begin{cases} 1 \text{ if } i \le b\lfloor N/B \rfloor \\ 0 \text{ otherwise,} \end{cases} \quad \forall\, i = 1, \dots N. \qquad (11)$$

We refer to $\mathtt{BV}_b$ as the bth *bucket vector*.

During training, it is convenient to keep a record of the path taken by the samples in the training data, and the selector vectors $s_b^{(j)}$ will play a crucial role, see Section 4.2.2. Equality (10) means that one can compute the secret selector vector $s_b^{(j)}$ by applying the secret permutation π_j^{-1} to the public bucket vector $\mathtt{BV}_b$, which can be written as

$$[\![s_b^{(j)}]\!] = [\![\pi_j^{-1}]\!](\mathtt{BV}_b).$$

3.2 Bucket vectors and permutations

Naïvely, one would compute the selector vectors $\pi_j^{-1}(\mathtt{BV}_b)$ for feature j and buckets $b = 1, \dots, B-1$, with $B-1$ calls to obliv_perm. However, it is possible to do this with only $\log_2 B$ calls. This optimization is a major contribution of the paper and leads to practical speedups.

To achieve this, we first explain how to construct the bucket vectors $\mathtt{BV}_1, \dots, \mathtt{BV}_{B-1}$ from a set of $\log_2 B$ publicly known *generating vectors* via Algorithm 1. We then leverage the fact that Algorithm 1 commutes with the function obliv_perm, to achieve an efficient generation of the selector vectors in Algorithm 2.

3.2.1 Generating bucket vectors

For simplicity of the exposition and without loss of generality, assume that $B = 2^s$ and that B divides N, each bucket of the histogram therefore holding exactly N/B samples.

Let C be the $N \times s$ binary matrix holding the binary expansion of $(B-1) - \lfloor (i-1)/(N/B) \rfloor$ in row i. For example, if $B = 4$ and $N = 4$, we have

$$C = \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$

We write $C^{(m)}$ for column m of C, that is, for $i = 1, \dots, N$ and $m = 1, \dots, s$, we have

$$C_i^{(m)} := [(B-1) - b_i]_{m-1}, \qquad (12)$$

where $b_i \in \{0, \dots, B-1\}$ is the quotient of the euclidean division of $i-1$ by N/B and where $[\cdot]_r$ denotes the rth bit of the binary expansion, for $r = 0, \dots, B-1$.

The s columns of C are called *generating vectors* and can be used to generate any bucket vector as shown in Algorithm 1, where we are using the notation

$$b\,?\,a : c = \begin{cases} a & \text{if } b = 1 \\ c & \text{if } b = 0. \end{cases}$$

Algorithm 1 bucket_vector

```
Input:   – $b$ - bucket index, $1 \le b \le B-1$
         – $C = [C^{(1)}|\dots|C^{(s)}]$ - matrix from (12)
Output: The $b$th bucket vector $\mathtt{BV}_b$
1: res ← $0_{N\times 1}$                          ▷ vector of 0s
2: for $m = 1, \dots, s$ do
3:     res ← $[b]_{m-1}$ ? res $\vee C^{(m)}$ : res $\wedge C^{(m)}$
4: end for
5: return res
```

Lemma 3.1. *Algorithm 1 outputs the vector* $\mathtt{BV}_b$.

A proof is given in Appendix B.

3.2.2 Constructing selector vectors

Since Algorithm 1 only requires AND and OR operations on the generating vectors $C^{(1)}, \dots, C^{(s)}$, the function obliv_perm and Algorithm 1 commute, i.e., for each sorting permutation π_j, the selector vector $\pi_j^{-1}(\mathtt{BV}_b)$ is given by

$$\begin{aligned} \pi_j^{-1}(\mathtt{BV}_b) &= \pi_j^{-1}(\mathtt{bucket\_vector}(b, C)) \\ &= \mathtt{bucket\_vector}(b, \pi_j^{-1}(C)). \end{aligned}$$

where $\pi_j^{-1}(C)$ is the matrix $[\pi_j^{-1}(C^{(1)})|\dots|\pi_j^{-1}(C^{(s)})]$. Hence, if we define

$$\mathrm{C}_{j,m} := \pi_j^{-1}\left(C^{(m)}\right),\ j = 1,\dots,k,\ m = 1,\dots,s, \quad (13)$$

then one can reconstruct all $(B-1)\cdot k$ selector vectors via Algorithm 2 using only $s\cdot k$ calls to `obliv_perm` (to compute $[\![C_{j,m}]\!] := [\![\pi_j^{-1}]\!]\left(C^{(m)}\right)$).

Algorithm 2 `sel_vec`

```
Input:  – $b$ - bucket index, $1 \le b \le B-1$
        – $\{[\![C_{j,m}]\!]\colon m = 1,\dots,s\}$ - secret shared generating vectors as from (13)
Output: Secret shared selector vector $[\![s_b^{(j)}]\!]$
1: $[\![\mathrm{res}]\!] \leftarrow [\![0_{N\times 1}]\!]$    ▷ vector of 0s
2: for $m = 1,\dots,s$ do
3:     $[\![\mathrm{res}]\!] \leftarrow [b]_{m-1}\,?\,[\![\mathrm{res}]\!] \vee [\![C_{j,m}]\!] : [\![\mathrm{res}]\!] \wedge [\![C_{j,m}]\!]$
4: end for
5: return $[\![\mathrm{res}]\!]$
```

3.3 Oblivious bucketing algorithm (preprocessing)

We summarize our oblivious bucketing algorithm in 3:

Algorithm 3 `obliv_bucket`

```
Input: $[\![X]\!]$ - secret shared $N \times k$ feature matrix
Output: – $[\![\Pi]\!] = ([\![\pi_1]\!],\dots,[\![\pi_k]\!])$ - secret shared bucketing permutations
        – $\{[\![\mathrm{ID}_{b,j}]\!]\colon b = 1,\dots,B-1,\ j = 1,\dots,k\}$ - secret shared identifiers
1: for $j = 1,\dots,k$ do
2:     $[\![\pi_j]\!], \{[\![\mathrm{t}_b^{(j)}]\!]\}_{b=1}^{B-1} \leftarrow \mathtt{sort}([\![X^{(j)}]\!], B)$
3:     for $m = 1,\dots,s$ do
4:         $[\![C_{j,m}]\!] \leftarrow \mathtt{obliv\_perm}([\![\pi_j^{-1}]\!], C^{(m)})$
5:     end for
6:     $[\![\mathrm{e}^{(j)}]\!] \leftarrow [\![\delta_{ij}]\!]$    ▷ Kronecker delta (in $\mathbb{R}^k$)
7:     for $b = 1,\dots,B-1$ do
8:         $[\![s_b^{(j)}]\!] \leftarrow \mathtt{sel\_vec}(b, \{[\![C_{j,m}]\!]\}_{m=1}^{s})$
9:         $[\![\mathrm{ID}_{b,j}]\!] \leftarrow ([\![\mathrm{t}_b^{(j)}]\!], [\![\mathrm{e}^{(j)}]\!], [\![s_b^{(j)}]\!])$
10:    end for
11: end for
12: return $[\![\Pi]\!]$, $\{[\![\mathrm{ID}_{b,j}]\!]\}_{b,j}$
```

Here, we assume `sort` is an oblivious sorting procedure. For further computational gains, we can use a partial sort at the B threshold positions, see for example `Manticore`'s partial quicksort algorithm from [9, §5.2] The saving of applying oblivious permutations is achieved in the body of the main loop (in particular, lines 4 and 7). The identifier will serve as one of the inputs to the `argmax` function.

`argmax`
Here, the `argmax` function, similar to [3, A.4], is a recursive divide-and-conquer algorithm, taking as input the secret shared gain matrix $\mathtt{gains}_n$ for a given node $n \in L_d$, as well as the secret-shared identifier matrix `ID` from Algorithm 3. The function `argmax` obliviously computes the argument of the maximum of $\mathtt{gains}_n$. That is, if $(\mathtt{gains}_n)_{b_*,j_*}$ is maximal, `argmax` returns $[\![\mathrm{ID}_{b_*,j_*}]\!]$.

4 Description of the XORBoost training algorithm

In general, the input to the training algorithm is the secret shared feature matrix $[\![X]\!]$ as well as the secret shared response vector $[\![y]\!]$. To train an ensemble of a specified number T of binary decision trees, we proceed as follows: assuming that we have already trained the first $t-1$ trees, to grow the tth tree to a given depth D we iterate by layers starting from layer zero, the root node. At each iteration, we 'split' each leaf into a left and a right child via a *splitting criterion*, a pair of a feature index and a threshold value, chosen to maximize the gain (see Algorithm 4).

The data preprocessing phase described in Section 3.3 has already discretized the threshold values by introducing the histogram/buckets, i.e., obliviously sorting each feature as described in Section 3.1 and obliviously building the histograms of Section 3.1.1. To compute the optimal splitting criterion, a plaintext algorithm would first compute a $(B-1)\times k$ gain matrix of all possible gain values and choose the largest entry. In order to adapt this optimization procedure to the MPC setting, we compute the optimal splitting criteria by first computing the gain matrix obliviously and then computing (also obliviously) the (secret shared) feature selector e, the (secret shared) threshold t and the (secret shared) selector vector $s_b^{(j)}$ corresponding to the feature index j and the bucket index b of the optimal splitting criterion. This allows us to decide which samples would go to the left child and to the right child, thus, defining the split of the node. We now go into more detail for each of these steps.

Algorithm 4 `xorboost_train`

```
Input:  – $[\![X]\!]$ - secret shared feature matrix of size $N \times k$
        – $[\![y]\!]$ - secret shared response vector of size $N$
Output: A tree ensemble $\{[\![\texttt{Tree}^{(1)}]\!], \ldots, [\![\texttt{Tree}^{(T)}]\!]\}$.
        Each secret shared tree $[\![\texttt{Tree}]\!]$ consists of the following data:
        – $[\![\mathbf{T}]\!]$ - set of secret shared threshold values $[\![\texttt{t}_n]\!]$ for each non-leaf node $n$
        – $[\![\mathbf{E}]\!]$ - set of secret shared feature selectors $[\![\texttt{e}_n]\!]$ for each non-leaf node $n$
        – $[\![\mathbf{W}]\!]$ - set of secret shared weights $[\![\texttt{w}_\ell]\!]$ for each leaf node $\ell$
1: $[\![\widehat{y}^{(0)}]\!] \leftarrow \texttt{initialize}([\![X]\!], [\![y]\!])$
2: $[\![\Pi]\!], \{[\![\texttt{ID}_{b,j}]\!]\}_{b,j} \leftarrow \texttt{obliv\_bucket}([\![X]\!])$
3: for $t = 1, \ldots, T$ do
4:     $[\![g^{(t-1)}]\!] \leftarrow \partial_b \texttt{loss}\left([\![y]\!], [\![\widehat{y}^{(t-1)}]\!]\right)$    ▷ element-wise
5:     $[\![h^{(t-1)}]\!] \leftarrow \partial_b^2 \texttt{loss}\left([\![y]\!], [\![\widehat{y}^{(t-1)}]\!]\right)$    ▷ element-wise
6:     $[\![\texttt{Tree}^{(t)}]\!], [\![\texttt{Tree}^{(t)}(X)]\!] \leftarrow \texttt{grow\_tree}([\![g^{(t-1)}]\!], [\![h^{(t-1)}]\!], [\![\Pi]\!], \{[\![\texttt{ID}_{b,j}]\!]\}_{b,j})$
7:     $[\![\widehat{y}^{(t)}]\!] \leftarrow [\![\widehat{y}^{(t-1)}]\!] + \eta \cdot [\![\texttt{Tree}^{(t)}(X)]\!]$
8: end for
9: return $\{[\![\texttt{Tree}^{(1)}]\!], \ldots, [\![\texttt{Tree}^{(T)}]\!]\}$
```

4.1 Computing initial predictions

The computation of the first and second order statistics vectors g and h of the loss function only depends on the response variable y and the current estimate $\widehat{y}^{(t)}$. Since the tree ensemble is initially empty, we must provide an initial estimate $\widehat{y}^{(0)}$ in order to grow the first tree. There are several possibilities for the initialization of $\widehat{y}^{(0)}$:

- The zero vector.
- The constant vector with value α minimizing $\sum_{i=1}^{N} \texttt{loss}(y_i, \alpha)$. For instance, for L_2-loss function, this corresponds to $\alpha = \frac{1}{N}\sum_{i=1}^{N} y_i$, and for logistic loss this corresponds to $\alpha = \sigma^{-1}\left(\frac{1}{N}\sum_{i=1}^{N} y_i\right)$, where σ is the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$.
- Leveraging previous work on ridge regression (resp., logistic regression) [9], we can use its prediction to initialize the boosted trees model in the case of L_2-loss (resp., the logistic loss). The aim here is to bootstrap the gradient boosting procedure by starting with a better initial value for $\widehat{y}^{(0)}$ (namely $\widehat{y}^{(0)} = X \cdot \theta$) allowing to eliminate the linear relation between X and y and reduce the number of trees required to obtain a model with good predictive power.

In all cases, we assume that we have defined a function $\texttt{initialize}(X, y)$ that will compute the initial predictions.

4.2 Oblivious permutations and computing gain matrices

We now explain how to efficiently apply oblivious permutations in order to compute gains and weights. Recall from Section 3.1 that we have obliviously sorted the feature columns of X using a set $[\![\Pi]\!] = \{[\![\pi_1]\!], \ldots, [\![\pi_k]\!]\}$ of k secret shared sorting permutations. The computation of the weights and the gain matrices includes two types of secret shared vectors: *instance vectors* and the already introduced *bucket vectors* from Section 3.2.

Associated to each node $n \in \mathcal{N}$ is an instance vector $\texttt{IV}_n$, that is, a binary vector of size N indicating which of the N samples in the training dataset go through node n when evaluated on the tree. Throughout, we keep these vectors secret shared and use the notation $[\![\texttt{IV}_n]\!]$ for the secret-shared vector. Define

$$[\![\Pi_n]\!] := \texttt{obliv\_perm}([\![\Pi]\!], [\![\texttt{IV}_n]\!]) \tag{14}$$

$$= [[\![\pi_1]\!]([\![\texttt{IV}_n]\!])| \ldots |[\![\pi_k]\!]([\![\texttt{IV}_n]\!])] \tag{15}$$

to be the permuted instance matrix for node n, a $N \times k$ binary secret shared matrix.

We grow trees layer-by-layer. For a given depth d, recall that L_d denotes the set of the 2^d nodes at depth d (L_0 consists of the root only, L_1 consists of the two children of the root, etc.). The following two properties of instance vectors are easy to verify:

$$\sum_{n \in L_d} \texttt{IV}_n = 1_N; \tag{16}$$

$$\texttt{IV}_n = \texttt{IV}_{n_{\text{left}}} + \texttt{IV}_{n_{\text{right}}}. \tag{17}$$

4.2.1 Computing gain matrices

Recall that computing gain (7) requires computing the quantities $G_n, H_n, G_{n_{\text{left}}}, H_{n_{\text{left}}}, G_{n_{\text{right}}}$ and $H_{n_{\text{right}}}$ from (6), for a given node n and their left and right children nodes n_{left} and n_{right}, respectively. During training, one must compute these quantities for all the $(B-1)k$ possible splits. If we know the instance vector of a node then these quantities can conveniently be computed as:

$$G = \langle g, \mathtt{IV}\rangle,\ H = \langle h, \mathtt{IV}\rangle, \tag{18}$$

or more generally, if σ is any permutation on N elements (in particular, any of the bucketing permutations), we have

$$G = \langle \sigma(g), \sigma(\mathtt{IV})\rangle,\ H = \langle \sigma(h), \sigma(\mathtt{IV})\rangle. \tag{19}$$

Naïvely, one can compute the quantities from (7) as follows: for each of the $(B-1)k$ possible splits, compute the corresponding instance vectors of the two children nodes via the oblivious binary AND operation $\mathtt{IV}_n \wedge s_b^{(j)}$ on binary vectors of size N, and then leverage (18) to compute $G_{n_{\text{left}}}$ by executing $(B-1)k$ inner-products of vectors of length N (and similarly for $H_{n_{\text{left}}}$).

Instead, we observe the fact that π_j is sorting the feature column $X^{(j)}$ into buckets. That is, the top $b\lfloor B/N\rfloor$ entries of $\pi_j(\mathtt{IV}_n)$ are in correspondence with the samples in node n that satisfy $X_i^{(j)} < \mathtt{t}_b^{(j)}$. Using (19), we find

$$G_{n_{\text{left}}} = \langle \pi_j(g), \pi_j(\mathtt{IV}_n)\rangle_{b\lfloor \frac{N}{B}\rfloor},\ H_{n_{\text{left}}} = \langle \pi_j(h), \pi_j(\mathtt{IV}_n)\rangle_{b\lfloor \frac{N}{B}\rfloor}, \tag{20}$$

where $\langle\cdot,\cdot\rangle_r\colon \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}$ denotes the inner-product over the first r entries. Using (19) and (17) we have

$$G_{n_{\text{right}}} = G_n - G_{n_{\text{left}}},\ H_{n_{\text{right}}} = H_n - H_{n_{\text{left}}}.$$

for every pair of children nodes. We then define a function $\mathtt{partial\_inner}\colon \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}^{B-1}$ by

$$\mathtt{partial\_inner}(\cdot,\cdot) := \begin{pmatrix} \langle\cdot,\cdot\rangle_{\lfloor N/B\rfloor} \\ \vdots \\ \langle\cdot,\cdot\rangle_{(B-1)\lfloor N/B\rfloor} \end{pmatrix}, \tag{21}$$

which extends naturally to a function $\mathtt{partial\_inner}\colon \mathbb{R}^{N\times k} \times \mathbb{R}^{N\times k} \to \mathbb{R}^{(B-1)\times k}$.

In Section 4.2.2, we explain how to compute all 2^d permuted instance vectors $\pi_j(\mathtt{IV}_n)$ for $n \in L_d$ using a single call to obliv_perm. Thus, computing the $2^d(B-1)k$ quantities $G_{n_{\text{left}}}, H_{n_{\text{left}}}$ at depth d with the naïve strategy results in an asymptotic complexity of $\mathcal{O}(N2^dBk)$ oblivious scalar multiplications at depth d (2^dBk multiplications of two vectors of length N each). In comparison, the asymptotic complexity of our strategy is $\mathcal{O}(kN(\log_2 N + 2^d))$ oblivious scalar multiplications (2^dk multiplications of vectors of length N and a single call to obliv_perm that requires $2k\log_2 N$ multiplications of length $N/2$). This allows to choose much larger values of B.

4.2.2 Efficiently computing instance vectors

We define the *compressed instance vector* IVC at depth d

$$\mathtt{IVC} := \bigoplus_{n\in L_d} \mathtt{IV}_{n_{\text{left}}}, \tag{22}$$

where $\bigoplus$ is the logical XOR operator.

Using (16), we see that the $\oplus$ operation in the definition is equivalent to $\vee$. This allows us to obliviously compute the permuted instance matrix $[\![\Pi_{n_{\text{left}}}]\!]$ for the left child of a given node n by simply applying the $\wedge$ operator on the permutation instance matrix for n with the compressed instance vector for that level, i.e., by combining the identity

$$[\![\mathtt{IV}_{n_{\text{left}}}]\!] = [\![\mathtt{IV}_n]\!] \wedge [\![\mathtt{IVC}]\!] \tag{23}$$

with the commutativity of permutations and logical operations. Thus,

$$[\![\Pi_{n_{\text{left}}}]\!] = [\![\Pi_n]\!] \wedge \mathtt{obliv\_perm}([\![\Pi]\!], [\![\mathtt{IVC}]\!]) \tag{24}$$

This explains the interest in IVC: instead of applying $[\![\Pi]\!]$ to each $[\![\mathtt{IV}_{n_{\text{left}}}]\!]$, we can apply $[\![\Pi]\!]$ once to $[\![\mathtt{IVC}]\!]$ and compute $[\![\Pi_{n_{\text{left}}}]\!]$ using a significantly cheaper $\wedge$ operation for each $n \in L_d$.

Algorithm 5 computes the $(B-1)\times k$ matrix of gains for each split candidate. Here, the function priv_div is any element-wise private division in the multiparty computation setting (see e.g. [9, §4.3] inspired by Goldschmidt's method). The subtraction of the scalars $\mathtt{priv\_div}([\![G_n^2]\!], 2\cdot([\![H_n]\!]+\lambda))$ and γ has to be understood coordinate-wise.

4.3 Growing a tree

We now present the main privacy-preserving algorithm for growing a tree. The algorithm iterates over the layers and the nodes in each layer and successively splits each node until the desired depth D of the tree is achieved. It then computes the leaf weights to obtain the secret shared tree $[\![\mathtt{Tree}]\!]$. For each inner node n, the secret shared threshold value $[\![\mathtt{t}_n]\!]$ and the secret shared feature selector $[\![\mathtt{e}_n]\!]$ are computed during the layer splitting whereas the secret shared weights $[\![\mathtt{w}_\ell]\!]$, for $\ell \in L_D$, are computed once the tree has reached its full depth. See Algorithm 6.

Algorithm 5 gain

```
Input:  – $[\![\Pi_g]\!]$ - secret shared $N \times k$ matrix for the permuted first order statistics vector
        – $[\![\Pi_h]\!]$ - secret shared $N \times k$ matrix for the permuted second order statistics vector
        – $[\![\Pi_n]\!]$ - secret shared permuted instance matrix of size $N \times k$ for node $n$ (see (14))
Output: $[\![\mathtt{gains}_n]\!]$ - secret shared $(B-1) \times k$ gain matrix for node $n$, see (7)
 1: $[\![G_{n_\mathrm{left}}]\!] \leftarrow$ partial_inner($[\![\Pi_g]\!], [\![\Pi_n]\!]$)    ▷ see (21)
 2: $[\![G^2_{n_\mathrm{left}}]\!] \leftarrow [\![G_{n_\mathrm{left}}]\!] \cdot [\![G_{n_\mathrm{left}}]\!]$    ▷ element-wise
 3: $[\![G_n]\!] \leftarrow \langle [\![\Pi_g^{(1)}]\!], [\![\Pi_n^{(1)}]\!] \rangle$    ▷ does not depend on the choice of permutation
 4: $[\![G^2_n]\!] \leftarrow [\![G_n]\!] \cdot [\![G_n]\!]$
 5: $[\![G_{n_\mathrm{right}}]\!] \leftarrow [\![G_n]\!] - [\![G_{n_\mathrm{left}}]\!]$    ▷ element-wise
 6: $[\![G^2_{n_\mathrm{right}}]\!] \leftarrow [\![G_{n_\mathrm{right}}]\!] \cdot [\![G_{n_\mathrm{right}}]\!]$    ▷ element-wise
 7: $[\![H_{n_\mathrm{left}}]\!] \leftarrow$ partial_inner($[\![\Pi_h]\!], [\![\Pi_n]\!]$)
 8: $[\![H_n]\!] \leftarrow \langle [\![\Pi_h^{(1)}]\!], [\![\Pi_n^{(1)}]\!] \rangle$    ▷ does not depend on the choice of permutation
 9: $[\![H_{n_\mathrm{right}}]\!] \leftarrow [\![H_n]\!] - [\![H_{n_\mathrm{left}}]\!]$    ▷ element-wise
10: Compute
        $[\![\mathtt{gains}_n]\!] \leftarrow$ priv_div($[\![G^2_{n_\mathrm{left}}]\!]$, $2([\![H_{n_\mathrm{left}}]\!] + \lambda)$)
                       + priv_div($[\![G^2_{n_\mathrm{right}}]\!]$, $2([\![H_{n_\mathrm{right}}]\!] + \lambda)$)
                       − priv_div($[\![G^2_n]\!]$, $2([\![H_n]\!] + \lambda)$)
                       − $\gamma$
        ▷ coordinate-wise subtractions of the scalars
11: return $[\![\mathtt{gains}_n]\!]$
```

5 Prediction

Our privacy-preserving prediction algorithm provides the same privacy guarantees as our training protocol: only the shape of the model is public knowledge (tree depth and number of trees) while everything else remains secret (threshold values, feature selectors, leaf weights and prediction values). Similar to [**?**] we achieve this by obliviously evaluating the predicate of each node in the tree and thus hiding the path taken. Previous work on private and secure decision tree inference was done in [14] and [17] for passive and active security in the 2-player settings.

Recall that each non-leaf node consists of a feature selector e and a threshold value t (both secret shared). To evaluate a non-leaf node at a sample x (public or private), the feature selector is used to extract the feature of interest via an inner-product $\langle x, \mathtt{e} \rangle$, which is then compared against the threshold value t. This can be computed securely, yielding a secret shared bit that indicates if the left or the right subtree is to be evaluated next. To evaluate a tree of depth D at a sample x we proceed in two steps. In a first step we compute a binary unit vector β of size 2^D with a 1 at the position of the index (in L_D) of the unique leaf-node the sample visits. In a second step we compute the inner-product $\langle \beta, \mathtt{w}_{L_D} \rangle$, where $\mathtt{w}_{L_D}$ is the vector of weights $(\mathtt{w}_1, \dots, \mathtt{w}_{2^D})$.

Note: Algorithm 7 can easily be adapted to evaluate multiple trees at multiple samples simultaneously. In practice, we compute line 3 for all split layers in one matrix-multiplication and one vector-wise oblivious comparison.

6 Categorical features

Many datasets include categorical features which are informative and should be handled adequately by the machine learning model. While the original XGBoost approach does not support categorical features, there has been subsequent work on this problem (see [35] for a survey). Here, we describe an approach that is compatible with the work presented so far.

Given a feature column $X^{(j)}$ with categorical values drawn from a set A, tree splits for this feature are obtained by partitioning A into a disjoint union of two subsets $A = A_1 \sqcup A_2$. We are interested in finding the split/partition that produces the greatest reduction in loss. By the main result of [13], the following procedure will find the best split if the loss function is either L_2 loss or logistic loss. It only considers $|A| - 1$ possible splits.

- For each categorical value $a \in A$, compute $y_a = \frac{\sum_{i, X_i^{(j)} = a} y_i}{\sum_{i, X_i^{(j)} = a} 1}$ the average response variable over all samples for a.
- Order the categorical values according to the y_a values, yielding a total order $\prec$ (breaking ties arbitrarily) on the set A.
- For every $a \in A$ besides the minimal element for $\prec$, consider the split S_a defined by $A_1 = \{a' : a' \prec a\}$ and $A_2 = \{a' : a' \succeq a\}$.

Let π_A be the sorting permutation for the values y_a. Given M the one hot encoded matrix for $X^{(j)}$, we apply π_A to its columns to obtain M_A, the matrix where the columns are sorted according to the values of y_a. In the context of categorical variables, the bucket vectors' role is played by indicator vectors for the splits S_a.

Algorithm 6 `grow_tree`

Input:
- $[\![g]\!]$ - first order statistics vector
- $[\![h]\!]$ - second order statistics vector
- $[\![\Pi]\!] = ([\![\pi_1]\!], \dots, [\![\pi_k]\!])$ - secret shared bucketing permutations
- $\{[\![\mathtt{ID}_{b,j}]\!] \colon b = 1, \dots, B-1,\ j = 1, \dots, k\}$ - secret shared identifiers

Output:
- $[\![\mathtt{Tree}]\!] = \{[\![\mathbf{T}]\!], [\![\mathbf{E}]\!], [\![\mathbf{W}]\!]\}$
- $[\![\mathtt{Tree}(X)]\!]$ - evaluation of Tree at X

```
1:  $[\![\Pi_g]\!] \leftarrow \mathtt{obliv\_perm}([\![\Pi]\!], [\![g]\!])$
2:  $[\![\Pi_h]\!] \leftarrow \mathtt{obliv\_perm}([\![\Pi]\!], [\![h]\!])$
3:  $[\![\mathtt{IV}_{\mathtt{Root}}]\!] \leftarrow [\![1_{N\times 1}]\!]$    ▷ vector of 1s
4:  $[\![\Pi_{\mathtt{Root}}]\!] \leftarrow [\![1_{N\times k}]\!]$    ▷ matrix of 1s
5:  $[\![\mathtt{Tree}(X)]\!] \leftarrow [\![0_{N\times 1}]\!]$    ▷ vector of 0s
6:  for $d = 0, \dots, D-1$ do
7:      for $n \in L_d$ do
8:          $[\![\mathtt{gains}_n]\!] \leftarrow \mathtt{gain}([\![\Pi_g]\!], [\![\Pi_h]\!], [\![\Pi_n]\!])$
9:          $[\![\mathtt{t}_n]\!], [\![\mathtt{e}_n]\!], [\![s_n]\!] \leftarrow \mathtt{argmax}\left([\![\mathtt{gains}_n]\!], \{[\![\mathtt{ID}_{b,j}]\!]\}_{b,j}\right)$    ▷ Section 3.3
10:         $[\![\mathtt{IV}_{n_{\mathrm{left}}}]\!] \leftarrow [\![\mathtt{IV}_n]\!] \wedge [\![s_n]\!]$    ▷ IV of left child
11:         $[\![\mathtt{IV}_{n_{\mathrm{right}}}]\!] \leftarrow [\![\mathtt{IV}_n]\!] \oplus [\![\mathtt{IV}_{n_{\mathrm{left}}}]\!]$    ▷ IV of right child
12:     end for
13:     $[\![\mathtt{IVC}]\!] \leftarrow \bigoplus_{n\in L_d} [\![\mathtt{IV}_{n_{\mathrm{left}}}]\!]$    ▷ see (22)
14:     $[\![\Pi_{\mathtt{IVC}}]\!] \leftarrow \mathtt{obliv\_perm}([\![\Pi]\!], [\![\mathtt{IVC}]\!])$
15:     for $n \in L_d$ do
16:         $[\![\Pi_{n_{\mathrm{left}}}]\!] \leftarrow [\![\Pi_n]\!] \wedge [\![\Pi_{\mathtt{IVC}}]\!]$    ▷ see (24)
17:         $[\![\Pi_{n_{\mathrm{right}}}]\!] \leftarrow [\![\Pi_n]\!] \oplus [\![\Pi_{n_{\mathrm{left}}}]\!]$
18:     end for
19:     $[\![\Pi_n]\!] \leftarrow [\![\Pi_{n_{\mathrm{left}}}]\!] \cup [\![\Pi_{n_{\mathrm{right}}}]\!]$
20: end for
21: for $\ell \in L_D$ do
22:     $[\![\mathtt{w}_\ell]\!] \leftarrow \mathtt{priv\_div}\left(-\langle [\![g]\!], [\![\mathtt{IV}_\ell]\!]\rangle, \langle [\![h]\!], [\![\mathtt{IV}_\ell]\!]\rangle + \lambda\right)$
23:     $[\![\mathtt{Tree}(X)]\!] \leftarrow [\![\mathtt{Tree}(X)]\!] + [\![\mathtt{w}_\ell]\!] \cdot [\![\mathtt{IV}_\ell]\!]$
24: end for
25: $[\![\mathbf{T}]\!] \leftarrow \{[\![\mathtt{t}_n]\!] \colon n \in L_d,\ d = 0, \dots, D-1\}$
26: $[\![\mathbf{E}]\!] \leftarrow \{[\![\mathtt{e}_n]\!] \colon n \in L_d,\ d = 0, \dots, D-1\}$
27: $[\![\mathbf{W}]\!] \leftarrow \{[\![\mathtt{w}_\ell]\!] \colon \ell \in L_D\}$
28: return $\{[\![\mathbf{T}]\!], [\![\mathbf{E}]\!], [\![\mathbf{W}]\!]\}$, $[\![\mathtt{Tree}(X)]\!]$
```

Algorithm 7 `xorboost_predict`

Input:
- $\mathtt{Tree} = \{[\![\mathbf{T}]\!], [\![\mathbf{E}]\!], [\![\mathbf{W}]\!]\}$ - secret shared tree of depth D
- x - public or secret shared sample (we therefore omit the bracket notation)

Output: $[\![\mathtt{Tree}(x)]\!]$

```
1: $[\![\beta]\!] \leftarrow [\![1_{1\times 1}]\!]$
2: for $d = 0, \dots, D-1$ do
3:     $[\![\beta_{L_d}]\!] \leftarrow (\langle x, [\![\mathtt{e}_n]\!]\rangle < [\![\mathtt{t}_n]\!] \colon n \in L_d)$
4:     $[\![\beta_{\mathrm{left}}]\!] \leftarrow [\![\beta]\!] \wedge [\![\beta_{L_d}]\!]$
5:     $[\![\beta_{\mathrm{right}}]\!] \leftarrow [\![\beta]\!] \oplus [\![\beta_{\mathrm{left}}]\!]$
6:     $[\![\beta]\!] \leftarrow ([\![\beta_{\mathrm{left},1}]\!], [\![\beta_{\mathrm{right},1}]\!] \dots [\![\beta_{\mathrm{left},2^d}]\!], [\![\beta_{\mathrm{right},2^d}]\!])$
7: end for
8: $[\![\mathtt{Tree}(x)]\!] \leftarrow \langle [\![\beta]\!], [\![\mathtt{w}_{L_D}]\!]\rangle$
9: return $[\![\mathtt{Tree}(x)]\!]$
```

These indicator vectors can be computed as sums over the rows of the sorted matrix M_A. For every $a \in A$ besides the smallest element and for every $i = 1, \dots, N$ we have:

$$(\mathtt{BV}_a)_i = \sum_{a' \prec a} (M_A^{(a')})_i = \begin{cases} 1 \text{ if } X_i^{(j)} \in \{a' \colon a' \prec a\} \\ 0 \text{ otherwise.} \end{cases}$$

7 Complexity analysis

Our complexity parameters are the following:

- N - number of training samples
- k - number of features
- D - depth of the tree
- B - number of buckets
- T - number of trees in the ensemble

We need to carefully list these parameters as the main terms in the various asymptotic complexity analyses would be different for the different ranges of these parameters. There is another parameter - the number of compute parties p in the MPC protocol. Since we will measure the complexity of the overall protocol in terms of oblivious scalar multiplications, oblivious binary operators and oblivious comparisons, this parameter will determine the complexity of these building blocks. The complexity grows quadratically with the number of players.

7.1 Training algorithm

The XORBoost training algorithm (Algorithm 4) has the following major phases adding to the asymptotic complexity:

1. obliv_bucket - Algorithm 3 for oblivious bucketing
   - k calls to sort on N elements and B buckets each
   - $k \log_2 B$ calls to obliv_perm on oblivious permutations of N elements
   - kB calls to sel_vec which reduces to $2kNB \log_2 B$ oblivious AND operations and $2kNB \log_2 B$ oblivious XOR operations; the major term in the cost is thus $\mathcal{O}(kNB \log_2 B)$ oblivious AND operations (on scalars, though our implementation perform operations directly on tensors; yet, this does not change the analysis from an asymptotic complexity perspective)
2. grow_tree - Algorithm 6 for tree growing (this is called T times)
   - $2k$ calls to obliv_perm on vectors of size N
   - $2^D - 1$ calls to gains
   - $2^D - 1$ calls to argmax
   - $(2^D - 1)N$ calls to oblivious AND operations on Boolean scalars and $(2^D - 1)N$ calls to oblivious XOR operations ($2^D - 1$ calls to AND and XOR on Boolean vectors of size N)
   - kD calls to obliv_perm on vectors of size N
   - $(2^D - 1)Nk$ calls to oblivious AND operations on Boolean scalars and $(2^D - 1)Nk$ calls to oblivious XOR operations ($2^D - 1$ calls to AND and XOR on Boolean matrices of size $N \times k$)
   - 2×2^D calls to an inner product on vectors of size N (equivalent to $2 \times 2^D N$ multiplications)
   - 2^D calls to priv_div
3. The extra computations such as the procedure initialize, the computation of the 1st and 2nd order statistics (Lines 4–5) as well as the T MPC additions in Line 7 do not change the asymptotic complexity of the overall algorithm.

Oblivious sorting

Our main oblivious bucketing algorithm is a slight generalization of the oblivious sorting algorithm based on quicksort and described in more detail in [9, §5.2].

Oblivious permutations

Obliviously applying a secret shared permutation is expensive: for an input vector of size N it requires $2 \lfloor \log_2 N \rfloor + 1$ element-wise multiplications of two vectors of length $N/2$. As there are a total of $(2 + D)k$ calls to obliv_perm, the overall complexity is $\mathcal{O}(DkN \log_2 NT)$ oblivious scalar multiplications.

argmax

The overall complexity of argmax is given by $\mathcal{O}(\log_2(Bk))$ vector comparisons whose sizes are $Bk/2, Bk/2^2, \ldots, Bk/2^{\lfloor \log_2(Bk) \rfloor}$, respectively, thus, resulting in a total of $\mathcal{O}(Bk)$ comparisons and $\mathcal{O}(\log_2(Bk))$ vector multiplications (again, of sizes $Bk/2, \ldots, Bk/2^{\lfloor \log_2(Bk) \rfloor}$), resulting in a total of $\mathcal{O}(Bk)$ scalar multiplications. The overall cost of the argmax calls is $\mathcal{O}(2^D BkT)$ oblivious scalar multiplications.

private division

One way to perform private division is to use a variant of Goldschmidt's algorithm; see [9, §4.3]. In our practical implementation, we use 10 multiplication and an initialization circuit whose cost is equivalent to three comparisons. Thus, the overall cost of the operations priv_div is $\mathcal{O}(2^D T)$ oblivious scalar multiplications and $\mathcal{O}(2^D T)$ oblivious comparisons.

gain

To complete the complexity analysis, we only need to analyze the complexity of Algorithm 5:

- The two calls to partial_inner require $\mathcal{O}(Nk)$ oblivious scalar multiplications (Lines 1 and 7).
- Lines 3 and 8 can reuse the oblivious scalar multiplications from Lines 1 and 7 respectively.
- Lines 2 and 6 require $\mathcal{O}(Bk)$ oblivious scalar multiplications, and Line 4 uses one oblivious scalar multiplication.
- Line 10 requires $2 \times (B - 1)k + 1$ calls to priv_div which, by Section 7.1, requires $\mathcal{O}(Bk)$ multiplications.

Hence, assuming $N > k$, the overall complexity of the computation of a single gain matrix is $\mathcal{O}(Nk)$. Thus, the overall complexity of the gain calls in the training algorithm is $\mathcal{O}(2^D NkT)$ oblivious scalar multiplications.

The total complexity of the training algorithm (Algorithm 4) is

– k oblivious sorting of length N
– $\mathcal{O}(TNk(D\log_2 N + 2^D) + Nk\log_2 B(B + \log_2 N))$ oblivious scalar multiplications
– $\mathcal{O}(TBk2^D)$ oblivious scalar comparisons
– $\mathcal{O}(T2^D)$ oblivious divisions of length $(B-1)k$ that corresponds to $\mathcal{O}(TBk2^D)$ oblivious scalar multiplications

The major term in the cost is thus $\mathcal{O}(TNk(D\log_2 N + 2^D) + Nk\log_2 B(B + \log_2 N) + TBk2^D)$ oblivious scalar multiplications. Observe that the optimization of section 3.2 allows one to have $\log_2 B$ instead of $\log_2 N$ in the second summand above, which is significant if $N >> B$.

7.2 Prediction algorithm

The cost of at depth d is

– 2^d oblivious inner products of vectors of size k
– an oblivious comparison of two vectors of size 2^d
– oblivious AND operation of two vectors of size 2^d
– oblivious XOR operation of two vectors of size 2^d,

followed by one inner product of two vectors of size 2^D. The overhead is thus $\mathcal{O}(2^d k)$ oblivious scalar multiplications and hence, the complexity is bounded by $\mathcal{O}(2^D k)$ oblivious scalar multiplications.

Since this is the complexity of evaluating a single tree on a single sample, the overall complexity of the prediction is $\mathcal{O}(2^D N_{\text{pred}} kT)$ oblivious scalar multiplications, where N_{pred} is the number of samples for prediction.

8 Benchmarks

Most benchmarks have been done on a single `n1-standard-8` (8 vCPUs, 30GB of RAM, SSD drive, Intel Xeon CPU Skylake 2.00GHz) Google Compute Engine virtual machine for 2 players. As such, these reported times do not take into account network transfer time. On the other hand, the measurements reported in Figure 5 were made in a distributed deployment (within the same region of Google Cloud) for 2 and 3 players and include the communication overhead. We only show measurements for the training phase since it far outweighs the prediction phase in terms of resource utilization.

8.1 Parameter scaling

In Figure 1, we show the impact of varying the dataset size for the utilization of network, memory and time.

Figure 2 highlights the strong dependency on the depth and the number of buckets. The total processing time remains reasonable to grow 10 trees. We have verified that the execution time grows linearly with the number of trees. We show in Figure 5 (Appendix) the influence of the number of players. As expected, the amount of data transferred and the wall time increase with the number of players. The memory stays stable since the matrix dimensions remain the same and most previous computed matrices can be discarded when training a new tree.

Figure 3 is a direct analog of [12, Figure 5] using the parameters defined in this paper [2].

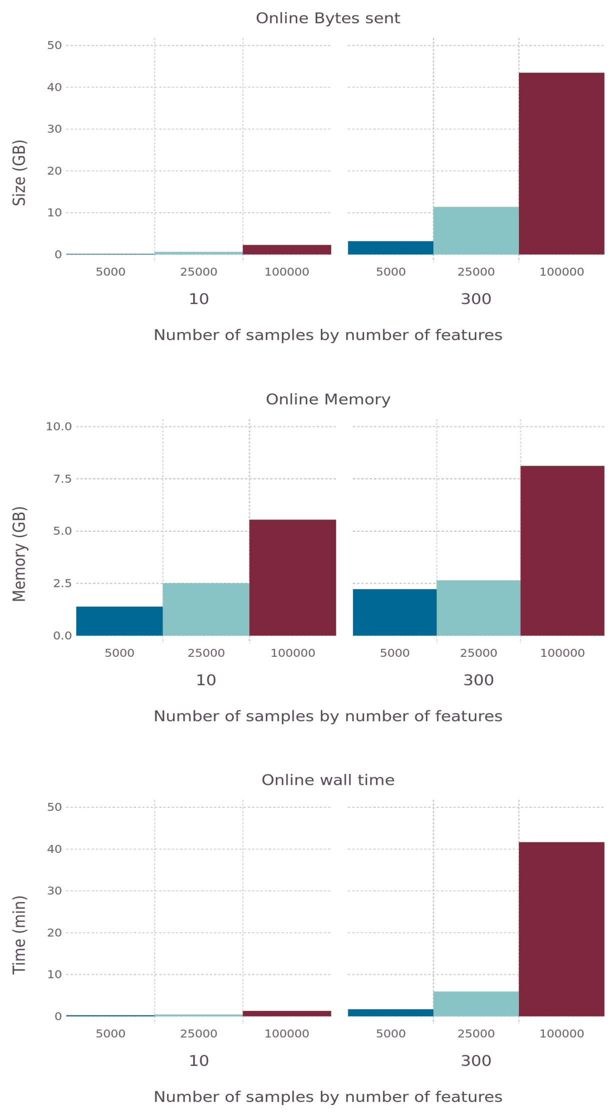
We see that the run time is of the same order while the privacy guarantees are improved since XORboost does not reveal intermediary results.

Although our framework leverages a bucketing strategy to handle larger datasets, we compared it with [3] where no bucketing is possible by setting the number of buckets to be equal to the number of samples. For 8192 samples, tree depth 1 and 2 features, runtime is around 5 seconds and communication is 25MB for XORBoost compared to 35 seconds and 3.5GB for the passive security setting from Abspoel et al. [3, Table 1].

8.2 Comparison with plaintext algorithms

We have also ascertained that minimal predictive power is lost with respect to plaintext implementations. Since there is no unique minimum loss model, implementation decisions such as how the bucketing is performed result in different models even when comparing plaintext models. We compared the $L2$ loss of the predictions made by `XORboost` and by several well-known plaintext implementations on the training dataset: `scikit-learn` (with and without bucketing), `xgboost`, and `lightgbm`. We generated 50 different datasets using sklearn's `make_regression` functionality with parameters: `n_samples=5000, n_features=30, n_informative=20, bias=0, noise=1`. We trained the different algorithms on these datasets and computed the training loss. For the ith dataset, let $\texttt{minLoss}_i$ be

2 We use their benchmark numbers divided by a factor of 3 to reflect the difference of hardware



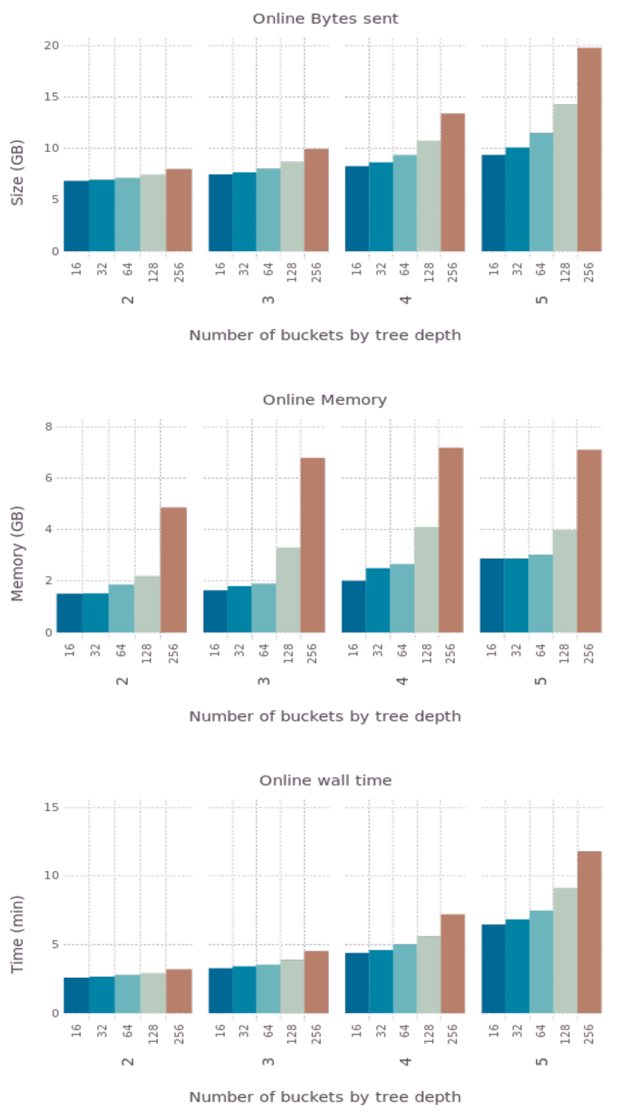


Fig. 1. Network Size, RAM usage and Wall time for depth 4, 64 buckets, 10 trees, $N \in \{5K, 25K, 100K\}$ samples and $k \in \{10, 300\}$ features.

Fig. 2. Network size, Memory and Wall time for 2 players working on a dataset of dimension $20K \times 300$ and a model with 10 trees.

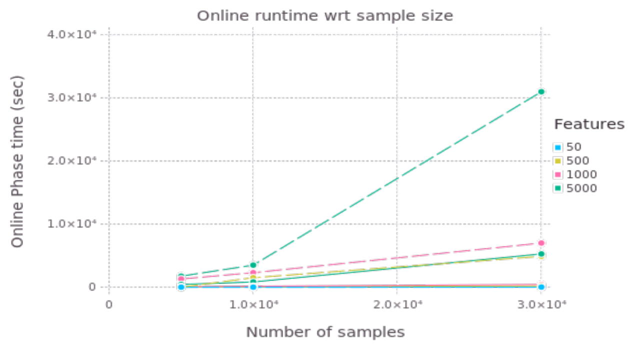


Fig. 3. Runtime comparison with [12] (dashed line) and our work (full line)

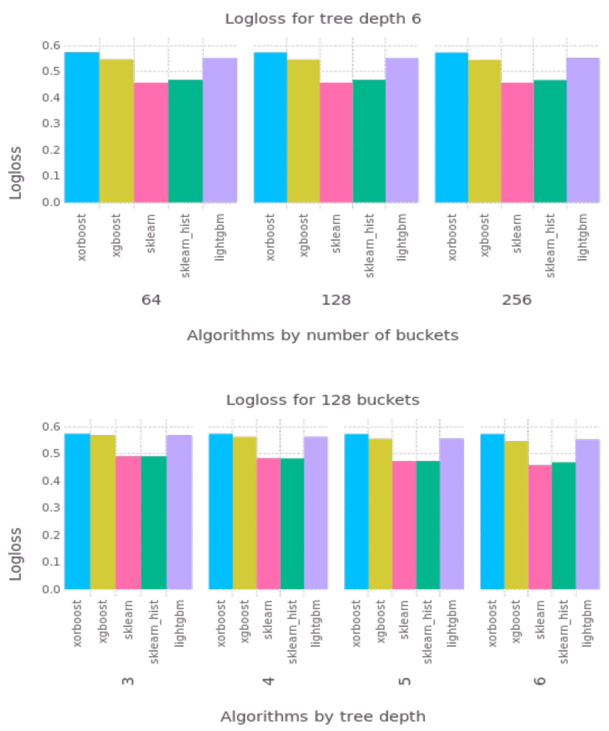


Fig. 4. Logloss comparison with plaintext implementations

smallest loss value across all algorithms and $\texttt{maxLoss}_i$ be the largest loss value. Averaging over all 50 datasets, `maxLoss` is 12% higher than `minLoss` and `XORBoost`'s loss is 6% higher than `minLoss`. We conclude from this that `XORBoost` behaves similarly to other gradient boosting implementations with respect to predictive power. Note that the `sklearn` algorithm has 2 versions: one with bucketing and one without. The version without bucketing will typically outperform other algorithms in terms of performance, and XORboost behaves similarly to other algorithms with bucketing, as highlighted by Figure 4. This indicates that the bucketing strategy results in a minimal loss of predictive power.

We have made a comparison between our implementation and the `xgboost` library on a publicly available dataset related to credit fraud found on Kaggle. We reduced the size of the original dataset, by selecting at random 100 genuine transactions for each of the 492 fraudulent transactions, yielding a new dataset with 99 % of frauds. The dataset, which has 29 features, was further split into train and test with ratio 80% / 20%. The training dataset has 39,753 samples and the testing dataset has 9,939 samples. Using 50 trees, a depth of 6 and 256 buckets, the results were very similar, both yielding an AUC of 0.98. See Table 2 for the respective confusion matrices. More information can be found on this blogpost.

	Predicted Genuine		Predicted Fraud	
Total=9939	XORBoost	XGBoost	XORBoost	XGBoost
Genuine	9834	9837	7	4
Fraud	14	14	84	84

Table 2. Confusion matrix for `XORBoost` (this work) and `XGBoost` [22].

9 Conclusion

We presented an efficient protocol for gradient boosting in the multiparty computation setting. Our protocol supports both training and prediction for generically splitted datasets. The models produced by this protocol are comparable in accuracy to their plaintext equivalents. Moreover, the time/memory/storage resources required to train the model remain reasonable, even for sizable datasets. This is all done without needing to reveal any information during the processing.

Acknowledgements

We are grateful to Joan Feigenbaum, Jonathan Katz and Nigel Smart for the numerous helpful discussions on the subject.

References

[1] Scale and mamba. https://github.com/KULeuven-COSIC/SCALE-MAMBA.

[2] XGBoost: eXtreme Gradient Boosting. https://github.com/dmlc/xgboost.

[3] Mark Abspoel, Daniel Escudero, and Nikolaj Volgushev. Secure training of decision trees with continuous attributes. Cryptology ePrint Archive, Report 2020/1130, 2020. https://eprint.iacr.org/2020/1130.

[4] S. Adams, C. Choudhary, M. De Cock, R. Dowsley, D. Melanson, A. C. A. Nascimento, D. Railsback, and J. Shen. Privacy-preserving training of tree ensembles over continuous data. *IACR Cryptol. ePrint Arch.*, page 754, 2021.

[5] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

[6] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

[7] C. Boura, I. Chillotti, N. Gama, D. Jetchev, S. Peceny, and A. Petric. High-precision privacy-preserving real-valued function evaluation. *IACR Cryptology ePrint Archive*, 2017:1234, 2017.

[8] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Sublinear gmw-style compiler for mpc with preprocessing. In *Annual International Cryptology Conference*, pages 457–485. Springer, 2021.

[9] Sergiu Carpov, Kevin Deforth, Nicolas Gama, Mariya Georgieva, Dimitar Jetchev, Jonathan Katz, Iraklis Leontiadis, M. Mohammadi, Abson Sae-Tang, and Marius Vuille. Manticore: Efficient framework for scalable secure multiparty computation protocols. Cryptology ePrint Archive, Report 2021/200, 2021. https://eprint.iacr.org/2021/200.

[10] S. Chatel, A. Pyrgelis, J. R. Troncoso-Pastoriza, and J.-P. Hubaux. Sok: Privacy-preserving collaborative tree-based model learning. *Proc. Priv. Enhancing Technol.*, 2021(3):182–203, 2021.

[11] T. Chen and C. Guestrin. XGBoost, a scalable tree boosting system. In *Proceedings of the 22 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2016, San Francisco, California, United States, August, 2016*, pages 785–794. ACM, 2016.

[12] K. Cheng, T. Fan, Y. Jin, Y. Liu, T. Chen, and Q. Yang. SecureBoost: A lossless federated learning framework. *CoRR*, abs/1901.08755, 2019.

[13] Philip Chou. Optimal partitioning for classification and regression trees. *IEEE transactions on pattern analysis and machine intelligence*, 13(4):340–354, 1991.

[14] Martine De Cock, Rafael Dowsley, Caleb Horst, Raj Katti, Anderson C. A. Nascimento, Stacey C. Newman, and Wing-Sea Poon. Efficient and private scoring of decision trees, support vector machines and logistic regression models based on pre-computation. Cryptology ePrint Archive, Report 2016/736, 2016. https://eprint.iacr.org/2016/736.

[15] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD$\mathbb{Z}_{2^k}$: Efficient mpc mod 2^k for dishonest majority. In *Advances in Cryptology – CRYPTO 2018*, pages 769–798.

[16] I. Damgård, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[17] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. Cryptology ePrint Archive, Report 2019/599, 2019. https://eprint.iacr.org/2019/599.

[18] S de Hoogh, B Shoenmarkers, P Chen, and H op den Akker. Practical secure decision tree learning in a teletreatment application. In *International Conference on Financial Cryptography and Data Security*, pages 179–194. Springer, 2014.

[19] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In *40th Annual International Cryptology Conference, CRYPTO*, volume 12171 of *Lecture Notes in Computer Science*, pages 823–852, 2020.

[20] W. Fang, C. Chen, J. Tan, C. Yu, Y. Lu, L. Wang, L. Wang, J. Zhou, and A. X. A hybrid-domain framework for secure gradient tree boosting. *CoRR*, abs/2005.08479, 2020.

[21] Z. Feng, H. Xiong, C. Song, S. Yang, B. Zhao, L. Wang, Z. Chen, S. Yang, L. Liu, and J. Huan. Securegbm: Secure multi-party gradient boosting. *CoRR*, abs/1911.11997, 2019.

[22] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[23] M. Keller. MP-SPDZ: A versatile framework for multi-party computation. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1575–1590, 2020.

[24] M. Keller, E. Orsini, and P. Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842, 2016.

[25] M. Keller, V. Pastro, and D. Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT 2018*, volume 10822 of *Lecture Notes in Computer Science*, pages 158–189, 2018.

[26] Andrew Law, Chester Leung, Rishabh Poddar, Raluca Ada Popa, Chenyu Shi, Octavian Sima, Chaofan Yu, Xingmeng Zhang, and Wenting Zheng. Secure collaborative training and inference for xgboost, 2020.

[27] C. Leung. Towards privacy-preserving collaborative gradient boosted decision trees. 2020.

[28] Y. Liu, Y. Liu, Z. Liu, J. Zhang, C. Meng, and Y. Zheng. Federated forest. *CoRR*, abs/1905.10053, 2019.

[29] Y. Liu, Z. Ma, X. Liu, S. Ma, S. Nepal, R. Deng, and K. Ren. Boosting privately: Federated extreme gradient boosting for mobile crowdsensing. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020, Singapore, November 29 - December 1, 2020*, pages 1–11. IEEE, 2020.

[30] Y. Liu, Z. Ma, X. Liu, S. Ma, S. Nepal, R. Deng, and K. Ren. Boosting privately: Federated extreme gradient boosting for mobile crowdsensing. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020, Singapore, November 29 - December 1, 2020*, pages 1–11. IEEE, 2020.

[31] X. Meng and J. Feigenbaum. Privacy-preserving xgboost inference. *CoRR*, abs/2011.04789, 2020.

[32] P. Mohassel and P. Rindal. Aby^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 35–52. ACM, 2018.

[33] P. Mohassel and Y. Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 19–38. IEEE Computer Society, 2017.

[34] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. Aby2. 0: Improved mixed-protocol secure two-party computation. In *30th USENIX Security Symposium*, 2021.

[35] L. Prokhorenkova, G. Gusev, and A. Gulip A. Vorobev, A. Dorogush. CatBoost: unbiased boosting with categor-

ical features. In *Proceedings of the Advances in Neural Information Processing Systems 31 NEURIPS*, 2018.

[36] Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *Proceedings on Privacy Enhancing Technologies*, 2019(3):26–49, 2019.

[37] WeBank. FATE: an industrial grade federated learning framework. (accessed March 2, 2021).

[38] F. Yamamoto, L. Wang, and S. Ozawa. New approaches to federated xgboost learning for privacy-preserving data analysis. In Haiqin Yang, Kitsuchart Pasupa, Andrew Chi-Sing Leung, James T. Kwok, Jonathan H. Chan, and Irwin King, editors, *Neural Information Processing - 27th International Conference, ICONIP 2020, Bangkok, Thailand, November 23-27, 2020, Proceedings, Part II*, volume 12533 of *Lecture Notes in Computer Science*, pages 558–569. Springer, 2020.

[39] M. Yang, L. Song, J. Xu, C. Li, and G. Tan. The tradeoff between privacy and accuracy in anomaly detection using federated xgboost. *CoRR*, abs/1907.07157, 2019.

[40] L. Zhao, L. Ni, S. Hu, Y. Chen, P. Zhou, F. Xiao, and L. Wu. Inprivate digging: Enabling tree-based distributed data mining with differential privacy. In *2018 IEEE Conference on Computer Communications, INFOCOM 2018, Honolulu, HI, USA, April 16-19, 2018*, pages 2087–2095. IEEE, 2018.

A Appendix: Gradient and Hessian of $\mathcal{L}$

Proof. (Lemma 2.1) In a training context, the dataset X and y are constant, and since we are doing a gradient descent to train the weights of $\mathtt{Tree}^{(T+1)}$, so all previous trees (structure and weights) $\mathtt{Tree}^{(1)}, \dots, \mathtt{Tree}^{T}$ are fixed, as well as the structure of the current tree. The only free variables that remain are the tree weights: $(w_1, ..., w_L)$ associated to the leaves of $\mathtt{Tree}^{(T+1)}$.

For a sample $i \in [1, N]$ and a leaf $j \in [1, L]$, let $\delta_{i\in n_j}$ be the Kronecker symbol of the partition induced by the structure of $\mathtt{Tree}^{(T+1)}$:

$$\delta_{i\in n_j} = \begin{cases} 1 \text{ iff. } \mathtt{Tree}(x_i) \text{ ends in leaf } n_j \\ 0 \text{ otherwise.} \end{cases}$$

For all sample $i \in [1, N]$, the evaluation function rewrites as:

$$\mathtt{eval}_{x_i}(\mathtt{Tree}^{(T+1)}) = \sum_{j=1}^{2^d} w_j \delta_{i\in n_j}$$

in particular, this implies:

$$\frac{\partial \mathtt{eval}_{x_i}}{\partial w_j}(w_1, ..., w_{2^d}) = \delta_{i\in n_j} \text{ is constant.}$$

Applying it to the loss function $\mathcal{L}$ of Eq (3), and since y_i, $\widehat{y_i}^{(T)}$ are all consistent, we deduce:

$$\frac{\partial \mathcal{L}}{\partial w_j} = \sum_{i=1}^{N} \partial_b \,\mathtt{loss}(y_i, \widehat{y_i}^{(T)} + \mathtt{eval}_{x_i}(\mathtt{Tree}^{(T+1)}))\cdot$$
$$\cdot \frac{\partial \mathtt{eval}_{x_i}}{\partial w_j}(\mathtt{Tree}^{(T+1)}) + \frac{\partial \mathrm{Reg}}{\partial w_j} =$$
$$= \sum_{i=1}^{N} \partial_b \,\mathtt{loss}(y_i, \widehat{y_i}^{(T)} + \mathtt{eval}_{x_i}(\mathtt{Tree}^{(T+1)}))\cdot \delta_{i\in n_j} + \lambda w_j.$$

And thus, the second derivative across w_i and w_k, we get:

$$\frac{\partial^2 \mathcal{L}}{\partial w_j \partial w_k} = \sum_{i=1}^{N} \partial_b^2 \,\mathtt{loss}(y_i, \widehat{y_i}^{(T)} + \mathtt{eval}_{x_i}(\mathtt{Tree}^{(T+1)}))\cdot$$
$$\cdot \delta_{i\in n_j}\delta_{i\in n_k} + \lambda\delta_{j,k}.$$

All second derivatives across 2 different variables are zero, so the hessian of $\mathcal{L}$ is a pure Diagonal. Applied to the zero weights (i.e. $\mathtt{eval}_{x_i}(\mathtt{Tree}^{(T+1)} = 0)$, the gradient and hessian are:

$$\frac{\partial \mathcal{L}}{\partial w_j}(0, \dots, 0) = \sum_{i=1}^{N} g_i \delta_{i\in n_j} = G$$
$$\frac{\partial^2 \mathcal{L}}{\partial w_j^2}(0, \dots, 0) = \sum_{i=1}^{N} h_i \delta_{i\in n_j} + \lambda = H + \lambda$$

which concludes the proof of Lemma 2.1. ■

B Appendix: Proof Lemma 3.1

Proof. (Lemma 3.1)

Let $b \in \{1, \dots, B-1\}$ be an input for Algorithm 1 and let res_m be the state of variable res in the mth iteration of the for-loop on line 3 of Algorithm 1, with $\mathrm{res}_0 = 0_N$ the initial value. Now, let $i \in \{1, \dots, N\}$ and let $b_i \in \{0, \dots, B-1\}$, $r_i \in \{0, \dots, N/B - 1\}$, such that

$$i - 1 = b_i \cdot N/B + r_i.$$

By definition of BV (11), it suffices to prove that we have

$$(\mathrm{res}_s)_i = 1 \iff b_i < b. \tag{25}$$

Recall that by definition (12), for any $m \in \{1, \dots, s\}$, we have

$$C_i^{(m)} = \neg\, [b_i]_{m-1}\,. \tag{26}$$

Substituting (26) in line 3 of the algorithm yields

$$(\mathrm{res}_m)_i = \begin{cases} (\mathrm{res}_{m-1})_i \ \texttt{OR} \ \neg\, [b_i]_{m-1} & \text{if } [b]_{m-1} = 1 \\ (\mathrm{res}_{m-1})_i \ \texttt{AND} \ \neg\, [b_i]_{m-1} & \text{else.} \end{cases} \tag{27}$$

Note that whenever $[b_i]_{m-1} = [b]_{m-1}$, we can substitute in (27):

$$(\mathrm{res}_m)_i = (\mathrm{res}_{m-1})_i\,.$$

Hence, if we define m_* as the maximal index such that $[b_i]_{m_*-1} \neq [b]_{m_*-1}$, with the convention of $m_* = 0$, if $b_i = b$, we find that

$$(\mathrm{res}_s)_i = (\mathrm{res}_{m_*})_i\,. \tag{28}$$

Finally, one observes that (25) holds for each of the three possible cases which concludes the proof:

case $b_i > b$: we find $1 = [b_i]_{m_*-1} > [b]_{m_*-1} = 0$ and thus

$$(\mathrm{res}_s)_i \overset{(28)}{=} (\mathrm{res}_{m_*})_i \overset{(27)}{=} (\mathrm{res}_{m_*-1})_i \ \texttt{AND} \ 0 = 0;$$

case $b_i = b$: we have

$$(\mathrm{res}_s)_i \overset{(28)}{=} (\mathrm{res}_0)_i = (0_N)_i = 0;$$

case $b_i < b$: we find $0 = [b_i]_{m_*-1} < [b]_{m_*-1} = 1$ and thus

$$(\mathrm{res}_s)_i \overset{(28)}{=} (\mathrm{res}_{m_*})_i \overset{(27)}{=} (\mathrm{res}_{m_*-1})_i \ \texttt{OR} \ 1 = 1.$$

■

C Appendix: Benchmark with 2 and 3 players

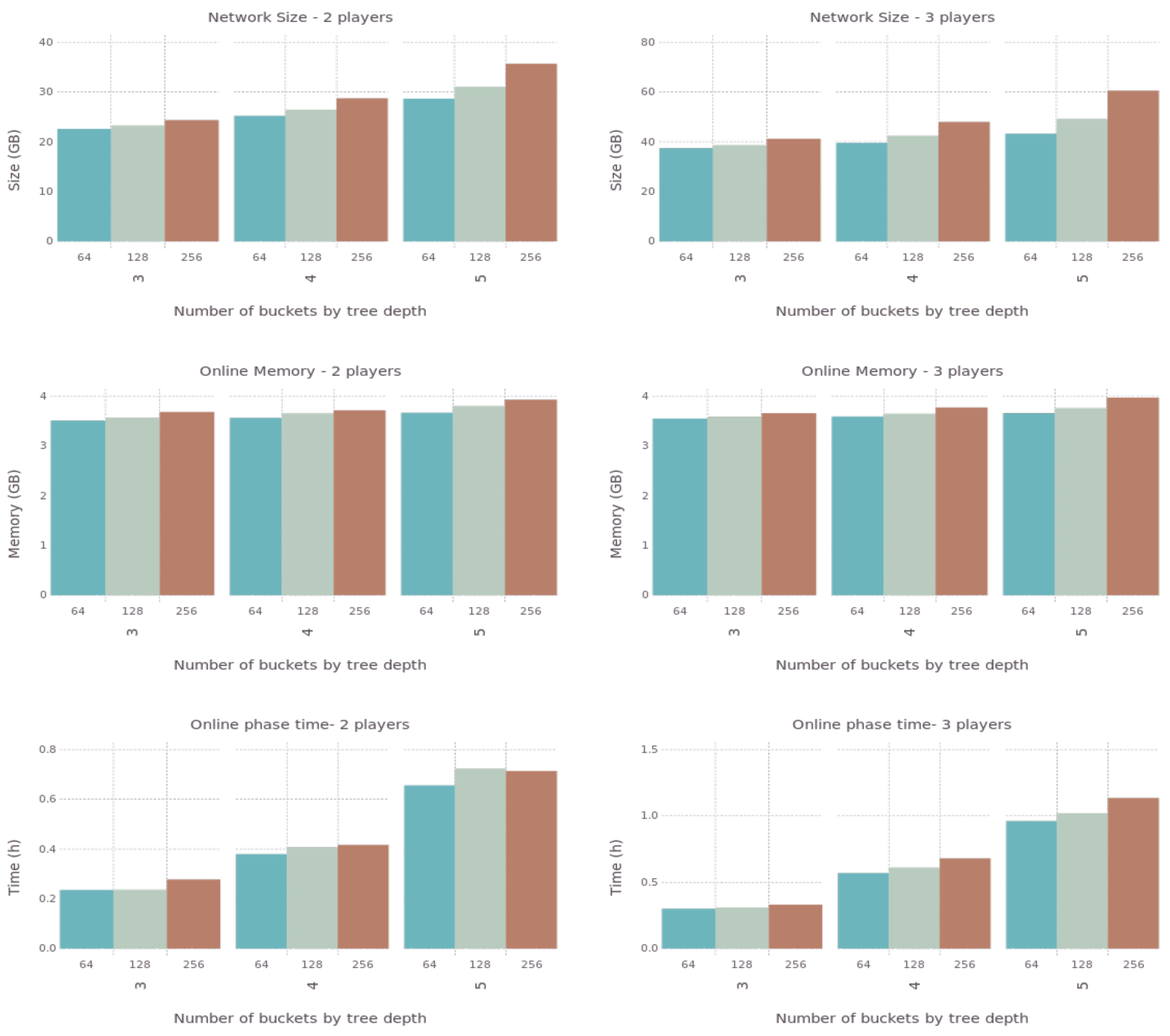


Fig. 5. Network Size, RAM usage and Wall time for 10 trees for a dataset of size $80K \times 300$