XORBoost: Tree Boosting in the Multiparty Computation Setting

Kevin Deforth[1], Marc Desgroseilliers[1], Nicolas Gama[1], Mariya Georgieva[1], Dimitar Jetchev[1], Marius Vuille[1]

Inpher

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. Similarly to [2], our protocol supports training for generically split datasets (vertical and horizontal splitting, or combination of those) while keeping all the information about the features and thresholds associated with the nodes private, thus, having only the depths and the number of the binary trees as public parameters of the model. By using optimization techniques reducing the number of oblivious permutation evaluations as well as the quicksort and real number arithmetic algorithms from the recent Manticore MPC framework [5], we obtain a scalable implementation operating under information-theoretic security model in the honest-but-curious setting with a trusted dealer. On a training dataset of 25,000 samples and 300 features in the 2-player setting, we are able to train 10 regression trees of depth 4 in less than 1.5 minutes per tree (using histograms of 128 bins).

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 [14]. XGBoost [1], [6] is currently one of the most popular open-source libraries supporting gradient boosting for various programming environments and architectures.

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 among which machine and statistical learning [3], [4], [5], [10], [12], [15], [16], [17], [21], [22], [24].

1.1 Prior work

Specific attempts have been made to adapt classical boosting methods to privacy-preserving settings, both for training and inference [7], [13], [18], [19], [20].

In [7] and [13], 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, all the data 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). This limitation is not present in our work, where any data partition type is supported.

A different method for training that does not require vertical splitting is proposed in [19]. The protocol based on federated learning, secret-sharing and homomorphic encryption reveals some information about the structure of every tree: for every internal node, the feature and threshold leading to the maximum reduction of the loss function are revealed. This potentially leaks sensitive information about the original data and that can be problematic for some use cases. Even more information about the underlying data and the trained model is revealed in [7].

Recent work on `XGBoost` inference based on fully-homomorphic encryption (FHE) is studied in [20].

A solution based on secure enclaves is proposed in [18]. As such, 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 in [18], the approach remains vulnerable to attacks targeting the secure enclave.

Finally, in [2], a protocol is presented for training and evaluating classification trees on data split horizontally or vertically (or any mixture thereof). The authors implemented their protocol on the `MP-SPDZ` framework [15]. The authors of [2] observe that the output of their chosen tree learning algorithm (C4.5 trees) only depends on the relative order of the input, which allows to map the input data to the integer domain in an arbitrary, but order-preserving manner and to skillfully avoid costly privacy-preserving operations on fixed- or floating-point numbers.

The gradient boosted tree model outlined in here is based on XGBoost [6] and is inherently different from single classification and regression trees or random forests. Whereas random forest trains many trees independently on subsampled datasets, XGBoost constitutes an ensemble of learners by, at each step, adding to the ensemble the tree with the greatest loss reduction. Furthermore the protocol outlined in this paper leverages fixed-point arithmetic, which allows to compute prediction weights accurately to train regression trees instead of being limited to classification trees with categorical response variables.

[1] The implementation of [7] has been extended to allow for horizontally split datasets as well. However, we have not found any accompanying papers Homo SecureBoost [25].

1.2 Our contributions

We present a generic algorithm to train and evaluate a gradient boosted tree model based on [6] in the MPC setting. As previously mentioned, our method supports any combination of horizontal and vertical splittings of the dataset.

In contrast to [19] and [7], the computing parties in our protocol learn only the shape of the model, i.e., the tree depth and the number of trees.

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. We also ensure that during training, no information about the first and second order statistics is revealed. Furthermore, during training and prediction, it is impossible to deduce the path taken by any sample in a decision tree. Similarly to [2], our protocol is built on generic primitives and can thus be implemented on any MPC framework supporting these primitives.

We used the Manticore MPC framework which provides access to Boolean arithmetic as well as arithmetic with real numbers represented using modular integers [5] or the prior floating-point numbers framework [4]. Manticore is operating in semi-honest security model with an offline trusted dealer, with full-threshold security across an arbitrary number of players. Furthermore all communication between the trusted dealer and the players, as well as all communication between the players during the online phase, is end-to-end encrypted. This makes it secure against malicious external adversaries.

By building on top of Manticore's representation of real numbers via modular integers, our protocol is secure in the information-theoretic setting.

Similarly to [2], Manticore allows us to privately compute the permutation sorting a given feature column. Applying these permutations is an expensive operation and we introduce in section 3.2 a novel algorithm that allows to reduce the number of permutation calls with respect to the naive algorithm.

The improvements presented in this paper allow to train a gradient boosted tree model of moderate depth within a reasonable time frame (see Section 7). These include the use of a very efficient sorting mechanism [5], the precomputation of generator vectors to apply inverse permutations (Section 3.2), the use of compressed instance vectors (Section 4.2) and storing permuted instance vectors to reduce the number of times the permutation function is called. Taken together, the number of permutation calls after an initial preprocessing phase is $2 + D$ per tree, with D the tree depth.

2 Background and Preliminaries

For a detailed review of `XGBoost`, we refer the reader to [6]. 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). Associated to each inner node is a pair (j, t) of a feature index $j \in \{1, \ldots, k\}$ and a threshold t (a real number). Associated to each `leaf` node is a weight value w (a real number). We thus represent a tree as

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

where

$$\texttt{TreeStructure} = ((j_1, t_1), \ldots, (j_{2^D-1}, t_{2^D-1}))$$

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

$$\texttt{TreeWeights} = (w_1, \ldots, w_{2^D})$$

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

$$\begin{aligned}
&\texttt{eval}(\mathbf{x}, n) = \\
&\quad \text{if } n \text{ is a leaf of weight } w: \text{ return } w \\
&\quad \text{else } n \text{ is an inner node } (j, t): \\
&\qquad \text{if } x_j < t \text{ return } \texttt{eval}(\mathbf{x}, n_{\texttt{left}}) \\
&\qquad \text{else return } \texttt{eval}(\mathbf{x}, n_{\texttt{right}}),
\end{aligned}$$

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

We also define $\texttt{eval}(\mathbf{x}, \texttt{Tree})$ to be the `eval` procedure called on $\mathbf{x}$ 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 $\texttt{Tree}(\mathbf{x})$ seen as a piecewise constant function $\texttt{Tree}\colon \mathbb{R}^k \to \mathbb{R}$, and on training scenario where $\mathbf{x}$ is fixed and the tree varies, we use $\texttt{eval}_{\mathbf{x}}(\texttt{Tree})$, which, for a fixed `TreeStructure`, is continuously differentiable over the `TreeWeights`.

If there are many samples, we write $\texttt{Tree}(X) \in \mathbb{R}^N$, to mean `Tree` evaluated at each row of X. Given a tree ensemble $\{\texttt{Tree}^{(1)}, \ldots, \texttt{Tree}^{(T)}\}$ and a learning rate parameter η, one defines the predictions on X recursively as

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

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

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

2.2 Objective function

Gradient tree boosting is an iterative process using a current prediction $\widehat{y}^{(T)}$ on T trees to greedily (see [14] for a definition) grow a new $(T+1)$th tree that most reduces a certain objective function. For a given function $\texttt{loss}\colon \mathbb{R}\times\mathbb{R}\to\mathbb{R}$ (e.g., mean-squared error or logistic loss) and a fixed training set (X, y), consider the function

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

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

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

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

The goal is to perform a greedy optimization, that is, given the ensemble $\{\texttt{Tree}^{(1)}, \ldots, \texttt{Tree}^{(T)}\}$ and the corresponding vector $\widehat{y}^{(T)}$ of predictions of the ensemble on the training data at time T, find a tree $\texttt{Tree}^{(T+1)}$ 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/binning, 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

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

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

Remark 1. 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 $f(x^{(0)})$ 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

$$f(x^{(0)} + \delta_{\min}) = f(x^{(0)}) - \frac{1}{2}\texttt{grad}(f)|^t_{x^{(0)}} \cdot \texttt{Hess}(f)|^{-1}_{x^{(0)}} \cdot \texttt{grad}(f)|_{x^{(0)}}.$$

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

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

Lemma 1. *Let $\partial_b \texttt{loss}$ and $\partial_b^2 \texttt{loss}$ be the first and second 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 $1 \leq i \leq N$. One has (see [6]):*

$$\texttt{score}(\texttt{TreeStructure}) = \sum_{\texttt{leaves } L} \frac{G_L^2}{2(H_L + \lambda)}, \tag{5}$$

where $G_L = \sum_{i \in L} g_i$ and $H_L = \sum_{i \in L} h_i$.

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:

$$\texttt{gain} = \frac{G_{n_{\text{left}}}^2}{2(H_{n_{\text{left}}} + \lambda)} + \frac{G_{n_{\text{right}}}^2}{2(H_{n_{\text{right}}} + \lambda)} - \frac{G_n^2}{2(H + \lambda)} - \gamma \tag{6}$$

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 represenation of a tree

The `TreeStructure` is represented as follows: for every internal node n, the feature index j_n is secret and is encoded as an additively secret-shared elementary vector $\mathtt{e}_n$ of size k.

If we need to access the j_nth column, we simply multiply the original dataset X with $\mathtt{e}_n$. This technique is used throughout to keep the tree structure private. The threshold $\mathtt{t}_n$ is secret and is additively secret-shared. The `TreeWeights` are secret and are additively secret-shared.

The `Tree Structure` and `Tree Weights` are secret-shared throughout training and evaluation. Several MPC protocols and libraries in the literature provide a practical method for combination of arithmetic and Boolean shares [22,21,15,12,5]. We implement `XORBoost` leveraging Manticore's [5] efficient conversions between fixed-point number representation and boolean representation, but our `xgboost` algorithm is agnostic of the choice of MPC framework. We optimize memory and runtime with the following choice: the real-valued `Tree Weights` and `thresholds` are secret-shared in the fixed-point back-end while the feature indices corresponding to the inner node splits are secret-shared in the boolean back-end as vectors of length k. More specifically, a feature-index $j \in \{1, \dots k\}$ is represented as the j-th standard unit vector in $\{0,1\}^k$. 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 overall `xgboost` training procedure is that *a priori*, the maximization of the gain is done over one discrete variable (the feature index) and one continuous variable (the threshold corresponding to that feature). To discretize the search for the threshold, we use 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 sorting permutations, as explained in Section 3.1.

The sorting permutations are required later on in the training procedure (see Algorithm 6). In Section 3.2 we introduce a novel algorithm that reduces the number of times these permutations need to be applied, thus, 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 bins in the histogram).

3.1 Sorting feature vectors and building histograms

The `Manticore` framework enables for efficient sorting of numerical vectors. It does so by applying a dealer-generated, secret-shared uniformly random permutation to the target vector, on which a variant of quicksort is applied [5, §5.2]. Throughout the computation, the input vector remains secret-shared and nothing about the underlying data is revealed. As an output, we receive the sorted

vector and / or the secret-shared sorting permutation and their inverses. 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 feature vector $X^{(j)}$ of size N, σ acts on $X^{(j)}$ by permuting the indices:

$$\sigma(X^{(j)}) := (X^{(j)}_{\sigma(i)})_{i=1}^{N}.$$

  We define the function `apply_permutation`, that takes as input a secret-shared vector v of size N, together with a list of r permutations $\sigma_1, \ldots, \sigma_r$ and returns a $N \times r$ matrix, whose jth column is the vector $\sigma_j(v)$.
- We use π_j as a label for the sorting-permutation of feature-column $X^{(j)}$, and we use π_j^{-1} to denote the inverse permutation.

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 bins B in the histogram and a feature vector $X^{(j)}$, we only consider the $B-1$ values

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

as possible threshold candidates for that feature.

Recall that an inner node for sample x is evaluated using the predicate $x_j < t$ where j is the feature index and t is the threshold value. Thus, assuming that all elements in feature column $X^{(j)}$ are unique, we have the following equality of binary vectors of size N:

$$(X^{(j)} < \boldsymbol{t}_b^{(j)}) = \pi_j^{-1}\left(\mathtt{BV}_b\right), \ \forall = 1, \ldots, B-1,$$

where

$$(\mathtt{BV}_b)_i := \begin{cases} 1 \text{ if } i \leq b\lfloor N/B \rfloor \\ 0 \text{ otherwise,} \end{cases} \qquad \forall\, i = 1, \ldots N. \tag{7}$$

We refer to $\mathtt{BV}_b$ as the bth *bucket vector* and to $\pi_j^{-1}\left(\mathtt{BV}_b\right)$ as the *selector vector* of the bth bucket for feature j.

During training, it is convenient to keep a record of the path taken by the samples in the training data, thus, requiring all $(B-1)k$ selector vectors $\pi_j^{-1}\left(\mathtt{BV}_b\right)$ (see Section 4.2.1).

3.2 Bucket vectors and permutations

Naively, one would compute the selector vectors $\pi_j^{-1}(\mathtt{BV}_b)$ for feature j and buckets $b = 1, \ldots, B-1$, with $B-1$ calls to `apply_permutation`. Yet, it is possible

to do this with only $\log_2 B$ ones. This optimization is a major contribution of the paper and leads to practical speedups.

For this, we first explain how to construct the bucket vectors $\texttt{BV}_1, \ldots, \texttt{BV}_{B-1}$ from a set of publicly known *generating vectors* via Algorithm 1. We then leverage the fact that Algorithm 1 commutes with the function `apply_permutation`, 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, we assume that B is a power of two and divisor of N, thus each bin of the histogram holds N/B samples.

For any integer $1 \leq j \leq \log_2 B$ and any $b = 0, \ldots B-1$, let $[b]_j$ be the jth least significant digit in the binary expansion of b, that is[2]

$$b = \sum_{j=1}^{\log_2 B} [b]_j 2^{j-1}, \ \forall\, b = 0, \ldots, B-1.$$

We now define a binary matrix C' of dimension $B \times \log_2 B$, such that row i corresponds to the binary expansion of $B - i$, with the least-significant bit on the left, and the most significant bit on the right, e.g., we have

$$\begin{aligned} C'_{i,j} &:= [B-i]_j, \ j = 1, \ldots, \log_2 B, \ \forall\, i = 1, \ldots, B \\ &= \neg\, [i-1]_j, \ j = 1, \ldots, \log_2 B, \ \forall\, i = 1, \ldots, B \end{aligned}$$

Further we construct the $N \times \log_2(B)$ binary matrix C from C', by repeating each row N/B times. Its columns are the generating vectors in algorithm 1:

$$\begin{aligned} C_i^{(j)} &:= C'_{1+\lfloor (i-1)B/N \rfloor, j}, && j = 1, \ldots, \log_2 B, \ i \in [1, N]. && (8) \\ &= [B - 1 - \lfloor (i-1)B/N \rfloor]_j, && j = 1, \ldots, \log_2 B, \ i \in [1, N]. && (9) \\ &= \neg\, [\lfloor (i-1)B/N \rfloor]_j, && j = 1, \ldots, \log_2 B, \ i \in [1, N]. && (10) \end{aligned}$$

For example, if $N = 32$ and $B = 8$ we have

$$C^t = \begin{pmatrix} 11110000111100001111000011110000 \\ 11111111000000001111111100000000 \\ 11111111111111110000000000000000 \end{pmatrix}.$$

[2] Note the use of 1-indexing for the bits.

Algorithm 1 `bucket_vector`

Input: This algorithm generates a bucket vector and takes as input
- the public bucket index $b \in [1, B-1]$
- The generating vectors $\{C^{(1)}, \dots, C^{(\log_2 B)}\}$ as defined in equation 8

Output: bth bucket vector $\texttt{BV}_b$

1: res $= 0_N$ (the zero vector of size N)
2: **for** $j = 1, \dots, \log_2 B$ **do**
3: res $= [b]_j$? res $\vee C^{(j)}$: res $\wedge C^{(j)}$
4: **end for**
5: **return** res

Lemma 2. *Algorithm 1 yields the bth bucket vector* $\texttt{BV}_b$.

Proof. Let $b \in [1, B-1]$ an input for algorithm 1 and let res_j be the state of variable res in the j-th iteration of the for-loop on line 3 of algorithm 1, with $\text{res}_0 = 0_N$ the initial value. Now, let $i \in [1, N]$ and let $b_i \in [0, B-1]$, $r_i \in [0, N/B-1]$, such that

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

By definition of `BV` (c.f. equation 7), it suffices to prove that we have

$$(\text{res}_{\log_2 B})_i = 1 \iff b_i < b. \tag{11}$$

Recall that by definition (c.f. equation 8), for any $j \in [1, \log_2 B]$, we have

$$C_i^{(j)} = \neg\, [b_i]_j\,. \tag{12}$$

Substituting (12) in line 3 of the algorithm yields

$$(\text{res}_j)_i = \begin{cases} (\text{res}_{j-1})_i \ \texttt{OR} \ \neg\, [b_i]_j & \text{if } [b]_j = 1 \\ (\text{res}_{j-1})_i \ \texttt{AND} \ \neg\, [b_i]_j & \text{else.} \end{cases} \tag{13}$$

Note that whenever $[b_i]_j = [b]_j$, we can substitute in equation 13:

$$(\text{res}_j)_i = (\text{res}_{j-1})_i\,.$$

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

$$\left(\text{res}_{\log_2 B}\right)_i = (\text{res}_{j_m})_i\,. \tag{14}$$

Finally, one observes that (11) holds for each of the three possible cases which concludes the proof:

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

$$\left(\text{res}_{\log_2 B}\right)_i \overset{(14)}{=} (\text{res}_{j_m})_i \overset{(13)}{=} (\text{res}_{j_m-1})_i \ \texttt{AND}\ 0 = 0;$$

case $b_i = b$: we have

$$\left(\text{res}_{\log_2 B}\right)_i \overset{(14)}{=} (\text{res}_0)_i = (0_N)_i = 0;$$

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

$$\left(\text{res}_{\log_2 B}\right)_i \overset{(14)}{=} (\text{res}_{j_m})_i \overset{(13)}{=} (\text{res}_{j_m-1})_i \texttt{ OR } 1 = 1.$$

■

3.2.2 Constructing selector vectors. MARIUS: This section is great, I think we can emphasize a bit more on how much we actually gain thanks to this trick.**MARC:** If you want to improve this section please make a concrete proposal of how you would improve it. Since Algorithm 1 only requires `AND` and `OR` operations on the generating vectors $C^{(1)}, \ldots, C^{(\log_2 B)}$, the function `apply_permutation` and Algorithm 1 commute, i.e., for each sorting permutation π_j, the selector vector $\pi_j^{-1}(\texttt{BV}_b)$ is given by

$$\begin{aligned}\pi_j^{-1}(\texttt{BV}_b) &= \pi_j^{-1}\left(\texttt{bucket\_vector}\left(b, \{C^{(1)}, \ldots, C^{(\log_2 B)}\}\right)\right) \\ &= \texttt{bucket\_vector}\left(b, \{\pi_j^{-1}(C^{(1)}), \ldots, \pi_j^{-1}(C^{(\log_2 B)})\}\right)\end{aligned}$$

Hence, if

$$C_{j,m} := \pi_j^{-1}\left(C^{(m)}\right),\ j = 1, \ldots, k,\ m = 1, \ldots, \log_2(B),$$

one can reconstruct all $(B-1)k$ selection vectors via Algorithm 2 and only require $\log_2(B)k$ calls to `apply_permutation`.

Algorithm 2 `selector_vector`

Input: This algorithm takes as input:
- Public bucket index $b \in [1, B-1]$
- Secret-shared generating vectors $\{C_{j,1}, \ldots, C_{j,\log_2 B}\}$ for feature j

Output: Secret-shared selection-vector $\pi_j^{-1}(\texttt{BV}_b)$ for the first b bins of feature j

1: res $= 0_N$ (the zero vector of size N)
2: **for** $m = 1, \ldots, \log_2 B$ **do**
3: res $:= [b]_m$? res $\vee C_{j,m}$: res $\wedge C_{j,m}$
4: **end for**
5: **return** res

4 Description of the `XORBoost` Training Algorithm

The input to the training algorithm is the feature matrix X as well as the response vector y (both secret shared among the players). 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, that is, the root node. In each iteration, we 'split' each leaf into a left and a right child via a *splitting criterion*, that is, a pair of a feature index and a threshold value, maximizing the gain.

For efficiency reasons, we have made the possible threshold values discrete by introducing the histogram/buckets in the data preprocessing phase, i.e., obliviously sorting each feature as described in Section 3.1 and building the histograms of Section 3.1.1. To compute the optimal splitting criterion in plaintext, we simply iterate over all possible splitting criteria (all feature indices and all bucket indices) and select the optimal one. This yields a $(B-1) \times k$ gain matrix (of all possible gain values). In order to adapt this simple optimization procedure to the MPC setting, we compute the optimal splitting criteria obliviously by first computing the gain matrix obliviously and then computing (also obliviously) the (secret shared) feature selector $\mathtt{e}$ and (secret shared) threshold selector $\mathbf{t}$ corresponding to the feature index j and the bucket index b of the optimal splitting criterion. Subsequently, we pick the correct generator vectors $C_{j,m}$ for $\mathtt{BV}_b$ and compute (again obliviously) the preimage $\pi_j^{-1}(\mathtt{BV}_b)$ of the bucket vector. 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 3 `xorboost_train`

Input: The training algorithm takes the following input:
- X - feature matrix of size $N \times k$ (secret shared)
- y - response vector of size N (secret shared)
- T - the size of the ensemble
- D - the depth of each binary decision tree

Output: A tree ensemble $\mathtt{ensemble} = \left\{\mathtt{Tree}^{(1)}, \ldots, \mathtt{Tree}^{(T)}\right\}$. Each tree $\mathtt{Tree}^{(t)}$ consists of the following data:
- $\mathbf{t}_n^{(t)}$ - secret shared threshold selector for each non-leaf node n
- $\mathtt{e}_n^{(t)}$ - secret shared feature selector for each non-leaf node n
- $w_\ell^{(t)}$ - secret shared weight for each leaf node ℓ
- $\mathtt{Tree}^{(t)}(X)$ - secret shared vector of predictions on the training data

1: $\widehat{y}^0 := \mathtt{initialize}(X, y)$
2: $\mathtt{ensemble} = \{\}$
3: **for** $t = 1, \ldots, T$ **do**
4: $\quad g^{(t-1)} := \left(\partial_b \, \mathtt{loss}(y_i, \widehat{y}_i^{(t-1)})\right)_{i \in [1,N]}$
5: $\quad h^{(t-1)} := \left(\partial_b^2 \, \mathtt{loss}(y_i, \widehat{y}_i^{(t-1))})\right)_{i \in [1,N]}$
6: $\quad \mathtt{Tree}^{(t)} := \mathtt{grow\_tree}(g^{(t-1)}, h^{(t-1)})$
7: $\quad$ Add $\mathtt{Tree}^{(t)}$ to $\mathtt{ensemble}$
8: **end for**
9: **return** $\mathtt{ensemble}$

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 $L2$ loss, 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 (respectively logistic regression) [5], we can use its prediction to initialize the boosted trees model in the case of $L2$ loss (respectively logistic loss). The aim here is to bootstrap the gradient boosting procedure by starting with a better initial value for $\widehat{y}^0$ and reducing 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 by 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.

4.2.1 Efficiently computing instance vectors. The main idea lies behind manipulating secret-shares of the so-called *instance vectors*. Associated to each node $n \in \mathcal{N}$ is an instance vector $\texttt{IV}_n$, that is, the binary vector of size N indicating which samples of the dataset X go through this node. To parallelize computations and reduce complexity, we also store the permuted instance matrix $\Pi(\texttt{IV}_n)$ - this is an $N \times k$ matrix.

When growing the tree, we will be working per layer. For a given depth d, let L_d be the dth layer, that is, 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.). Observe the following two basic properties:

1. $\sum_{n \in L_d} \texttt{IV}_n = 1_N$,
2. $\texttt{IV}_n = \texttt{IV}_{n_{\text{left}}} + \texttt{IV}_{n_{\text{right}}}$.

Since applying the function `apply_permutation` to a vector of size N is costly, we use an optimization technique to reduce the number of calls to this function from $O(2^D)$ to $O(D)$ (i.e., linear on the depth of the tree). More precisely, we define the *compressed instance vectors* `IVC`

$$\texttt{IVC}_{d+1} := \bigoplus_{n \in L_d} \texttt{IV}_{n_{\text{left}}},$$

where $\bigoplus$ is the logical `XOR` operator. Using Property 1. above, we see that the $\oplus$ operation in the definition is equivalent to $\vee$. This allows us to compute the oblivious permutation matrix $\Pi(\texttt{IV}_{n_{\text{left}}})$ for the left child of a given node by simply applying the `AND` operator between the permutation matrix for that node with the compressed instance vector for that level, i.e., by combining the identity

$$\texttt{IV}_{n_{\text{left}}} = \texttt{IV}_n \wedge \texttt{IVC}_{d+1} \tag{15}$$

with the commutativity of permutations and logical operations. Thus,

$$\Pi(\texttt{IV}_{n_{\text{left}}}) = \Pi(\texttt{IV}_n) \wedge \Pi(\texttt{IVC}_{d+1}) \tag{16}$$

This explains the interest in `IVC`: instead of applying Π to each $\texttt{IV}_{n_{\text{left}}}$, we can apply Π to $\texttt{IVC}_{d+1}$ and compute $\Pi(\texttt{IV}_{n_{\text{left}}})$ using a significantly cheaper $\wedge$ operation.

4.2.2 Computing gain matrices. Recall that computing `gain` (eqs. (6)) requires computing the quantities G_n and H_n for various nodes n and their left and right children. These can conveniently be written in terms of instance vectors and sorting permutations as follows:

$$G_n = \pi(g)^t \cdot \pi(\texttt{IV}_n) \qquad \text{and} \qquad H_n = \pi(h)^t \cdot \pi(\texttt{IV}_n), \tag{17}$$

where π is any permutation on N letters (in particular, any of the sorting permutation). Note that this expression does not depend on the choice of π.

We are now ready to compute the `gain` function to be used in the tree growing algorithm. Since our goal is to bucket each feature value into one of the B possible buckets, we only want to compute the gains corresponding to $B-1$ splittings at the bucket thresholds (thus, discretizing the possible splits, as explained in Section 3.1.1). As we do this for any of the k input features, we can conveniently package these gains into a $(B-1) \times k$ matrix `gains` that can be computed via (17) as

$$\begin{aligned}\texttt{gains}_n(i,j) = \texttt{score}(G_{n_{\text{left}}}(i,j), H_{n_{\text{left}}}(i,j), \lambda) + \texttt{score}(G_{n_{\text{right}}}(i,j), H_{n_{\text{right}}}(i,j), \lambda) \\ - \texttt{score}(G_n, H_n, \lambda) - \gamma,\end{aligned}$$

for $i = 1, \ldots, B-1$ and $j = 1, \ldots, k$, where

$$G_{n_{\text{left}}} = (\Pi(g) \odot \Pi(\texttt{IV}_n))^t \cdot \texttt{BM}, \qquad G_{n_{\text{right}}} = (\Pi(g) \odot \Pi(\texttt{IV}_n))^t \cdot \neg\texttt{BM},$$

$$H_{n_{\text{left}}} = (\Pi(h) \odot \Pi(\texttt{IV}_n))^t \cdot \texttt{BM}, \qquad H_{n_{\text{right}}} = (\Pi(h) \odot \Pi(\texttt{IV}_n))^t \cdot \neg\texttt{BM},$$

and

$$G_n = G_{n_{\text{left}}} + G_{n_{\text{right}}}, \qquad H_n = H_{n_{\text{left}}} + H_{n_{\text{left}}},$$

where $\odot$ denotes the Hadamard product (i.e., elementwise product) and $\texttt{BM}$ is the concatenation of the bucket vectors: $\texttt{BM} = [\texttt{BV}_1 \,|\, \ldots \,|\, \texttt{BV}_{B-1}]$ (a N-by-$(B-1)$ matrix). See Section 2 for the definition of the score function and an explanation of the gain terminology. Note that G_n and H_n are constant matrices, hence we drop the indices for simplicity and treat them as scalars.

Algorithm 4 `gain_matrix`

Input: This algorithm takes as input
- $N \times k$ matrices $\Pi(g)$ and $\Pi(h)$ for the first- and second-order statistics vectors
- Permutation matrix $\Pi(\texttt{IV}_n)$ for node n

Output: The $(B-1) \times k$ gain matrix $\texttt{gains}_n$ for node n, as well as the scalars G_n and H_n

1: $G_{n_{\text{left}}} = (\Pi(g) \odot \Pi(\texttt{IV}_n))^t \cdot \texttt{BM}$
2: $G_{n_{\text{right}}} = (\Pi(g) \odot \Pi(\texttt{IV}_n))^t \cdot \neg\texttt{BM}$
3: $H_{n_{\text{left}}} = (\Pi(h) \odot \Pi(\texttt{IV}_n))^t \cdot \texttt{BM}$
4: $H_{n_{\text{right}}} = (\Pi(h) \odot \Pi(\texttt{IV}_n))^t \cdot \neg\texttt{BM}$
5: $G_n = G_{n_{\text{left}}} + G_{n_{\text{right}}}$
6: $H_n = H_{n_{\text{left}}} + H_{n_{\text{right}}}$
7: **return** $\texttt{score}(G_{n_{\text{left}}}, H_{n_{\text{left}}}, \lambda) + \texttt{score}(G_{n_{\text{right}}}, H_{n_{\text{right}}}, \lambda) - \texttt{score}(G_n, H_n, \lambda) - \gamma$, G_n, H_n

Algorithm 4 summarizes the above computation. Note that in Step 7, we have overloaded notation so that $\texttt{score}(G_{n_{\text{left}}}, H_{n_{\text{left}}}, \lambda)$ is now a $(B-1) \times k$ matrix.

Remark 2. By tracking a 0-1 matrix X_{nan} of missing values in the initial dataset, we can adapt the algorithm to handle missing values. Let

$$\Pi(X_{nan}) = [\pi_1(X^{(1)}), \ldots, \pi_k(X^{(k)})].$$

We can consider $\Pi(\texttt{IV}_{n,nan}) = \Pi(\texttt{IV}_n)\,\texttt{XOR}\,\Pi(X_{nan})$. We then obtain two gain matrices, one where $G_{n_{\text{left}}}$ and $H_{n_{\text{left}}}$ are computed using $\Pi(\texttt{IV}_{n,nan})$ instead of $\Pi(\texttt{IV}_n)$ and another where it is $G_{n_{\text{right}}}$ and $H_{n_{\text{right}}}$ that use $\Pi(\texttt{IV}_{n,nan})$. We can then proceed to take the argmax over the concatenation of these two matrices.

4.3 Tree growing

The algorithm will split the layers until the desired depth D and compute the leaf weights to obtain the predictions $\texttt{Tree}(X)$. The threshold selectors $\mathbf{t}_n$ and feature selectors $\mathtt{e}_n$ are computed during the layer splitting whereas the weights $w = (w_\ell)_{\ell \in L_D}$ are computed once the tree has reached its full depth. We denote by $\mathbf{T} = \{\mathbf{t}_n \colon n \in [1, 2^D - 1]\}$ the set of thresholds of $\texttt{Tree}$ and by $\mathbf{E} = \{\mathtt{e}_n \colon n \in [1, 2^D - 1]\}$ the set of feature selectors. The data $\{\mathbf{T}, \mathbf{E}\}$ constitutes the tree structure and is secret-shared.

Algorithm 5 `grow_tree`

Input: As input, this algorithm takes
- First and second order statistic vectors g and h
- Depth D of the tree to grow

Output: A tree $\{\mathbf{T}, \mathbf{E}, w\}$

$\Pi(g) \leftarrow \texttt{apply\_permutation}(g)$
$\Pi(h) \leftarrow \texttt{apply\_permutation}(h)$
$\texttt{IV}_{Root} \leftarrow 1_{N \times 1}$
$\Pi(\texttt{IV}_{Root}) \leftarrow 1_{N \times k}$
$\mathbf{T}, \mathbf{E}, \{G_n \colon n \in [1, 2^D - 1]\}, \{H_n \colon n \in [1, 2^D - 1]\}, \{\texttt{maxGain}_n \colon n \in [1, 2^D - 1]\}, \{\texttt{IV}_\ell \colon \ell \in L_D\} \leftarrow \texttt{split\_layer}\,(1, \mathbf{T} = \emptyset, \mathbf{E} = \emptyset, \Pi(g), \Pi(h), \{\texttt{IV}_{Root}\}, \{\Pi(\texttt{IV}_{Root})\}, D)$
for $\ell \in L_D$ **do**
    $w_\ell \leftarrow \texttt{privatedivide}(-g^T \cdot \texttt{IV}_\ell, h^T \cdot \texttt{IV}_\ell + \lambda)$
end for
$\texttt{Tree}(X) \leftarrow \sum_{\ell \in L_D} w_\ell \cdot \texttt{IV}_\ell$
return $\mathbf{T}, \mathbf{E}, w = (w_\ell)_{\ell \in L_D}, \texttt{Tree}(X)$

4.4 Splitting a layer

We present a detailed description of the algorithm that, given the dth layer of a tree, adds a new layer to a tree by splitting each node at depth d into a left and right child node at depth $d + 1$. Note that all the output data is secret-shared.

Algorithm 6 `split_layer` at depth d

Input: d - current depth, $\mathbf{T}$, $\mathbf{E}$, $\Pi(g)$, $\Pi(h)$, $\{\texttt{IV}_n : n \in L_d\}$, $\{\Pi(\texttt{IV}_n) : n \in L_d\}$, D - final depth

Output: $\mathbf{T}$, $\mathbf{E}$, $\{\texttt{IV}_n : n \in L_{d+1}\}$, $\{\Pi(\texttt{IV}_n) : n \in L_{d+1}\}$

for $n \in L_d$ **do**
  $\texttt{gains}_n, G_n, H_n \leftarrow \texttt{gain\_matrix}(\Pi(g), \Pi(h), \Pi(\texttt{IV}_n))$
  $\texttt{maxGain}_n, \mathbf{t}_n, \mathbf{e}_n, b, j \leftarrow \texttt{maxAndArgmax}(\texttt{gains}_n)$
  $\texttt{Selector} \leftarrow \texttt{selector\_vector}(\{C_{m,j} : 1 \leq m \leq \log_2 B\}, b)$
  $\texttt{IV}_{n_{\text{left}}} \leftarrow \texttt{Selector} \wedge \texttt{IV}_n$
  $\texttt{IV}_{n_{\text{right}}} \leftarrow \neg\texttt{Selector} \wedge \texttt{IV}_n$
end for
$\mathbf{T} \leftarrow \mathbf{T} \cup \{\mathbf{t}_n : n \in L_d\}$, $\mathbf{E} \leftarrow \mathbf{E} \cup \{\mathbf{e}_n : n \in L_d\}$
$\{G_n : n \in L_0 \cup \cdots \cup L_d\} \leftarrow \{G_n : n \in L_0 \cup \cdots \cup L_{d-1}\} \cup \{G_n : n \in L_d\}$
$\{H_n : n \in L_0 \cup \cdots \cup L_d\} \leftarrow \{H_n : n \in L_0 \cup \cdots \cup L_{d-1}\} \cup \{H_n : n \in L_d\}$
$\{\texttt{maxGain}_n : n \in L_0 \cup \cdots \cup L_d\} \leftarrow \{\texttt{maxGain}_n : n \in L_0 \cup \cdots \cup L_{d-1}\} \cup \{\texttt{maxGain}_n : n \in L_d\}$
$\{\texttt{IV}_n : n \in L_{d+1}\} \leftarrow \{\texttt{IV}_{n_{\text{left}}} : n \in L_d\} \cup \{\texttt{IV}_{n_{\text{right}}} : n \in L_d\}$
$\texttt{IVC} \leftarrow \bigoplus_{n \in L_d} \texttt{IV}_{n_{\text{left}}}$
$\Pi(\texttt{IVC}) \leftarrow \texttt{apply\_permutation}(\texttt{IVC})$
for $n \in L_d$ **do**
  $\Pi(\texttt{IV}_{n_{\text{left}}}) \leftarrow \Pi(\texttt{IVC}) \wedge \Pi(\texttt{IV}_n)$
  $\Pi(\texttt{IV}_{n_{\text{right}}}) \leftarrow \neg\Pi(\texttt{IVC}) \wedge \Pi(\texttt{IV}_n)$
end for
$\{\Pi(\texttt{IV}_n) : n \in L_{d+1}\} \leftarrow \{\Pi(\texttt{IV}_{n_{\text{left}}}) : n \in L_d\} \cup \{\Pi(\texttt{IV}_{n_{\text{right}}}) : n \in L_d\}$
if $d = D - 1$ **then**
  return $\mathbf{T}, \mathbf{E}, \{G_n : n \in [1, 2^D - 1]\}, \{H_n : n \in [1, 2^D - 1]\}, \{\texttt{maxGain}_n : n \in [1, 2^D - 1]\}, \{\texttt{IV}_n : n \in L_D\}$
else
  return $\texttt{split\_layer}\,(d + 1, \Pi(g), \Pi(h), \{\texttt{IV}_n : n \in L_{d+1}\}, \{\Pi(\texttt{IV})_n : n \in L_{d+1}\}, D)$
end if

Note that the argmax only returns the secret-shared vectors $\mathbf{t}$ and $\mathbf{e}$ from which we could compute b and j. This is purely for exposition since we can obtain the right generator vectors $C_{j,m}$ using $\mathbf{e}$ and matrix multiplication.

5 Prediction

Our inference protocol respects the same privacy requirements as our training protocol: only the shape of the model is public knowledge (tree depth, number of trees) everything else remains secret (threshold values, feature selectors, leaf- and prediction-values). The data used to evaluate the model remains secret. Similar to [2] we achieve this by 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 [9] and [11] for passive and active security in two player settings.

In order to improve efficiency, the communication intensive comparisons and multiplications required for securely evaluating a tree are batched in Algorithm 7. We give a brief explanation of the protocol for a batch of size one, i.e. a single tree.

Recall that each non-leaf node consists of a feature selector and a threshold value. To evaluate a non-leaf node with a sample x, the feature selector is used to extract the feature of interest from x, which is then compared against the threshold value. This can be achieved securely via a multiplication of the secret-shared feature selector and the secret-shared sample x, followed by an oblivious comparison with the threshold value, yielding a secret-shared Boolean that indicates, if the left or the right subtree is to be evaluated next. Our iterative algorithm 7 evaluates one layer at a time - starting from the bottom layer and going up until reaching the root of the tree, each time obliviously selecting between secret-shared leaf-values and reducing the number of leaf candidates by a factor of two. Note how multiplications and comparisons on lines one and two of algorithm 7 have been batched to reduce the number of communication rounds.

Algorithm 7 `xorboost_predict_batch`

Input: The prediction function for a batch of trees takes as input:
- secret-shared data matrix X' of dimensions $N' \times k$
- secret-shared tree ensemble of T trees, each tree $\texttt{Tree}^{(t)}$ of depth D and consisting of
  - threshold values $\{\mathbf{t}_n^t : n \in [1, 2^D - 1]\}$
  - feature selectors $\{\mathbf{e}_n^t : n \in [1, 2^D - 1]\}$: vectors of length k each
  - prediction weights $w^t = (w_\ell)_{\ell \in [1, 2^D]}$ (leaf-values)

Output: Predictions $\texttt{Tree}^{(t)}(X')$ for $t \in [1, T]$

Extract the feature-values and compare them to the threshold values:

1: $X'^{(t,n)} := X' \cdot \mathbf{e}_n^t$ for all $n \in [1, 2^D - 1]$ and $t \in [1, T]$
2: $\beta^{t,n} := X'^{(t,n)} < \mathbf{t}_n^t$, for all $n \in [1, 2^D - 1]$ and $t \in [1, T]$

Re-arrange the leaf values and obliviously extract the correct candidate:

3: $w^{t,2^{D-1}+\ell} = w_\ell^t \cdot \mathbb{1}_{N' \times 1}, \ell = 1, \ldots, 2^D$, for all $t \in [1, T]$
4: **for** $d = D, \ldots, 1$ **do**
5: $\quad w^{t,n} = w^{t,2n+1} + \beta^{t,n} \cdot (w^{t,2n} - w^{t,2n+1})$ for $n = 1, ..., 2^{d-1}$ and $t \in [1, T]$
6: **end for**
7: **return** $\{w^{t,1} \ : \ t \in [1, T]\}$

Evaluating a gradient boosted tree model follows naturally: The tree-ensemble is split into batches of reasonable sizes, which are evaluated independently. The predictions for each batch are then multiplied with the learning parameter η (c.f. section 2) and summed together with the initial prediction (c.f. section 4.1) as outlined in algorithm 8.

Algorithm 8 `xorboost_predict`

Input: The prediction function for the entire model takes as input:
- secret-shared data $N' \times k$ matrix X',
- learning rate parameter η
- secret-shared initial prediction value $\hat{y}^0$
- secret-shared tree ensemble of T trees, each tree $\texttt{Tree}^{(t)}$ of depth D and consisting of
  - threshold values $\{\mathbf{t}_n^t : n \in [1, 2^D - 1]\}$
  - feature selectors $\{\mathbf{e}_n^t : n \in [1, 2^D - 1]\}$: vectors of length k each
  - prediction weights $w^t = (w_\ell)_{\ell \in [1, 2^D]}$ (leaf-values)

Output: secret-shared prediction vector $\hat{y} := \hat{y}^0 + \sum_{t=1}^{T} \eta \texttt{Tree}^{(t)}(X')$

1: $\widehat{y}^1, \ldots, \widehat{y}^T := \texttt{xorboost\_predict\_batch}(X', \texttt{Tree}^{(1)}, \ldots, \texttt{Tree}^{(T)})$
2: $\hat{y} := \hat{y}^0 + \eta \sum_{t=1}^{T} \widehat{y}^t$
3: **return** $\hat{y}$

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 development in this area. For a survey of different ways to do this, see [23]. Here, we develop 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 [8], the following procedure will find the best split if the loss function is either $L2$ loss or logistic loss. It only considers $|A| - 1$ possible splits.

- For each categorical value $a \in A$, compute $y_a = \dfrac{\sum_{i, X_i^{(j)} = a} y_i}{\sum_{i, X_i^{(j)} = a} 1}$ 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 $a \in A$, consider the split S_a defined by $A_1 = \{a' : a' \preceq a\}$ and $A_2 = \{a' : a' \succ a\}$.

Let π_A be the sorting permutation for the values y_a. Given the original one hot encoded matrix M, 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. These indicator vectors for the splits S_a can be computed as row sums of the sorted matrix M_A. If we let a be the bth categorical value according to the total order $<$, we have:

$$\texttt{BV}_b = \sum_{j \leq b} M_A^{(j)} = \begin{cases} 1 \text{ if } X^{(j)} = a', a' \leq a \\ 0 \text{ otherwise,} \end{cases} \qquad \forall\, i = 1, \ldots N.$$

7 Benchmarks

All 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, all reported times do not take into account network transfer time.

7.1 Parameter scaling

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

The benchmarks presented in Figure 2 and Figure 3 are for a dataset of $N = 20K$ samples and $k = 300$ features, 2 players, and models with 5 trees.

Referring to Figure 2 we see the strong dependency on the depth and the number of buckets. The total processing time remains reasonable to grow 5 trees. Figure 3 highlights that total wall-time hovers around 1 minute per tree for depth 4 and total time grows more or less linearly with the number of trees, as expected.

If we compare our results to [7], they are similar. We obtain a running time of 7 minutes for 5 trees, dataset dimensions of 20K by 300, a tree depth of 4 and 64 buckets. They obtain a running time of around 5 minutes for a single tree with $30K$ samples, 25 features and a tree depth of 4. Delivering similar performance while not revealing any information is a significant improvement.

Although our framework leverages a bucketing strategy to handle larger datasets, Figure 4 presents a comparison with the main benchmarks of [2] where no bucketing is possible. We use a tree depth of 1 and 2 features in order to use the same parameters. For 8192 samples, end to end execution time is around 5 seconds and communication is 25MB for `XORBoost` compared to 35 seconds and 3.5GB for Table 1 in [2] for the passive security setting.

7.2 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 only 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`, `lightgbm`. We ran a simulation with 50 different datasets generated at random. If we let $\texttt{minLoss}_i$ be the loss value achieved by the best model and $\texttt{maxLoss}_i$ be the loss achieved by the worst model for the ith dataset, then

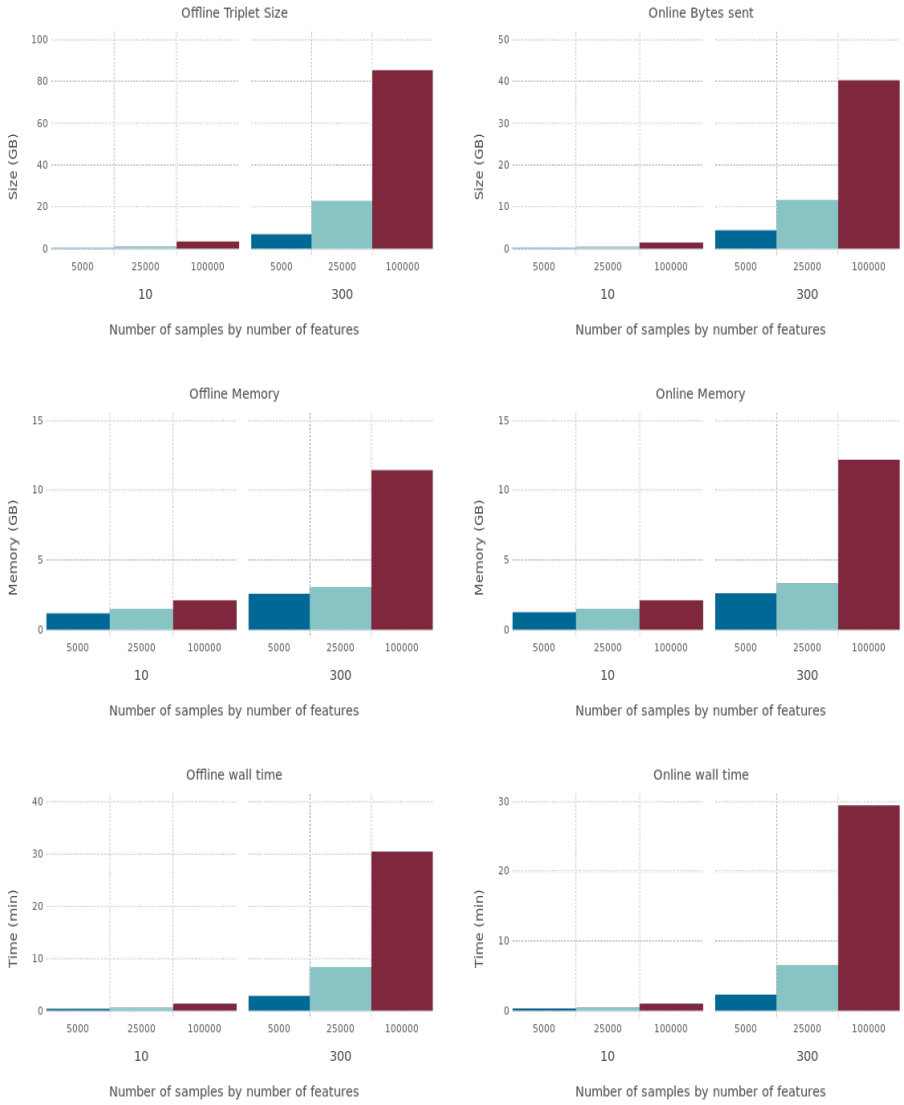

Fig. 1: Network Size, RAM usage and Wall time for depth 4, 64 buckets, 10 trees, $N \in \{5K, 25K, 100K\}$ samples and $k \in \{10, 300\}$ features.

`maxLoss` is on average 12% higher than `minLoss`. On average, `XORboost` is 6% higher than `minLoss`. We conclude from this that `XORBoost` behaves similarly to other gradient boosting implementations with respect to predictive power.

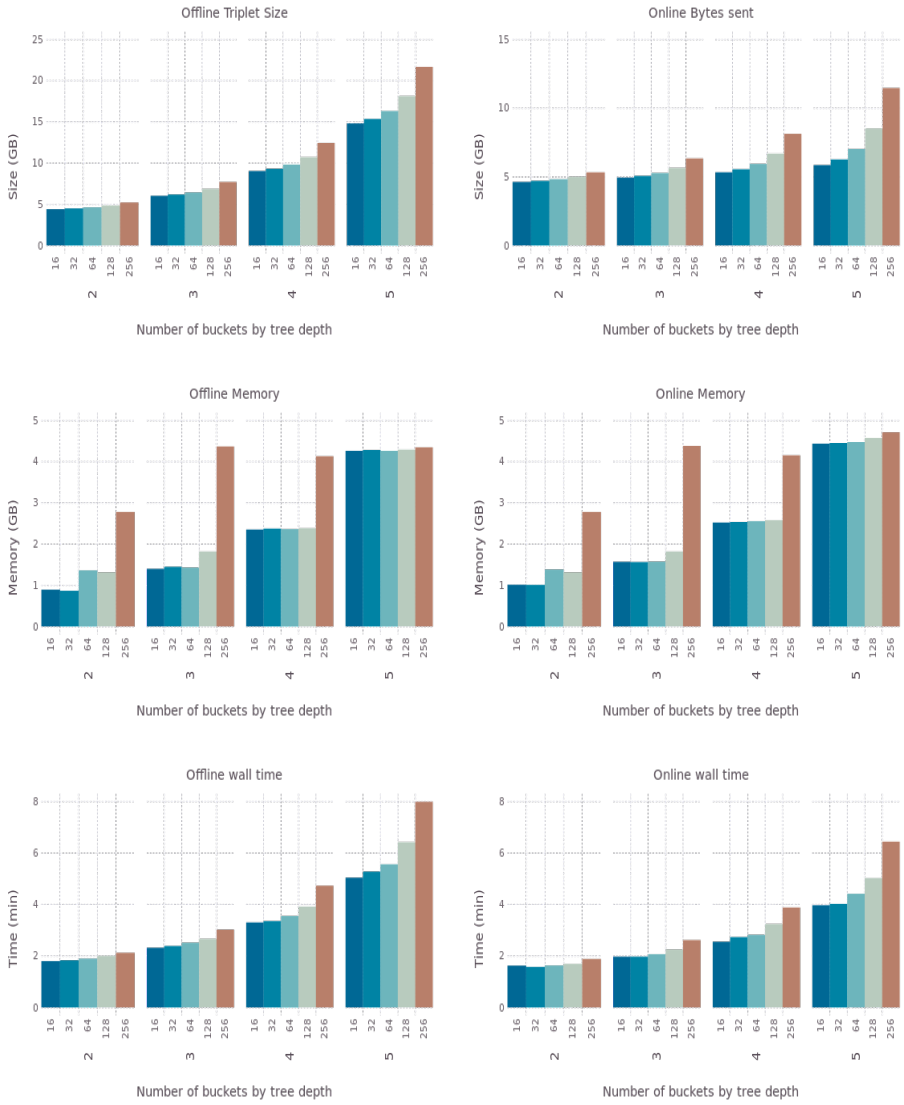

Fig. 2: Network size, Memory and Wall time for a dataset of dimension $20K \times 300$ and a model with 5 trees.

As an example, for a particular dataset, `XORBoost` achieved a loss of 58.1 (lower is better), while `sklearn` without histograms achieved 51.5, `sklearn` with histograms achieved 56.1, `lightgbm` achieved 60.8 and `xgboost` achieved 61.9.

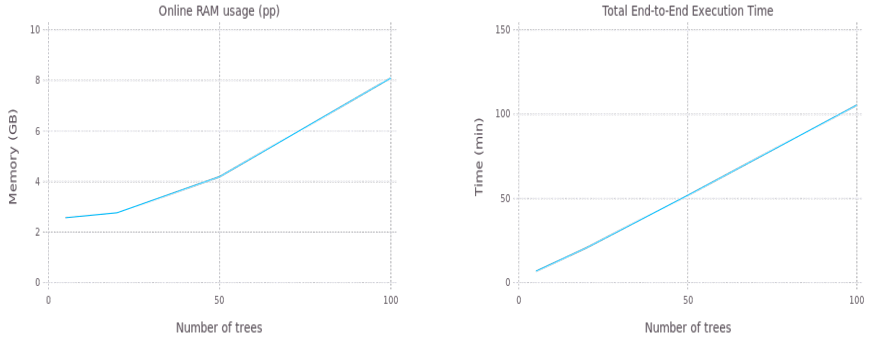

Fig. 3: Growth of RAM usage during the online phase and of total execution time for 64 buckets and a tree depth of 4.

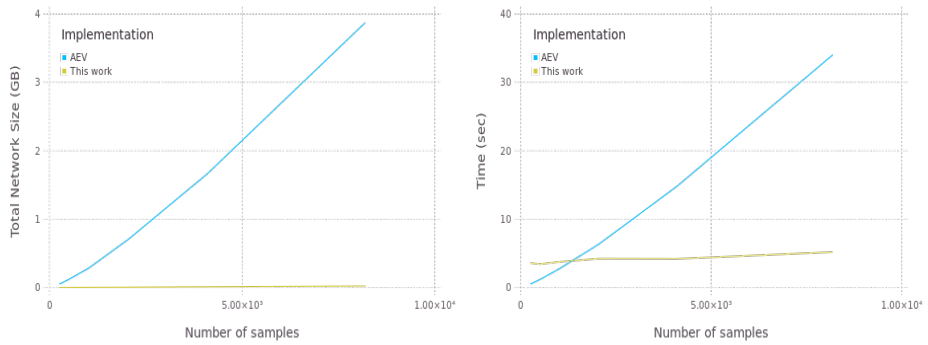

Fig. 4: Resource utilization without bucketing, comparison with (AEV) [2]

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

Proof. (Lemma 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 $\texttt{Tree}^{(T+1)}$, so all previous trees (structure and weights) $\texttt{Tree}^{(1)}, \ldots, \texttt{Tree}^{T}$ 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 $\texttt{Tree}^{(T+1)}$.

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 $\texttt{Tree}^{(T+1)}$:

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

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

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

in particular, this implies:

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

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

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

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

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

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. $\texttt{eval}_{x_i}(\texttt{Tree}^{(T+1)} = 0)$, the gradient and hessian are:

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

which concludes the proof of Lemma 1.

References

1. XGBoost: eXtreme Gradient Boosting. https://github.com/dmlc/xgboost
2. Abspoel, M., Escudero, D., Volgushev, N.: Secure training of decision trees with continuous attributes. Cryptology ePrint Archive, Report 2020/1130 (2020), https://eprint.iacr.org/2020/1130
3. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: European Symposium on Research in Computer Security. pp. 192–206. Springer (2008)

4. Boura, C., Chillotti, I., Gama, N., Jetchev, D., Peceny, S., Petric, A.: High-precision privacy-preserving real-valued function evaluation. IACR Cryptology ePrint Archive **2017**, 1234 (2017)
5. Carpov, S., Deforth, K., Gama, N., Georgieva, M., Jetchev, D., Katz, J., Leontiadis, I., Mohammadi, M., Sae-Tang, A., Vuille, M.: Manticore: Efficient framework for scalable secure multiparty computation protocols. Cryptology ePrint Archive, Report 2021/200 (2021), https://eprint.iacr.org/2021/200
6. Chen, T., Guestrin, C.: 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. pp. 785–794. ACM (2016)
7. Cheng, K., Fan, T., Jin, Y., Liu, Y., Chen, T., Yang, Q.: Secureboost: A lossless federated learning framework. CoRR **abs/1901.08755** (2019), http://arxiv.org/abs/1901.08755
8. Chou, P.: Optimal partitioning for classification and regression trees. IEEE transactions on pattern analysis and machine intelligence **13**(4), 340–354 (1991)
9. Cock, M.D., Dowsley, R., Horst, C., Katti, R., Nascimento, A.C.A., Newman, S.C., Poon, W.S.: 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
10. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Annual Cryptology Conference. pp. 643–662. Springer (2012)
11. Damgård, I., Escudero, D., Frederiksen, T., Keller, M., Scholl, P., Volgushev, N.: 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
12. Escudero, D., Ghosh, S., Keller, M., Rachuri, R., Scholl, P.: Improved primitives for MPC over mixed arithmetic-binary circuits. In: 40th Annual International Cryptology Conference, CRYPTO. Lecture Notes in Computer Science, vol. 12171, pp. 823–852 (2020)
13. Feng, Z., Xiong, H., Song, C., Yang, S., Zhao, B., Wang, L., Chen, Z., Yang, S., Liu, L., Huan, J.: Securegbm: Secure multi-party gradient boosting. CoRR **abs/1911.11997** (2019), http://arxiv.org/abs/1911.11997
14. Hastie, T., Tibshirani, R., Friedman, J.: The Elements of Statistical Learning. Springer Series in Statistics, Springer New York Inc., New York, NY, USA (2001)
15. Keller, M.: MP-SPDZ: A versatile framework for multi-party computation. In: CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security. pp. 1575–1590 (2020)
16. Keller, M., Orsini, E., Scholl, P.: Mascot: faster malicious arithmetic secure computation with oblivious transfer. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 830–842 (2016)
17. Keller, M., Pastro, V., Rotaru, D.: Overdrive: Making SPDZ great again. In: EUROCRYPT 2018. Lecture Notes in Computer Science, vol. 10822, pp. 158–189 (2018)
18. Law, A., Leung, C., Poddar, R., Popa, R.A., Shi, C., Sima, O., Yu, C., Zhang, X., Zheng, W.: Secure collaborative training and inference for xgboost (2020)
19. Liu, Y., Ma, Z., Liu, X., Ma, S., Nepal, S., Deng, R., Ren, K.: 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. pp. 1–11. IEEE (2020). https://doi.org/10.1109/ICDCS47774.2020.00017, https://doi.org/10.1109/ICDCS47774.2020.00017

20. Meng, X., Feigenbaum, J.: Privacy-preserving xgboost inference. CoRR **abs/2011.04789** (2020), https://arxiv.org/abs/2011.04789
21. Mohassel, P., Rindal, P.: Aby^3: A mixed protocol framework for machine learning. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 35–52. ACM (2018). https://doi.org/10.1145/3243734.3243760, https://doi.org/10.1145/3243734.3243760
22. Mohassel, P., Zhang, Y.: 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. pp. 19–38. IEEE Computer Society (2017). https://doi.org/10.1109/SP.2017.12, https://doi.org/10.1109/SP.2017.12
23. Prokhorenkova, L., Gusev, G., A. Vorobev, A. Dorogush, A.G.: CatBoost: unbiased boosting with categorical features. In: Proceedings of the Advances in Neural Information Processing Systems 31 NEURIPS (2018)
24. Wagh, S., Gupta, D., Chandran, N.: SecureNN: 3-party secure computation for neural network training. Proceedings on Privacy Enhancing Technologies **2019**(3), 26–49 (2019)
25. WeBank: FATE: an industrial grade federated learning framework, https://fate.fedai.org/, (accessed March 2, 2021)