Efficient decision tree training with new data structure for secure multi-party computation

Koki Hamada
NTT Social Informatics Laboratories
Musashino, Tokyo, Japan
koki.hamada.rb@hco.ntt.co.jp

Dai Ikarashi
NTT Social Informatics Laboratories
Musashino, Tokyo, Japan
dai.ikarashi.rd@hco.ntt.co.jp

Ryo Kikuchi
NTT Social Informatics Laboratories
Musashino, Tokyo, Japan
ryo.kikuchi.su@hco.ntt.co.jp

Koji Chida
Gunma University
Maebashi, Gunma, Japan
chida@gunma-u.ac.jp

ABSTRACT

We propose a secure multi-party computation (MPC) protocol that constructs a secret-shared decision tree for a given secret-shared dataset. The previous MPC-based decision tree training protocol (Abspoel et al. 2021) requires $O(2^h mn \log n)$ comparisons, being exponential in the tree height h and with n and m being the number of rows and that of attributes in the dataset, respectively. The cause of the exponential number of comparisons in h is that the decision tree training algorithm is based on the divide-and-conquer paradigm, where rows are padded after each split in order to hide the number of rows in the dataset. We resolve this issue via secure data structure that enables us to compute an aggregate value for every group while hiding the grouping information. By using this data structure, we can train a decision tree without padding to rows while hiding the size of the intermediate data. We specifically describes a decision tree training protocol that requires only $O(hmn \log n)$ comparisons when the input attributes are continuous and the output attribute is binary. Note that the order is now *linear* in the tree height h. To demonstrate the practicality of our protocol, we implement it in an MPC framework based on a three-party secret sharing scheme. Our implementation results show that our protocol trains a decision tree with a height of 4 in 404 seconds for a dataset of 2^{20} rows and 11 attributes.

KEYWORDS

secure multi-party computation, decision tree training

1 INTRODUCTION

Secure multi-party computation (MPC) [34] allows parties to jointly compute any function while keeping inputs private. Its large computational overhead has long been a barrier to practical use. In recent years, even efficient MPC protocols for machine learning methods such as neural network training [25, 29, 32] have been proposed.

Proceedings on Privacy Enhancing Technologies 2023(1), 343–364

https://doi.org/10.56553/popets-2023-0021

Decision tree is one of the classical machine learning methods. It is still widely used due to its computational simplicity and ease of interpretation. It is also an important component of other machine learning methods, such as gradient boosting decision tree [11] and random forest [5], which have been successful in recent years.

Since the work of Lindell and Pinkas [23] in the early days of privacy-preserving data mining, there has been a lot of research on MPC protocols for decision tree training. In order to be used as a component of MPC protocols for other machine learning methods, it is desirable to keep all the information, from the input to the trained decision tree, private. However, only a few protocols [1, 2, 8] with such a property have been proposed. This is mainly due to two kinds of computational difficulties in MPC.

The first difficulty is the computation on real numbers. Decision tree training requires computation of evaluation functions. Although there are many types of evaluation functions, all commonly used ones involve division or logarithm. Therefore, naive MPC protocols for decision tree training involve computation on real numbers, which increases the computational cost. On the contrary, de Hoogh et al. [8] cleverly avoided computation on real numbers by replacing fractional number comparisons with integer comparisons, and proposed an efficient protocol for the case where inputs are categorical values. Abspoel et al. [1] presented an efficient protocol that can be applied to the case where the input contains numerical values. The number of candidates to which the evaluation functions are applied is $\Theta(c)$ when the input consists only of categorical values, whereas it increases to $\Theta(n^2)$ when the input contains numerical values, where c is the number of possible values of the categorical value and n is the number of samples in the input. They used a sorting protocol to reduce the number of candidates to $O(n)$, and also extended the technique of de Hoogh et al. to the numerical case to avoid computation on real numbers. Adams et al. [2] dealt with the case where the input contains numeric values by a different approach: discretizing the numeric attributes of the input. Although the trained tree is slightly different from the one without discretization, this approach avoids the use of sorting, which is relatively computationally expensive, and allows us to use the efficient protocol of de Hoogh et al. [8].

The second difficulty is the protection of the intermediate data size. In decision tree training, the data is split recursively from the root node to the leaf nodes in a top-down fashion. As the tree height increases, the number of nodes increases exponentially. On

Table 1: Comparison of computational cost of MPC protocols for decision tree training with numerical input attributes.

Method	Number of operations	Rounds
Trivial [1]	$O(2^h mn^2)$	$O(h(\log n + \log m))$
Abspoel et al. [1]	$O((2^h + \log n)mn \log n)$	$O(h(\log n + \log m))$
Abspoel et al. [1] with sort in [13]	$O(2^h mn \log n)$	$O(h(\log n + \log m))$
Ours	$O(hmn \log n)$	$O(h(\log n + \log m))$

the other hand, the size of the intermediate data processed by each node also decreases exponentially on average, hence the overall computational cost is linear in the tree height. When this is implemented in MPC, the intermediate data size after splitting has to be hidden, so the existing protocols [1, 2, 8] used a data, which is padded, of the same size as the original one at each node. Therefore, we could not benefit from the size reduction by data splitting, and as a result, the overall computational cost was exponential in the tree height. The required tree height depends on the situation, but can be large. For example, Probst et al. [26] concluded that it is optimal to set the default maximum tree height to 21 when training decision trees using rpart. Thus, it is desirable to reduce the high dependence of the computational cost on h.

1.1 Our contribution

We propose an MPC protocol for decision tree training with linear computational cost on tree height, which is the first protocol to solve the second problem above. It trains a binary decision tree under the assumption that all input attributes are numeric and the output attribute is binary. As in the protocol by Abspoel et al. [1], it does not reveal any information other than the size of the input and the upper bound h on the tree height.

The computational cost of our protocol is $O(hmn \log n)$, assuming that the comparison and multiplication protocols are unit operations, where m is the number of input attributes in the dataset, and n is the number of samples in the dataset. This is an exponential improvement with respect to h over the computational cost $O(2^h mn \log n)$ of the protocol by Abspoel et al. (Actually, Abspoel et al. [1] claimed only a computational cost $O((2^h + \log n)mn \log n)$, but their protocol can easily be implemented to run in $O(2^h mn \log n)$ by replacing the sorting protocol to efficient one such as [13].) A comparison of the computational costs is shown in Table 1.

Our approach of exponential improvement in computational cost with respect to the tree height is general. For completeness, our protocol is instantiated with all input attributes being numeric, the output attribute being binary, and the evaluation function being the Gini index; however, it is easy to extend. In fact, the main protocol (Algorithm 10), which plays a central role in the exponential improvement of the computational cost, describes a process common to the major decision tree training algorithms CART [6], ID3 [27], and C4.5 [28]. Extensions to other input/other types are discussed in Section 4.8.

Our protocol is built on top of a set of basic protocols, such as multiplication and comparison, provided by many recent MPC frameworks, so it can be used on top of various implementations. More specifically, we build our protocol on top of an MPC model called arithmetic black box (ABB), which consists of a set of basic operations described in Section 2.2.1.

As a byproduct of our decision tree training protocol, we also propose a secure data structure that can compute aggregate values for each group while keeping the grouping information private. This data structure can be used to compute aggregate values such as sums and maximums within each group while keeping the grouping information private, even in cases other than decision tree training.

To see the practicality of our decision tree training protocol, we implemented it on an MPC framework based on a 2-out-of-3 secret sharing scheme. Our protocol trained a decision tree with a height of 4 for 2^{20} inputs of 11 input variables in 404 seconds.

1.2 Overview of our techniques

In general, MPC protocols are incompatible with divide-and-conquer algorithms. In divide-and-conquer algorithms, the problem is divided into smaller subproblems and solved recursively, but MPC protocols need to hide the size of the divided problem as well. A common way to hide the size of the problem is to pad. We hide the actual size of the data by padding (or leaving samples that should be removed) to the split data to make it appear to be the same size as the original. A disadvantage of this method is that it is computationally expensive; since it loses the property that the data size becomes smaller after splitting. For this reason, the previous study [1] required an exponential cost for the height of the tree.

We use the property that the total number of samples is invariant at each height in training decision trees. We keep the data of nodes of the same height together, and train them all at once without padding. This allows our protocol to process only $\Theta(hmn)$ samples in total, while the previous study [1] processes $\Theta(2^h mn)$ samples including dummies.

To implement this idea, we first define a data structure that looks like a private vector of length n, but is internally grouped. Specifically, we place the n grouped elements on a private vector of length n so that elements of the same group appear next to each other, and then create a private vector of length n with a flag corresponding to the first element of each group. This allows us to detect the boundaries of groups by referring to the flags internally, although we cannot distinguish the groupings outwardly.

In decision tree training, each group needs to be split when moving to the next height. We accomplish this within our data structure by stably sorting the elements using the binary branching result, which is computed for each element, as a key. Stability of the sort ensures that elements that are in the same group and have the same branching result will be placed sequentially after the sort. Since this split requires only one-bit-key sorting, it can be very efficient depending on the underlying MPC implementation.

We build the group-wise sum, maximum, and prefix sum computations on our data structure. We then use them to build a decision tree training algorithm similar to [1] on our data structure.

1.3 Related work

There have been many studies on designing efficient algorithms for MPC, and various techniques have been developed. Our research can also be placed in this context.

One technique that has contributed to the design of efficient algorithms is the Oblivious RAM (ORAM) proposed by Goldreich and Ostrovsky [12]. In the context of MPC, ORAM can be used to implement efficient array access. Recently, MPC-friendly ORAMs [31, 35] have also been proposed. These have also been used to construct efficient algorithms for such as stable marriage problem [9] and minimum spanning tree problem [16]. We can construct a decision tree training protocol by running a decision tree training algorithm on a random access machine using ORAM. However, since simple application would result in the number of rounds being equal to the running time of the algorithm in the clear, it is non-trivial to construct a protocol in which both the number of operations and the number of rounds are small.

Another clever technique is the oblivious parallel array access algorithm proposed by Laud [19]. This also aims to access arrays efficiently. It is particularly efficient when reading or writing a large amount of data at once. Efficient algorithms for such as shortest path problems [3] and minimum spanning tree problem [4, 19] have been designed using this algorithm. The data structure proposed in this paper uses the ideas of [19] and [3].

Besides those directly related to our protocol, various tricks for MPC have been studied. The oblivious shuffling algorithm proposed by Laur et al. [22] for reordering array elements into a random order is used as a component of various efficient MPC protocols such as sorting [13] and join [21]. Doerner et al. [9] proposed a secure data structure that can efficiently traverse a linked list when the elements of the list are accessed only once and in order.

2 PRELIMINARIES

In this section, we introduce a typical decision tree training algorithm in the clear and secure multi-party computation.

Before that, we introduce some notation. Throughout this paper, the index of a vector starts at 1. We refer to the i-th element of a vector $\vec{v}$ by $\vec{v}[i]$. That is, if $\vec{v}$ is a vector of length n, then $\vec{v} = (\vec{v}[1], \vec{v}[2], \ldots, \vec{v}[n])$. Let S_n denote a symmetric group on $[1, n]$. That is, S_n is the set of all bijective functions from $[1, n]$ to $[1, n]$. A *permutation* is an element of S_n. *Applying* a permutation $\pi \in S_n$ to a vector $\vec{x}$ of length n is the operation of rearranging $\vec{x}$ into a vector $\vec{z}$ satisfying $\vec{z}[\pi(i)] = \vec{x}[i]$ for $i \in [1, n]$. We denote this operation as $\pi(\vec{x})$. In logical operations, 0 represents false and 1 represents true.

2.1 Decision tree training

Decision tree training is a method in machine learning. The goal is to obtain a model called a *decision tree* that predicts a value of an *output attribute*, given values of *input attributes*. There are several famous algorithms for decision tree training, such as CART [6], ID3 [27], and C4.5 [28]. The general framework of these algorithms is the same, and in fact they are all greedy algorithms based on the divide-and-conquer paradigm. In this section, we present a typical algorithm, for which we plan to construct a secure version, for training a two-class classification binary tree, where all input attributes are numerical.

2.1.1 Typical decision tree training algorithm. Let us start with defining notation. Consider a dataset D with m input attributes $X_1, \ldots, X_m$ and an output attribute Y. Suppose there are *n samples*, each sample being a pair of an *input tuple* x and a *class label* y. Here, x is an m-tuple, and y is a value of the output attribute Y. The j-th element of x represents a value of the input attribute X_j. A decision tree consists of a binary tree and some additional information. Each internal node (non-leaf node) has a condition called a *test* of the form $X_j < t$. It asks if the j-th element in a given input tuple is less than a *threshold* t or not. Each edge is assigned a possible outcome of its source node's test, that is, true or false. An edge whose assigned outcome is true (false) is called a *true edge* (*false edge*, respectively). A child node whose incoming edge is a true edge (false edge) is called a *true-child node* (*false-child node*, respectively). Each leaf node is assigned a class label called *leaf label*. This information is used to predict a class label for a given input tuple as follows. Starting from the root node, we repeat evaluating the test of the internal node we reach and tracing an outgoing edge that is assigned the same value as the test outcome. When we reach a leaf node, we output its leaf label as the predicted class label.

Algorithm 1: A typical decision tree training algorithm in the clear.

Notation: $\mathcal{T} := \mathsf{Train}(\mathcal{D})$
Input: A training dataset $\mathcal{D}$.
Output: A decision tree $\mathcal{T}$.

if *the stopping criterion is met* **then**
  Let r be a leaf node whose leaf label is the most common class label in $\mathcal{D}$. Outputs a tree whose root node is r.
else
  Find the best test $X_j < t$ according to the variable selection measure.
  Recursively computes two subtrees $\mathcal{T}_{X_j<t} := \mathsf{Train}(\mathcal{D}_{X_j<t})$ and $\mathcal{T}_{X_j\geq t} := \mathsf{Train}(\mathcal{D}_{X_j\geq t})$.
  Let v be an internal node v whose test is $X_j < t$. Output a tree such that its root node is v, v's true-child node is $\mathcal{T}_{X_j<t}$'s root node, and v's false-child node is $\mathcal{T}_{X_j\geq t}$'s root node.

A typical decision tree training algorithm is shown in Algorithm 1. It trains a tree recursively from the root node to a leaf node in a top-down fashion. At each node, it checks if the stopping criterion is satisfied using the given training dataset $\mathcal{D}$ to determine the node type. If the stopping criterion is satisfied, the current node is set to be a leaf node. Then, it sets the most frequent class label in the dataset to the leaf label of the current node, and outputs a tree whose root is this node. If the stopping criterion is not satisfied, the current node is set as an internal node. In this case, we select a test of the form $X_j < t$ that gives the best data splitting with respect to a predetermined criterion, and split the training dataset $\mathcal{D}$ into $\mathcal{D}_{X_j<t}$ and $\mathcal{D}_{X_j\geq t}$ according to this test, where $\mathcal{D}_{X_j<t} := \{(x, y) \in \mathcal{D} \mid x(X_j) < t\}$ and $\mathcal{D}_{X_j\geq t} := \{(x, y) \in \mathcal{D} \mid x(X_j) \geq t\}$. Here, we refer to the j-th element of an input tuple x by $x(X_j)$. It then recursively trains decision trees $\mathcal{T}_{X_j<t}$ and $\mathcal{T}_{X_j\geq t}$ with $\mathcal{D}_{X_j<t}$ and $\mathcal{D}_{X_j\geq t}$ as the training data, respectively, and sets the roots of these trees as the child nodes of the current node, and outputs a tree whose root is the current node.

We use the commonly used stopping criterion: (1) the height of the node is h, or (2) the dataset cannot be split further (i.e., (i) all

class labels are the same, or (ii) all input tuples are the same), where h is an upper bound of the tree height, which is typically given as a hyperparameter.

We also assume that the threshold is chosen to be the average of some two different attribute values. This allows the number of candidate threshold values for a particular attribute to be $O(n)$.

2.1.2 Test selection measure. The size and shape of the decision tree depends on which tests are selected at the internal nodes. In general, it is desirable to make the tree as small as possible, but the problem of constructing a decision tree that minimizes the sum of the lengths of the paths from the root to each leaf is known to be NP-hard [15]. Therefore, we usually define a measure for goodness of local splitting and select a test that maximizes this measure.

Commonly used measures for goodness of split include the information gain used in ID3 [27] and the Gini index used in CART [6]. We use the Gini index, which is also used in previous studies such as [1, 8] due to its ease of computation in MPC.

Two types of Gini indices are defined: one for a dataset and one for a dataset and a test. The Gini index for a dataset $\mathcal{D}$, which we denote by $\mathsf{Gini}(\mathcal{D})$, is defined as follows: $\mathsf{Gini}(\mathcal{D}) := 1 - \sum_{c\in\{0,1\}} (|\mathcal{D}_{Y=c}|^2/|\mathcal{D}|^2)$, where $\mathcal{D}_{Y=c} := \{(x, y) \in \mathcal{D} \mid y = c\}$ is a subset of $\mathcal{D}$ whose class label is c. Intuitively, the smaller $\mathsf{Gini}(\mathcal{D})$ is, the purer $\mathcal{D}$ becomes in terms of class labels.

The Gini index for a dataset $\mathcal{D}$ and a test $X_j < t$, which we denote by $\mathrm{G}_{X_j<t}(\mathcal{D})$, is defined using Gini as follows:

$$\mathrm{G}_{X_j<t}(\mathcal{D}) := \frac{|\mathcal{D}_{X_j<t}|}{|\mathcal{D}|}\,\mathsf{Gini}(\mathcal{D}_{X_j<t}) + \frac{|\mathcal{D}_{X_j\geq t}|}{|\mathcal{D}|}\,\mathsf{Gini}(\mathcal{D}_{X_j\geq t}).$$

Intuitively, the smaller $\mathrm{G}_{X_j<t}(\mathcal{D})$ is, the purer each split dataset becomes (and hence the better the test is). Therefore, to find the best test for splitting a dataset $\mathcal{D}$, we compute a test T that minimizes $\mathrm{G}_T(\mathcal{D})$ [14].

Abspoel et al. [1] showed that minimization of $\mathrm{G}_{X_j<t}(\mathcal{D})$ is equivalent to maximization of $\mathrm{G'}_{X_j<t}(\mathcal{D})$ defined as

$$\begin{aligned}\mathrm{G'}_{X_j<t}(\mathcal{D}) := (&|\mathcal{D}_{X_j\geq t}|(|\mathcal{D}_{X_j<t\wedge Y=0}|^2 + |\mathcal{D}_{X_j<t\wedge Y=1}|^2)\\ &+|\mathcal{D}_{X_j<t}|(|\mathcal{D}_{X_j\geq t\wedge Y=0}|^2+|\mathcal{D}_{X_j\geq t\wedge Y=1}|^2))/(|\mathcal{D}_{X_j<t}||\mathcal{D}_{X_j\geq t}|),\end{aligned} \tag{1}$$

where $\mathcal{D}_{X_j<t\wedge Y=c} := \{(x, y) \in \mathcal{D} \mid x(X_j) < t \wedge y = c\}$ and $\mathcal{D}_{X_j\geq t\wedge Y=c} := \{(x, y) \in \mathcal{D} \mid x(X_j) \geq t \wedge y = c\}$. We refer to it as *modified Gini index* and use it as a measure in our protocol.

2.2 Secure multi-party computation

We model secure multi-party computation (MPC) with an ideal functionality called arithmetic black box (ABB). This ideal functionality allows a set of parties $P_1, \ldots, P_C$ to store values, operate on the stored values, and retrieve the stored values. We build our protocol on top of an ABB. This allows our protocol to run on any MPC implementation that realizes ABB, since concrete ABB implementation is separated from their construction.

2.2.1 Arithmetic black box. We assume a simple ABB named $\mathcal{F}_{\mathrm{ABB}}$ over a ring $\mathbb{Z}_M$ for some integer M as shown in Fig. 1. We denote a value referred to by a name x stored in $\mathcal{F}_{\mathrm{ABB}}$ as $[\![x]\!]$. In a typical case, where $\mathcal{F}_{\mathrm{ABB}}$ is realized by a secret sharing based MPC, $[\![x]\!]$ means that x is secret shared. We say a value is *private* if it is stored in $\mathcal{F}_{\mathrm{ABB}}$.

- A command $[\![z]\!] \leftarrow \mathsf{Enc}(x, P_i)$: Receive x from a party P_i and store it as $[\![x]\!]$.
- A command $z \leftarrow \mathsf{Dec}([\![x]\!])$: Send x to every party, who store it in the local variable z.
- A command $[\![z]\!] \leftarrow \mathsf{Add}([\![x]\!], [\![y]\!])$: Compute $z := x+y$ and store it as $[\![z]\!]$.
- A command $[\![z]\!] \leftarrow \mathsf{Mul}([\![x]\!], [\![y]\!])$: Compute $z := xy$ and store it as $[\![z]\!]$.
- A command $[\![z]\!] \leftarrow \mathsf{EQ}([\![x]\!], [\![y]\!])$: If $x = y$ then set $z := 1$, otherwise set $z := 0$. Store it as $[\![z]\!]$.
- A command $[\![z]\!] \leftarrow \mathsf{LT}([\![x]\!], [\![y]\!])$: If $x < y$ then set $z := 1$, otherwise set $z := 0$. Store it as $[\![z]\!]$.
- A command $[\![\pi]\!] \leftarrow \mathsf{SortPerm}([\![\vec{x}_1]\!], [\![\vec{x}_2]\!], \ldots, [\![\vec{x}_k]\!])$: Assume that $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_k$ are vectors of length n. Compute a permutation $\pi \in S_n$, such that it lexicographically and stably sorts $((\vec{x}_1[1], \ldots, \vec{x}_k[1]), \ldots, (\vec{x}_1[n], \ldots, \vec{x}_k[n]))$. Store π as $[\![\pi]\!]$.
- A command $[\![\vec{z}]\!] \leftarrow \mathsf{Apply}([\![\pi]\!], [\![\vec{x}]\!])$: Assume that $\pi \in S_n$ is a permutation and $\vec{x}$ is a vector of length n. Compute $\vec{z} := \pi(\vec{x})$ and store it as $[\![\vec{z}]\!]$.
- A command $[\![\vec{z}]\!] \leftarrow \mathsf{Unapply}([\![\pi]\!], [\![\vec{x}]\!])$: Assume that $\pi \in S_n$ is a permutation and $\vec{x}$ is a vector of length n. Compute $\vec{z} := \pi^{-1}(\vec{x})$ and store it as $[\![\vec{z}]\!]$.

We assume that the commands Add, Mul, EQ, and LT are also defined in the same way when one of the inputs is a public value.

Figure 1: The arithmetic black box functionality $\mathcal{F}_{\mathrm{ABB}}$.

We identify residue classes in $\mathbb{Z}_M$ with their representatives in $[0, M)$. We assume M is sufficiently large such that vector indices can be stored in $\mathcal{F}_{\mathrm{ABB}}$. We also assume that the number of parties C is constant.

For notational simplicity, $[\![z]\!] \leftarrow \mathsf{Add}([\![x]\!], [\![y]\!])$, $[\![z]\!] \leftarrow \mathsf{Mul}([\![x]\!], [\![y]\!])$, $[\![z]\!] \leftarrow \mathsf{EQ}([\![x]\!], [\![y]\!])$, and $[\![z]\!] \leftarrow \mathsf{LT}([\![x]\!], [\![y]\!])$ are also written as $[\![z]\!] \leftarrow [\![x]\!] + [\![y]\!]$, $[\![z]\!] \leftarrow [\![x]\!] \times [\![y]\!]$, $[\![z]\!] \leftarrow ([\![x]\!] \stackrel{?}{=} [\![y]\!])$, and $[\![z]\!] \leftarrow ([\![x]\!] \stackrel{?}{<} [\![y]\!])$, respectively. Furthermore, we denote $[\![x_1]\!] + [\![x_2]\!] + \cdots + [\![x_n]\!]$ by $\sum_{i=1}^{n} [\![x_i]\!]$.

2.2.2 Cost of MPC protocols. We define the cost of an MPC protocol as the number of invocations of relatively basic ABB operations Enc, Dec, Mul, EQ, or LT other than linear combination of private values. That is, we assume that the parties can compute $\mathsf{Add}([\![x]\!], [\![y]\!])$, $\mathsf{Add}(c, [\![y]\!])$, $\mathsf{Add}([\![x]\!], c)$, $\mathsf{Mul}(c, [\![y]\!])$, and $\mathsf{Mul}([\![x]\!], c)$ for free, where $[\![x]\!]$ and $[\![y]\!]$ are private values and c is a public value. This cost models the communication complexity on a typical MPC based on a linear secret sharing scheme, in which the parties can locally compute linear combination of secret shared values. We refer to the above ABB operations, except for linear combinations of private values, as *non-free operations*.

In this paper, we measure the cost of protocols by the number of invocations and depth of invocations of non-free operations. These correspond to the communication cost and communication rounds

(or multiplicative depth) of a typical MPC implementation by a constant factor, respectively.

The relatively complex ABB operations of SortPerm, Apply, and Unapply are typically constructed by combining other basic ABB operations. Therefore, we assume the costs of these operations to be as follows in accordance with typical constructions. We assume that SortPerm costs $O(n \log n)$ non-free operations in $O(\log n)$ rounds. This can be realized by the construction by Laud and Willemson [20]. We also assume that both Apply and Unapply cost $O(n \log n)$ non-free operations in $O(\log n)$ rounds. These can be realized by the construction by Falk and Ostrovsky [10].

2.2.3 Known protocols. We show known protocols that we will use as building blocks for our protocols. The protocols shown here are limited to those that can be built on $\mathcal{F}_{\mathrm{ABB}}$ for completeness. Some MPC implementations may provide the same functionality more efficiently, in which case we can use them instead of the protocols listed here to run our protocol more efficiently.

We start by defining some simple protocols.

- $[\![z]\!] \leftarrow [\![x]\!]$ OR $[\![y]\!]$ computes logical disjunction of bits x and y as $[\![x]\!]+[\![y]\!]-[\![x]\!]\times[\![y]\!]$, using $O(1)$ non-free operations in $O(1)$ rounds.
- $[\![z]\!] \leftarrow \neg[\![x]\!]$ computes negation of a bit x as $1-[\![x]\!]$, using no non-free operations.
- $[\![z]\!] \leftarrow \mathsf{IfElse}([\![c]\!], [\![t]\!], [\![f]\!])$ receives a bit c and two values t and f, and computes t if $c = 1$, f otherwise, as $[\![f]\!] + [\![c]\!] \times ([\![t]\!] - [\![f]\!])$, using $O(1)$ non-free operations in $O(1)$ rounds.
- $[\![z]\!] \leftarrow \mathsf{Max}([\![x]\!], [\![y]\!])$ computes the maximum value of x and y as $\mathsf{IfElse}(([\![x]\!] \overset{?}{<} [\![y]\!]), [\![y]\!], [\![x]\!])$, using $O(1)$ non-free operations in $O(1)$ rounds.

We require an extended Max protocol, which we call VectMax. We let $[\![z]\!] \leftarrow \mathsf{VectMax}([\![\vec{x}]\!], [\![\vec{y}]\!])$ denote the operation that computes a private value $[\![z]\!]$ such that $[\![\vec{x}]\!]$ and $[\![\vec{y}]\!]$ are private vectors of the same length n, $i = \arg\max_{j \in [1,n]} \vec{x}[j]$, and $z = \vec{y}[j]$. We use the construction by Abspoel et al. [1], which uses $O(n)$ non-free operations in $O(\log n)$ rounds.

To simplify the description, we introduce a small subprotocol Sort for sorting private vectors. We let

$$[\![\vec{z}_1]\!], \ldots, [\![\vec{z}_m]\!] \leftarrow \mathsf{Sort}([\![\vec{x}_1]\!], \ldots, [\![\vec{x}_k]\!]; [\![\vec{y}_1]\!], \ldots, [\![\vec{y}_m]\!])$$

denote the following procedure:

(1) $[\![\pi]\!] \leftarrow \mathsf{SortPerm}([\![\vec{x}_1]\!], \ldots, [\![\vec{x}_k]\!])$;
(2) $[\![\vec{z}_j]\!] \leftarrow \mathsf{Apply}([\![\pi]\!], [\![\vec{y}_j]\!])$ for $j \in [1, m]$.

We sometimes use similar notation when the same operation is applied to multiple inputs. For example,

$$[\![z_1]\!], \ldots, [\![z_m]\!] \leftarrow \mathsf{IfElse}([\![c]\!]; [\![t_1]\!], \ldots, [\![t_m]\!]; [\![f_1]\!], \ldots, [\![f_m]\!])$$

means parallel execution of $[\![z_j]\!] \leftarrow \mathsf{IfElse}([\![c]\!], [\![t_j]\!], [\![f_j]\!])$ for $j \in [1, m]$.

If vectors are given for a protocol defined for scalar values, it means that the protocol is applied on an element-by-element basis. That is, $[\![\vec{z}]\!] \leftarrow [\![\vec{x}]\!] \times [\![\vec{y}]\!]$ means parallel execution of $[\![\vec{z}[i]]\!] \leftarrow [\![\vec{x}[i]]\!] \times [\![\vec{y}[i]]\!]$ for $i \in [1, n]$, and $[\![\vec{z}]\!] \leftarrow \mathsf{IfElse}([\![\vec{c}]\!], [\![\vec{t}]\!], [\![\vec{f}]\!])$ means parallel execution of $[\![\vec{z}[i]]\!] \leftarrow \mathsf{IfElse}([\![\vec{c}[i]]\!], [\![\vec{t}[i]]\!], [\![\vec{f}[i]]\!])$ for $i \in [1, n]$.

If some of the inputs are scalar, the same scalar values are used for all executions. For example, $[\![\vec{z}]\!] \leftarrow 2 \times [\![\vec{y}]\!]$ means parallel execution of $[\![\vec{z}[i]]\!] \leftarrow 2 \times [\![\vec{y}[i]]\!]$ for $i \in [1, n]$, and $[\![\vec{z}]\!] \leftarrow \mathsf{IfElse}([\![c]\!], [\![\vec{t}]\!], 1)$ means parallel execution of $[\![\vec{z}[i]]\!] \leftarrow \mathsf{IfElse}([\![c]\!], [\![\vec{t}[i]]\!], 1)$ for $i \in [1, n]$.

2.2.4 Security. Our protocols inherit the security guarantees of an MPC implementing ABB. That is, if the underlying MPC is secure against passive (or active) adversaries our protocols will be too. We will prove the security of our protocols in a hybrid model, where parties run a real protocol and have access to a trusted party that computes a subfunctionality. If the subfunctionality is f, we say that the protocol is executed in the f-hybrid model.

3 OUR SECURE GROUP-WISE AGGREGATION PROTOCOLS

In this section, we propose group-wise aggregation protocols that compute aggregate values (sum, prefix sum, and maximum) for each group without revealing the grouping information of the input grouped values. These are executed on grouped values stored in our data structure. These protocols and the data structure play a central role in the construction of our decision tree training protocol proposed in Section 4.

3.1 Our data structure for privately grouped values

We propose a data structure that stores grouped values without revealing any information about the grouping. We store n values, divided into several groups, in a private vector $[\![\vec{x}]\!]$ of length n, called the *internally grouped vector*. Here, elements in the same group are stored as consecutive elements in the vector. That is, for any i, j, and k such that $1 \le i < j < k \le n$, if $\vec{x}[i]$ and $\vec{x}[k]$ are in the same group, then $\vec{x}[i]$ and $\vec{x}[k]$ are also in the same group. Along with such a vector, we maintain a private bit vector $[\![\vec{g}]\!]$ of length n, called the *group flag vector*, which indicates the boundaries between groups. Namely, we set $\vec{g}[i] = 1$ if the i-th element in $\vec{x}$ is the first element in a group, otherwise $\vec{g}[i] = 0$. By definition, $\vec{g}[1] = 1$ is always true.

We show an example. Suppose that six values are stored in an internally grouped vector $\vec{x}$ as $\vec{x} = (3, 1, 2, 2, 3, 2)$ and the corresponding group flag vector is $\vec{g} = (1, 0, 1, 1, 0, 0)$. Then, this means that the six values are divided into three groups, $(3, 1)$, (2), and $(2, 3, 2)$.

For the sake of simplicity, we introduce some notations. Let $\mathsf{Head}(\vec{g}, i)$ ($\mathsf{Tail}(\vec{g}, i)$) be the index of the first (last, respectively) element of the group that contains the i-th element within the grouping represented by a group flag vector $\vec{g}$. Formally, they are defined as $\mathsf{Head}(\vec{g}, i) := \max\{j \in [1, i] \mid \vec{g}[j] = 1\}$ and $\mathsf{Tail}(\vec{g}, i) := \min\{j \in (i, n] \mid \vec{g}[j] = 1\} \cup \{n + 1\} - 1$, respectively, where n is the length of $\vec{g}$. For example, if a group flag vector is defined as $\vec{g} = (1, 0, 1, 1, 0, 0)$, then $\mathsf{Head}(\vec{g}, 2) = 1$, $\mathsf{Head}(\vec{g}, 4) = 4$, $\mathsf{Tail}(\vec{g}, 4) = 6$, and $\mathsf{Tail}(\vec{g}, 3) = 3$.

Table 2: Example of input/output for our group-wise aggregation protocols, where $\vec{g}$ is a group flag vector and $\vec{x}$ is an internally grouped vector.

Input		Output		
$\vec{g}$	$\vec{x}$	Sum	Prefix sum	Max
1	3	4	3	3
0	1	4	4	3
1	2	2	2	2
1	2	7	2	3
0	3	7	5	3
0	2	7	7	3

Table 3: Truth table for extended binary operation.

f_a	f_b	f_c	d	f_d	e	f_e
0	0	0	$(a \oplus b) \oplus c$	0	$a \oplus (b \oplus c)$	0
0	0	1	c	1	c	1
0	1	0	$b \oplus c$	1	$b \oplus c$	1
0	1	1	c	1	c	1
1	0	0	$(a \oplus b) \oplus c$	0	$a \oplus (b \oplus c)$	0
1	0	1	c	1	c	1
1	1	0	$b \oplus c$	1	$b \oplus c$	1
1	1	1	c	1	c	1

3.2 Group-wise prefix-"operation" protocols

We construct protocols to compute group-wise prefix-sums over associative binary operators. For a binary operator $\oplus$, the *group-wise prefix-$\oplus$* protocol privately computes prefix sums according to $\oplus$ of each group in our data structure. It receives a private group flag vector $[\![\vec{g}]\!]$ of length n and a private internally grouped vector $[\![\vec{x}]\!]$ of values to be summed, and outputs a private vector $[\![\vec{y}]\!]$ of prefix sums for each group such that $\vec{y}[i] = \vec{x}[\mathsf{Head}(\vec{g}, i)] \oplus \vec{x}[\mathsf{Head}(\vec{g}, i) + 1] \oplus \cdots \oplus \vec{x}[i]$. An example of input/output when the binary operator is "add" is shown in Table 2. Columns 1 and 2 are the inputs, and column 4 is the output.

Anagreh et al. [3] proposed protocols for efficiently computing prefix min of values with the same key, using a generic algorithm for efficiently computing prefix sum. In this section, we first propose a generic group-wise prefix-$\oplus$ protocol for an arbitrary associative binary operator $\oplus$ using a similar approach to Anagreh et al. We then instantiate it with respect to addition and present a group-wise prefix-sum protocol.

3.2.1 Binary operation to compute group-wise prefix sum. For any given associative binary operation $\oplus$, we propose a method to create another associative binary operation $\oplus'$ such that the prefix sum over $\oplus'$ computes a group-wise prefix sum over $\oplus$. We will use $\oplus'$ and an efficient prefix sum algorithm A to efficiently compute group-wise prefix sum for $\oplus$.

A binary operation $\oplus$ is said to be *associative* if it satisfies $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ for any a, b, and c. For example, addition, multiplication, and max of real numbers are associative, but subtraction and division are not.

Our method of creating a new binary operation is as follows.

Definition 3.1. Let $\oplus$ be a binary operation on a set S. The binary operation $\oplus'$ on $S \times \{0, 1\}$ defined by

$$(x, f_x) \oplus' (y, f_y) = (\text{if } f_y \text{ then } y \text{ else } x \oplus y, f_x \vee f_y)$$

is called the *extended binary operation* of $\oplus$. We refer to x and f_x by *body* and *bit* of (x, f_x).

Intuition. Intuitively, the bits in the extended value can be interpreted as flags to reset. When the sum is computed from the left by an expanded binary operation $\oplus'$, the body of the result is equal to the sum by the original binary operation $\oplus$. However, once a value x with bit 1 is added from the right, the body of the result is reset to the body of x. Therefore, by setting the group flag to the bit and computing prefix sum, the prefix sum for each group is computed on the body.

Correctness. We show that the group-wise prefix sum of the original binary operation can be computed by the prefix sum of the extended binary operation. First, from Definition 3.1, the following propositions are immediate.

Proposition 3.2. *If $\oplus'$ is an extended binary operation of $\oplus$, then $(x, f_x) \oplus' (y, 1) = (y, 1)$.*

Proposition 3.3. *If $\oplus'$ is an extended binary operation of $\oplus$, then $(x, f_x) \oplus' (y, 0) = (x \oplus y, f_x)$.*

These are used to show that the extended operation is associative.

Theorem 3.4. *If a binary operation $\oplus$ is associative, then the extended binary operation $\oplus'$ of $\oplus$ is also associative.*

Proof. Let $S' := S \times \{0, 1\}$, $(a, f_a), (b, f_b), (c, f_c) \in S'$, $(d, f_d) := ((a, f_a) \oplus' (b, f_b)) \oplus' (c, f_c)$, and $(e, f_e) := (a, f_a) \oplus' ((b, f_b) \oplus' (c, f_c))$. We will show that $(d, f_d) = (e, f_e)$ always holds.

Using Propositions 3.2 and 3.3, we can list all cases as in Table 3. In all cases $(d, f_d) = (e, f_e)$, so $\oplus'$ is associative. □

Next, we show that the prefix sum of $\oplus'$ computes the group-wise prefix sum of $\oplus$.

Theorem 3.5. *Let $\oplus'$ be an extended binomial operation of an associative binary operation $\oplus$, i be an integer greater than or equal to 1, $(X_i, G_i) = (x_1, g_1) \oplus' (x_2, g_2) \oplus' \cdots \oplus' (x_i, g_i)$, and $g_1 = 1$. If $j \in [1, i]$ is an integer such that $g_j = 1$ and $g_k = 0$ for all $k \in [j + 1, i]$, then $X_i = x_j \oplus x_{j+1} \oplus \cdots \oplus x_i$.*

Proof. Using $g_j = 1$, associativity of $\oplus'$, and Proposition 3.2 in order, we can see that

$$\begin{aligned}
&(x_1, g_1) \oplus' (x_2, g_2) \oplus' \cdots \oplus' (x_j, g_j) \\
&= (x_1, g_1) \oplus' (x_2, g_2) \oplus' \cdots \oplus' (x_j, 1) \\
&= ((x_1, g_1) \oplus' (x_2, g_2) \oplus' \cdots \oplus' (x_{j-1}, g_{j-1})) \oplus' (x_j, 1) \\
&= (x_j, 1).
\end{aligned}$$

Then, using associativity of $\oplus'$, the assumption, and Proposition 3.3, we have that

$$\begin{aligned}(X_i, G_i) &= ((x_1, g_1) \oplus' (x_2, g_2) \oplus' \cdots \oplus' (x_j, g_j)) \\ &\quad \oplus' (x_{j+1}, g_{j+1}) \oplus' \cdots \oplus' (x_i, g_i) \\ &= (x_j, 1) \oplus' (x_{j+1}, 0) \oplus' \cdots \oplus' (x_i, 0) \\ &= (x_j \oplus x_{j+1}, 1) \oplus' (x_{j+2}, 0) \oplus' \cdots \oplus' (x_i, 0) \\ &= \cdots \\ &= (x_j \oplus x_{j+1} \oplus \cdots \oplus x_i, 1).\end{aligned}$$

□

Algorithm 2: Generic group-wise prefix-$\oplus$ protocol for an associative binary operator $\oplus$.

Input: A group flag vector $[\![\vec{g}]\!]$ of length n and a internally grouped vector $[\![\vec{x}]\!]$ of length n to be aggregated.
Output: An aggregation result vector $[\![\vec{y}]\!]$ of length n.

1 Define a binary operator $\oplus'$ by $([\![a]\!], [\![f_a]\!]) \oplus' ([\![b]\!], [\![f_b]\!]) :=$ $(\mathsf{IfElse}([\![f_b]\!], [\![b]\!], [\![a]\!] \oplus [\![b]\!]), [\![f_a]\!] \mathsf{OR} [\![f_b]\!])$.
2 Let $[\![\vec{v}]\!] := (([\![\vec{x}[1]]\!], [\![\vec{g}[1]]\!]), \ldots, ([\![\vec{x}[n]]\!], [\![\vec{g}[n]]\!]))$.
3 Apply the Ladner-Fisher prefix sum algorithm [18] to $[\![\vec{v}]\!]$ on $\oplus'$ operator and obtain a vector $(([\![\vec{y}[1]]\!], [\![\vec{g}'[1]]\!]), \ldots, ([\![\vec{y}[n]]\!], [\![\vec{g}'[n]]\!]))$.

3.2.2 Generic protocol. In this section, we construct a generic group-wise prefix-$\oplus$ protocol for an associative binary operator $\oplus$ using the method proposed in Section 3.2.1.

Intuition. First, we extend the binary operation $\oplus$ to construct the extended binary operation $\oplus'$ that computes group-wise prefix sum over $\oplus$. Then, the group-wise prefix sum over $\oplus'$ is computed using a generic prefix sum algorithm by Ladner and Fischer [18], which calls $O(n)$ binary operations in $O(\log n)$ rounds.

Protocol and its correctness. The protocol is shown in Algorithm 2. The binary operation $\oplus'$ defined in Line 1 is an extended binary operation defined in Definition 3.1. Therefore, by Theorem 3.5, we obtain the group-wise prefix sum over $\oplus$ as $\vec{y}$ in Line 3.

3.2.3 Instantiation. Let GroupPrefixSum be the protocol that runs Algorithm 2 when the binary operator is Add. It uses $O(n)$ non-free operations in $O(\log n)$ rounds since each invocation of extended binary operation uses $O(1)$ non-free operations in $O(1)$ rounds.

3.3 Group-wise "operation" protocols

We construct protocols to compute group-wise sums over associative binary operators using the generic group-wise prefix-"operation" protocols proposed in Section 3.2. For a binary operator $\oplus$, the *group-wise* $\oplus$ protocol privately computes sums according to $\oplus$ of each group in our data structure. It receives a private group flag vector $[\![\vec{g}]\!]$ of length n and a private internally grouped vector $[\![\vec{x}]\!]$ of length n, and then outputs a private vector $[\![\vec{y}]\!]$ of sums for each group such that $y[i] = x[\mathsf{Head}(\vec{g}, i)] \oplus x[\mathsf{Head}(\vec{g}, i) + 1] \oplus \cdots \oplus x[\mathsf{Tail}(\vec{g}, i)]$. An example for the case when the binary operator is "add" ("max", respectively) is shown in Table 2, where columns 1 and 2 are the inputs, and column 3 (column 5, respectively) is the output.

Algorithm 3: Generic group-wise $\oplus$ protocol for an associative binary operator $\oplus$.

Input: A group flag vector $[\![\vec{g}]\!]$ of length n and a internally grouped vector $[\![\vec{x}]\!]$ of length n to be aggregated.
Output: An aggregation result vector $[\![\vec{z}]\!]$ of length n.

1 Let $[\![\vec{y}']\!]$ be the aggregation result vector obtained by applying Algorithm 2 to $[\![\vec{g}]\!]$ and $[\![\vec{x}]\!]$ with respect to $\oplus$.
2 $[\![\vec{t}[i]]\!] \leftarrow [\![\vec{g}[i+1]]\!]$ for $i \in [1, n)$ and $[\![\vec{t}[n]]\!] \leftarrow 1$.
3 $[\![\vec{z}]\!] \leftarrow [\![\vec{y}']\!] \times [\![\vec{t}]\!]$.
4 $[\![\vec{y}]\!] \leftarrow$ $\mathsf{Reverse}(\mathsf{GroupPrefixSum}(\mathsf{Reverse}([\![\vec{t}]\!]), \mathsf{Reverse}([\![\vec{z}]\!])))$.

3.3.1 Generic protocol. We construct a generic group-wise $\oplus$ protocol for an associative binary operator $\oplus$ using the generic group-wise prefix-$\oplus$ protocol proposed in Section 3.2.

Intuition. We first compute group-wise prefix-sums over $\oplus$ by applying Algorithm 3. Then, it is easy to convert them to the group-wise sums over $\oplus$. Since the last element for each group is equal to the sum over $\oplus$ in the group, we can compute group-wise sum by clearing all but the last elements and copying the last elements by applying GroupPrefixSum in the reversed order.

Protocol and its correctness. The protocol is shown in Algorithm 3. In Line 1, the prefix-sums $\vec{y}'$ for each group are computed by applying Algorithm 3. Since the successor of the last element in a group is the first element in the succeeding group, $\vec{t}$ is a flag vector indicating whether it is the last element in a group (Line 2). In Line 3, all but the last elements for each group are cleared to 0 by $\vec{y}' \times \vec{t}$. Since the last elements are equal to the sums in each group, application of GroupPrefixSum in the reversed order copies these values and the group-wise sums are computed (Line 4).

3.3.2 Instantiation. Let GroupSum and GroupMax be the protocols that run Algorithm 2 when the binary operators are Add and Max, respectively. Both protocols use $O(n)$ non-free operations in $O(\log n)$ rounds since each invocation of extended binary operations uses $O(1)$ non-free operations in $O(1)$ rounds.

4 OUR EFFICIENT DECISION TREE TRAINING PROTOCOL

In this section, we present our decision tree training protocol. Given a private training data set, it outputs the trained decision tree in a private form. Our decision tree training protocol is efficient. While existing studies required exponential non-free operations on h to train a decision tree, our protocol requires only linear number of operations in h.

We first explain our intuition in Section 4.1. Then we introduce our tree standardization in order to simplify the protocol in Section 4.2. In Section 4.3, we explain how to efficiently store data. Then, we describe the proposed protocol, starting from its components.

For the reader's convenience, tables summarizing the variables and protocols are presented in Tables 10 and 11, respectively.

4.1 Intuition

Our training follows the same flow as the typical training algorithm seen in Section 2.1.1, where the node-by-node partitioning is done recursively from root to leaf. The recursive computation is performed on the partitioned data, but the size of the partitioned data must be hidden. Therefore, existing studies needed to make it look as if n samples were processed after the partitioning by padding, which required exponential non-free operations on h.

Our main idea is to store all data in a data structure of size n, focusing on the fact that the total number of samples is invariant when the data is split. Then, by placing elements of the same group consecutively and keeping the locations of their boundaries in memory, we internally partition the data and perform training at each node in a batch for each layer. This eliminates the need for padding, and the entire learning process can be completed by training each layer h times.

The main problem in implementing our idea is how to create the $k+1$-layer data from the k-layer data, split by the test results. We solve this using a stable sorting. After computing the test result for each sample in the k-layer data, we perform a stable sort using the test result as a key. The stability of the sort guarantees that elements that were originally in the same group and had the same test result will remain in the same order after sorting, and we can create grouped data in the $(k+1)$st layer.

Another difficulty is how to compute the best per-node test on our data structure. To address this, we take advantage of the fact that the efficient test selection algorithm by Abspoel et al.[1] can be written as an element-by-element computation and an aggregate computation (sum, max, prefix sum, and sort) of the entire data. The element-by-element computation can be done for each element in our data structure. The overall aggregate computation, which in our case must be done for each group, can be computed using the operations described in Section 3.

4.2 Decision tree normalization for efficiency

Our training protocol outputs an equivalent normalized decision tree instead of the one that should have been obtained when training in the clear. Equivalent in this case means that the output for any given input is the same. Roughly speaking, our normalization aligns the heights of all leaf nodes to the upper bound on the tree height by inserting internal nodes that forward any sample to the false child node. Although this increases the number of nodes in the tree, in MPC, it reduces the data size and computational cost (though by just a constant factor), and simplifies the protocol.

Before describing the details of our normalization, let us recall the decision tree we originally wanted to compute, which is the output of Algorithm 1. It is a binary tree with height less than or equal to h, and all tests for its internal nodes are of the form $X_j < t$, where j is an attribute number and t is a threshold.

Our normalization consists of two modifications. The first modification is to change each test $X_j < t$ to an equivalent test $2X_j < t'$ such that $t' = 2t$. In Algorithm 1, we compute $t = (\vec{x}_j[i] + \vec{x}_j[i+1])/2$ when computing a threshold, but this involves division, which is a costly operation in MPC. We avoid division by computing $t' = 2t = \vec{x}_j[i] + \vec{x}_j[i+1]$ instead of t.

Our second modification is to align the height of all leaf nodes without changing the tree's output. The modification is simple. We insert an internal node u, which does not actually split, into the position of a leaf node v with height less than h. The test for u is $2X_1 < \mathsf{MIN\_VALUE}$, which always returns false, and u has a false branch to v, where MIN_VALUE is a sufficiently small public value. Any input tuple that reaches u passes through the false branch to reach v, so the predicted label does not change. In the normalized tree, all nodes with height less than h are internal nodes, and all nodes with height h are leaf nodes. This makes our protocol simple and efficient.

4.3 Encoding of inputs and outputs

Appropriate data storage methods are essential to the design of efficient algorithms. Our decision tree training protocol receives a private training dataset and a public upper bound h on the height of the tree, and outputs a private decision tree of height h. This section explains how we store these values.

First, we introduce the representation of the input training data. The training dataset consists of n samples, each of which consists of input tuple and a binary value called a class label. Each input tuple consists of m numerical input attribute values. Our protocol receives it as m private vectors $[\![\vec{x}_j]\!]$ ($j \in [1, m]$) of length n and a private vector $[\![\vec{y}]\!]$ of length n. That is, the i-th input tuple of the training dataset and its associated class label correspond to $(\vec{x}_1[i], \vec{x}_2[i], \ldots, \vec{x}_m[i])$ and $\vec{y}[i]$, respectively.

Next, we describe how to represent the output decision tree. The output tree is a normalized binary tree as described in Section 4.2. It is stored in $3h+2$ private vectors. Since all nodes of height $k \in [0, h)$ are internal nodes, the node number, attribute number, and threshold of each node are stored in three vectors $\mathsf{NID}^{(k)}$, $\mathsf{AID}^{(k)}$, and $\mathsf{Threshold}^{(k)}$, respectively. For nodes of height h, since all nodes are leaf nodes, the node number and leaf label of each node are stored in two vectors $\mathsf{NID}^{(k)}$ and $\mathsf{Label}^{(k)}$, respectively. The length of each vector of height $k \in [0, h]$ is $\min\{n, 2^k\}$. If the actual number of nodes is smaller than the length of the vector, it is filled with a dummy value NULL. The vectors of each layer are collectively denoted as $[\![\mathsf{Layer}^{(k)}]\!] := ([\![\mathsf{NID}^{(k)}]\!], [\![\mathsf{AID}^{(k)}]\!], [\![\mathsf{Threshold}^{(k)}]\!])$ ($k \in [0, h)$) and $[\![\mathsf{Layer}^{(h)}]\!] := ([\![\mathsf{NID}^{(h)}]\!], [\![\mathsf{Label}^{(h)}]\!])$, which we call *layer information*.

Edge information is also needed to represent a tree. However, we avoid having it explicitly by using a commonly used technique to devise a numbering of the nodes. In order to represent the edges between nodes, each node is assigned an integer node number. The only node with height 0 is the root, and its node number is 1. For each child node (of height $k+1$) of a node of height k with node number d, assign node number d to the false child (if any) and node number $d + 2^k$ to the true child (if any). With this numbering scheme, in the k-th layer, all node numbers are assigned different values from $[1, 2^k]$.

4.4 Our batch test selection protocol

In this section, we show how to compute the best tests for each group from the current entire data at a given layer.

The main difficulty is how to construct the test selection algorithm by [1] on top of our data structure. Our idea is to rewrite the algorithm by [1] to element-by-element computation and aggregation operations over the entire data (max, sum, prefix sum, etc.). Then, the former can be computed as is and the latter can be realized in group-wise manner using the group-wise operations.

To simplify the explanation, we divide the computation into three protocols (modified Gini index, attribute-wise test selection, and global test selection), moving from computations in smaller ranges to computations in larger ranges, in order.

Each protocol is used as a component of a protocol with larger ranges.

Algorithm 4: Modified Gini index.

Notation: $[\![\vec{\mathsf{S}}]\!] \leftarrow \mathsf{ModifiedGini}([\![\vec{g}]\!], [\![\vec{y}]\!])$.
Input: A group flag vector $[\![\vec{g}]\!]$ of length n and an output attribute vector $[\![\vec{y}]\!]$ of length n.
Output: A score vector $[\![\vec{\mathsf{S}}]\!]$ of length n such that $\vec{\mathsf{S}}[i]$ is the modified Gini index when the dataset is split between the i-th and $(i+1)$st elements.
Cost: $O(n \log n)$ non-free operations in $O(\log n)$ rounds.

1 $[\![\vec{y}_0]\!] \leftarrow \neg[\![\vec{y}]\!]$ and $[\![\vec{y}_1]\!] \leftarrow [\![\vec{y}]\!]$.
2 $[\![\vec{u}_b]\!] \leftarrow \mathsf{GroupPrefixSum}([\![\vec{g}]\!], [\![\vec{y}_b]\!])$ for $b \in \{0,1\}$.
3 $[\![\vec{v}_b]\!] \leftarrow \mathsf{GroupSum}([\![\vec{g}]\!], [\![\vec{y}_b]\!])$ for $b \in \{0,1\}$.
4 $[\![\vec{w}_b]\!] \leftarrow [\![\vec{v}_b]\!] - [\![\vec{u}_b]\!]$ for $b \in \{0,1\}$.
5 $[\![\vec{u}]\!] \leftarrow [\![\vec{u}_0]\!] + [\![\vec{u}_1]\!]$ and $[\![\vec{w}]\!] \leftarrow [\![\vec{w}_0]\!] + [\![\vec{w}_1]\!]$.
6 $[\![\vec{p}]\!] \leftarrow [\![\vec{w}]\!] \times ([\![\vec{u}_0]\!]^2 + [\![\vec{u}_1]\!]^2) + [\![\vec{u}]\!] \times ([\![\vec{w}_0]\!]^2 + [\![\vec{w}_1]\!]^2)$.
7 $[\![\vec{q}]\!] \leftarrow [\![\vec{u}]\!] \times [\![\vec{w}]\!]$.
8 $[\![\vec{\mathsf{S}}]\!] \leftarrow [\![\vec{p}]\!]/[\![\vec{q}]\!]$. Here, for simplicity, we set $\vec{\mathsf{S}}$ to be the result of element-wise division of $\vec{p}$ and $\vec{q}$; however, in practice, we set $\vec{\mathsf{S}}$ to be the pair of $\vec{p}$ and $\vec{q}$, and a comparison of the elements in $\vec{\mathsf{S}}$ is replaced by the division-free comparison as [1].

4.4.1 Modified Gini index. We present a protocol to compute all modified Gini indices for privately grouped dataset when splitting is performed between adjacent elements. Thanks to the group-wise operations proposed in Section 3, the formula by Abspoel et al. [1] described in Equation (1) can be used almost directly.

The protocol is shown in Algorithm 4. The input is a group flag vector $[\![\vec{g}]\!]$ and an output attribute vector $[\![\vec{y}]\!]$ which is assumed to be sorted by an input attribute in each group. It uses the group-wise operations to compute the values required by Equation (1) in a privately grouped manner (Lines 2 to 5). Equation (1) is then evaluated for each element to obtain the output score vector $[\![\vec{\mathsf{S}}]\!]$ in Lines 6 to 8, where each $\vec{\mathsf{S}}[i]$ represents the modified Gini index for a split between the i-th and $(i+1)$st elements. A small example appears in Fig. 5.

Correctness. Since each $\vec{y}_b$ computed in Line 1 is a bit vector with $\vec{y}_b[i] = 1$ iff $\vec{y}[i] = b$, let the i-th element be e, then $\vec{u}_b[i]$ represents the number of b's up to e in the group, and $\vec{w}_b[i]$ represents the number of b's after e in the group. That is, each value corresponds to an element in the following 2×2 contingency table.

	# elements up to i-th	# elements after i-th	Total
# 0's	$\vec{u}_0[i]$	$\vec{w}_0[i]$	$\vec{v}_0[i]$
# 1's	$\vec{u}_1[i]$	$\vec{w}_1[i]$	$\vec{v}_1[i]$
Total	$\vec{u}[i]$	$\vec{w}[i]$	

From Equation (1), $p[i]/q[i]$ is the modified Gini index for splitting between the i-th and $(i+1)$st elements.

Complexity. The protocol uses $O(n \log n)$ non-free operations in $O(\log n)$ rounds.

Algorithm 5: Batch attribute-wise test selection.

Notation: $[\![\vec{\mathsf{T}}]\!], [\![\vec{\mathsf{S}}]\!] \leftarrow \mathsf{AttributewiseTestSelection}([\![\vec{g}]\!], [\![\vec{x}]\!], [\![\vec{y}]\!])$
Input: A group flag vector $[\![\vec{g}]\!]$ of length n, an input attribute vector $[\![\vec{x}]\!]$ of length n, and an output attribute vector $[\![\vec{y}]\!]$ of length n.
Output: a threshold vector $[\![\vec{\mathsf{T}}]\!]$ of length n and a score vector $[\![\vec{\mathsf{S}}]\!]$ of length n.
Cost: $O(n \log n)$ non-free operations in $O(\log n)$ rounds.

1 $[\![\vec{\mathsf{S}}]\!] \leftarrow \mathsf{ModifiedGini}([\![\vec{g}]\!], [\![\vec{y}]\!])$.
2 $[\![\vec{\mathsf{T}}[i]]\!] \leftarrow [\![\vec{x}[i]]\!] + [\![\vec{x}[i+1]]\!]$ for all $i \in [1,n)$ and $[\![\vec{\mathsf{T}}[n]]\!] \leftarrow \mathsf{MIN\_VALUE}$.
3 $[\![\vec{p}[i]]\!] \leftarrow [\![\vec{g}[i+1]]\!]$ OR $([\![\vec{x}[i]]\!] \stackrel{?}{=} [\![\vec{x}[i+1]]\!])$ for $i \in [1,n)$ and $[\![\vec{p}[n]]\!] \leftarrow 1$.
4 $[\![\vec{\mathsf{S}}]\!],[\![\vec{\mathsf{T}}]\!] \leftarrow \mathsf{IfElse}([\![\vec{p}]\!]; \mathsf{MIN\_VALUE}, \mathsf{MIN\_VALUE}; [\![\vec{\mathsf{S}}]\!], [\![\vec{\mathsf{T}}]\!])$.
5 $(([\![\vec{\mathsf{S}}[1]]\!], [\![\vec{\mathsf{T}}[1]]\!]), \ldots, ([\![\vec{\mathsf{S}}[n]]\!], [\![\vec{\mathsf{T}}[n]]\!])) \leftarrow \mathsf{GroupMax}^*([\![\vec{g}]\!], (([\![\vec{\mathsf{S}}[1]]\!],[\![\vec{\mathsf{T}}[1]]\!]), \ldots, ([\![\vec{\mathsf{S}}[n]]\!],[\![\vec{\mathsf{T}}[n]]\!])))$.

4.4.2 Attribute-wise test selection. Using ModifiedGini as a building block, we construct a protocol for computing the best tests in each group for a given numerical input attribute. The protocol implements Abspoel et al.'s technique [1] of reducing candidate split points from between all pairs ($\Theta(n^2)$ pairs) to between adjacent elements ($n-1$ pairs) by sorting input data by the input attribute, on our data structure using the group-wise operations proposed in Section 3.

The protocol is shown in Algorithm 5. It receives a group flag vector $[\![\vec{g}]\!]$, an input attribute vector $[\![\vec{x}]\!]$, and an output attribute vector $[\![\vec{y}]\!]$ such that $\vec{x}$ and $\vec{y}$ are privately grouped by $\vec{g}$ and sorted by $\vec{x}$ within each group. First, ModifiedGini is applied to compute the score for each element when the data is split immediately after the element. Then, the scores of points that cannot be split are overwritten with $-\infty$, and finally, GroupMax*, which is an extension of GroupMax, is used to select the test that has the maximum score within each group. GroupMax* is almost identical to GroupMax, except that the second input and output are not vectors of values, but vectors of *pairs* of values. Comparison of the pairs is assumed to be done by comparing the first element of each. Thus, the maximum of two pairs is computed as follows: $\mathsf{Max}((s_1,t_1),(s_2,t_2)) := (\text{if } s_1 \geq s_2 \text{ then } (s_1,t_1) \text{ else } (s_2,t_2))$. That is, the maximum value of the pair vector $((s_1,t_1),\ldots,(s_n,t_n))$ is (s_i,t_i) for $i = \arg\max_{j \in [1,n]} s_j$. A small example appears in Fig. 6.

Correctness. In Lines 1 and 2, the threshold $\vec{\mathsf{T}}[i]$ and the score $\vec{\mathsf{S}}[i]$ for split between the i-th and $(i+1)$st elements are computed. If the i-th element is the last element in a group (i.e., $i = n$ or $\vec{g}[i+1] = 1$) or if it has the same attribute value as the next element (i.e., $\vec{x}[i] = \vec{x}[i+1]$), we cannot split between the i-th and $(i+1)$st elements. In this case, we set $\vec{\mathsf{T}}[i] :=$ MIN_VALUE and $\vec{\mathsf{S}}[i] :=$ MIN_VALUE (Lines 3 and 4). In Line 5, the score and threshold of an element whose score is the maximum in a group are copied to other elements in the group.

Complexity. The protocol uses $O(n \log n)$ non-free operations in $O(\log n)$ rounds.

Algorithm 6: Batch global test selection.

Notation: $[\![\vec{\mathsf{A}}]\!], [\![\vec{\mathsf{T}}]\!] \leftarrow$ GlobalTestSelection$((([\![\vec{x}_j]\!])_{j\in[1,m]}, [\![\vec{y}]\!], [\![\vec{g}]\!])$

Input: m input attribute vectors $([\![\vec{x}_j]\!])_{j\in[1,m]}$ of length n, an output attribute vector $[\![\vec{y}]\!]$ of length n, and a group flag vector $[\![\vec{g}]\!]$ of length n.

Output: An attribute number vector $[\![\vec{\mathsf{A}}]\!]$ of length n and a threshold vector $[\![\vec{\mathsf{T}}]\!]$ of length n.

Cost: $O(mn \log n)$ non-free operations in $O(\log n + \log m)$ rounds.

1 **for each** $j \in [1, m]$ **do in parallel**
2 $\quad [\![\vec{u}_j]\!], [\![\vec{v}_j]\!] \leftarrow$ Sort(PrefixSum($[\![\vec{g}]\!]$), $[\![\vec{x}_j]\!]$; $[\![\vec{x}_j]\!]$, $[\![\vec{y}]\!]$).
3 $\quad [\![\vec{\mathsf{T}}_j]\!], [\![\vec{\mathsf{S}}_j]\!] \leftarrow$ AttributewiseTestSelection($[\![\vec{g}]\!]$, $[\![\vec{u}_j]\!]$, $[\![\vec{v}_j]\!]$).
4 **for each** $i \in [1, n]$ **do in parallel**
5 $\quad [\![\vec{\mathsf{A}}[i]]\!] \leftarrow$ VectMax($([\![\vec{\mathsf{S}}_1[i]]\!], \ldots, [\![\vec{\mathsf{S}}_m[i]]\!])$, $(1, \ldots, m)$).
6 $\quad [\![\vec{\mathsf{T}}[i]]\!] \leftarrow$ VectMax($([\![\vec{\mathsf{S}}_1[i]]\!], \ldots, [\![\vec{\mathsf{S}}_m[i]]\!])$, $([\![\vec{\mathsf{T}}_1[i]]\!], \ldots, [\![\vec{\mathsf{T}}_m[i]]\!])$).

4.4.3 Global test selection. Using AttributewiseTestSelection as a building block, we construct a protocol for computing the best test through all attributes for each node in a batch. The algorithm is straightforward: it calls the attribute-wise test selection protocol to compute the best test for each attribute and then selects the best test among them.

The protocol is shown in Algorithm 6. It receives the training data (input tuples and class labels) privately grouped by nodes, and outputs the information (attribute number and threshold) of the best test for each group. For each input attribute, it sorts the input attribute values and class labels within the group and selects the best test for each group when splitting on that attribute (Lines 1 to 3). Then we select the best test among all attributes in Lines 5 and 6. A small example appears in Fig. 7.

Correctness. The protocol is almost identical to the algorithm in the clear and the protocol in [1]. The difference is that we need to sort within each group in Line 2. Recalling that $\vec{g}$ is a bit vector where only the first element of each group is 1, we can see that PrefixSum($\vec{g}$) computes different and ascending values for each group. Thus, we can sort within each group by using PrefixSum($\vec{g}$) and $\vec{x}_j$ as keys in lexicographic order as in Line 2.

Complexity. The protocol uses $O(mn \log n)$ non-free operations in $O(\log m + \log n)$ rounds.

4.5 Batch training for internal nodes

Algorithm 7: Summarizing a privately grouped vector.

Notation: $[\![\vec{z}]\!] \leftarrow$ FormatLayer($[\![\vec{g}]\!]$, $[\![\vec{u}]\!]$, w).

Input: A group flag vector $[\![\vec{g}]\!]$ of length n, a vector $[\![\vec{u}]\!]$ to be summarized, and an integer w.

Output: A summarized vector $[\![\vec{z}]\!]$ of length w.

Cost: $O(cn \log n)$ non-free operations in $O(\log n)$ rounds.

1 $[\![\vec{r}]\!] \leftarrow$ IfElse($[\![\vec{g}]\!]$, $[\![\vec{u}]\!]$, NULL).
2 $[\![\vec{v}]\!] \leftarrow$ Sort($\neg[\![\vec{g}]\!]$; $[\![\vec{r}]\!]$).
3 Let $[\![\vec{z}]\!]$ be the first w elements of $[\![\vec{v}]\!]$.

Algorithm 8: Training internal nodes.

Notation: $[\![\mathsf{Layer}^{(k)}]\!], [\![\vec{b}]\!] \leftarrow$ TrainInternalNodes$(k, ([\![\vec{x}_j]\!])_{j\in[1,m]}, [\![\vec{y}]\!], [\![\vec{g}]\!], [\![\vec{\mathsf{N}}]\!])$.

Input: A layer height k, m input attribute vectors $([\![\vec{x}_j]\!])_{j\in[1,m]}$ of length n, an output attribute vector $[\![\vec{y}]\!]$ of length n, a group flag vector $[\![\vec{g}]\!]$ of length n, and a node number vector $[\![\vec{\mathsf{N}}]\!]$ of length n.

Output: A layer information $[\![\mathsf{Layer}^{(k)}]\!] = ([\![\mathsf{NID}^{(k)}]\!], [\![\mathsf{AID}^{(k)}]\!], [\![\mathsf{Threshold}^{(k)}]\!])$ and a test result vector $[\![\vec{b}]\!]$ of length n.

Cost: $O(mn \log n)$ non-free operations in $O(\log n + \log m)$ rounds.

1 $[\![\vec{\mathsf{A}}]\!], [\![\vec{\mathsf{T}}]\!] \leftarrow$ GlobalTestSelection$(([\![\vec{x}_j]\!])_j, [\![\vec{y}]\!], [\![\vec{g}]\!])$.
2 $[\![\vec{c}]\!] \leftarrow$ GroupSum($[\![\vec{g}]\!]$, $\vec{1}$), where $\vec{1}$ is a vector $(1, \ldots, 1)$ of length n.
3 $[\![\vec{c}_0]\!] \leftarrow$ GroupSum($[\![\vec{g}]\!]$, $\neg[\![\vec{y}]\!]$) and $[\![\vec{c}_1]\!] \leftarrow$ GroupSum($[\![\vec{g}]\!]$, $[\![\vec{y}]\!]$).
4 $[\![\vec{f}]\!] \leftarrow ([\![\vec{c}]\!] \stackrel{?}{=} [\![\vec{c}_0]\!])$ OR $([\![\vec{c}]\!] \stackrel{?}{=} [\![\vec{c}_1]\!])$.
5 $[\![\vec{\mathsf{A}}]\!], [\![\vec{\mathsf{T}}]\!] \leftarrow$ IfElse($[\![\vec{f}]\!]$; 1, MIN_VALUE; $[\![\vec{\mathsf{A}}]\!]$, $[\![\vec{\mathsf{T}}]\!]$).
6 Compute $[\![\mathsf{NID}^{(k)}]\!]$, $[\![\mathsf{AID}^{(k)}]\!]$, and $[\![\mathsf{Threshold}^{(k)}]\!]$ by applying FormatLayer($[\![\vec{g}]\!]$, $\cdot$, $\min\{2^k, n\}$) to $[\![\vec{\mathsf{N}}]\!]$, $[\![\vec{\mathsf{A}}]\!]$, and $[\![\vec{\mathsf{T}}]\!]$, respectively.
7 $[\![\vec{e}_j]\!] \leftarrow ([\![\vec{\mathsf{A}}]\!] \stackrel{?}{=} j)$ for $j \in [1, m]$.
8 $[\![\vec{x}_*[i]]\!] \leftarrow \sum_{j\in[1,m]} [\![\vec{x}_j[i]]\!] \times [\![\vec{e}_j[i]]\!]$ for $i \in [1, n]$.
9 $[\![\vec{b}]\!] \leftarrow (2 \times [\![\vec{x}_*]\!] \stackrel{?}{<} [\![\vec{\mathsf{T}}]\!])$.

In this section, we introduce a batch internal node training protocol that will be used to train k-th layer nodes ($k \in [0, h)$). It receives privately grouped dataset of k-th layer, computes the best tests in a batch considering the stopping criteria (2)(i) and (2)(ii), and converts the tests to the form of the output layer information. It also computes the test results.

Before presenting the internal node training protocol, we introduce a subprotocol of *summarizing* that removes redundancy from an internally grouped nodal information kept in sample-by-sample

form by a group flag vector. The protocol is shown in Algorithm 7. Assuming that a group flag vector $\vec{g}$ divides the samples into at most w groups, it converts an internally grouped vector $\vec{v}$ into a vector $\vec{z}$ of length w with no redundancy. The idea is to leave only the top element of each group and fill the rest with dummies. Since $\vec{g}[i] = 1$ if and only if it is the first element of a group, redundancy can be eliminated by overwriting the rest with NULL as in Line 1, sorting in descending order by $\vec{g}$ as in Line 2, and finally outputting the first w elements as in Line 3. A small example appears in Fig. 8.

Now we describe the internal node training protocol. The protocol is shown in Algorithm 8. In computing the best tests, we apply the GlobalTestSelection protocol to obtain tests that consider only the stopping criterion (2)(i) in Line 1 and correct it for (2)(ii) in Lines 2 to 5. The resulting test $[\![\vec{\mathsf{N}}]\!]$, $[\![\vec{\mathsf{A}}]\!]$, and $[\![\vec{\mathsf{T}}]\!]$ are redundant because they are stored element-by-element, so the FormatLayer protocol is used to convert them to the node-by-node stored format $[\![\mathsf{NID}^{(k)}]\!]$, $[\![\mathsf{AID}^{(k)}]\!]$, and $[\![\mathsf{Threshold}^{(k)}]\!]$ in Line 6. Finally, the test results $\vec{b}$ are computed efficiently on an element-by-element basis using redundant but element-by-element form of tests $[\![\vec{\mathsf{A}}]\!]$ and $[\![\vec{\mathsf{T}}]\!]$ (Lines 7 to 9). A small example is shown in Fig. 9.

Correctness. In Lines 2 to 4, $\vec{c}$, $\vec{c}_0$, and $\vec{c}_1$ represent the number of elements, the number of 0's, and the number of 1's in the group, respectively. Thus $\vec{f}[i] = 1$ if and only if all the elements of $\vec{y}$ are the same in the group containing the i-th element. In Line 5, for each i such that $\vec{f}[i] = 1$, $[\![\vec{\mathsf{A}}[i]]\!]$ and $[\![\vec{\mathsf{T}}[i]]\!]$ are overwritten with 1 and MIN_VALUE, respectively. This guarantees the stopping criterion (2)(ii). At the k-th layer, the number of nodes is at most 2^k, so FormatLayer correctly summarizes $[\![\vec{\mathsf{N}}]\!]$, $[\![\vec{\mathsf{A}}]\!]$, and $[\![\vec{\mathsf{T}}]\!]$ in Line 6.

In Lines 7 to 9, we compute the results $[\![\vec{b}]\!]$ of applying the tests denoted by $\vec{\mathsf{A}}$ and $\vec{\mathsf{T}}$ to the input tuples $(\vec{x}_j)_{j\in[1,m]}$. For each $i \in [1, n]$, it computes a flag $[\![\vec{e}_j[i]]\!]$ indicating whether $\vec{\mathsf{A}}[i] = j$ by an equality test (Line 7), and uses it to select the $\vec{\mathsf{A}}[i]$-th attribute value $\vec{x}_{\vec{\mathsf{A}}[i]}[i]$ (Line 8). Then, it computes the test result of each element in Line 9.

Complexity. The TrainInternalNodes protocol uses $O(mn \log n)$ non-free operations in $O(\log n + \log m)$ rounds.

4.6 Batch training for leaf nodes

In this section, we introduce a batch leaf node training protocol that will be used to train h-th layer nodes. Thanks to our group-wise operations, it straightforwardly computes the most frequent values of the output attribute for each group and outputs them as leaf labels. The protocol is shown in Algorithm 9. A small example appears in Fig. 10.

Correctness. In Lines 1 and 2, we compute the most frequent value of output attribute by comparing the number of 0's $\vec{c}_0$ and the number of 1's $\vec{c}_1$. The number of nodes in the input dataset is bounded by $\min\{2^h, n\}$ since the height of the layer is h. Thus, in Line 3, the FormatLayer protocol correctly formats $[\![\vec{\mathsf{N}}]\!]$ and $[\![\vec{\mathsf{L}}]\!]$.

Complexity. The protocol uses $O(n \log n)$ non-free operations in $O(\log n)$ rounds.

Algorithm 9: Training leaf nodes.

Notation: $[\![\mathsf{Layer}^{(h)}]\!] \leftarrow \mathsf{TrainLeafNodes}(h, [\![\vec{g}]\!], [\![\vec{y}]\!], [\![\vec{\mathsf{N}}]\!])$.
Input: A tree height h, a group flag vector $[\![\vec{g}]\!]$ of length n, an output attribute vector $[\![\vec{y}]\!]$ of length n, and a node number vector $[\![\vec{\mathsf{N}}]\!]$ of length n.
Output: $[\![\mathsf{Layer}^{(h)}]\!] = ([\![\mathsf{NID}^{(h)}]\!], [\![\mathsf{Label}^{(h)}]\!])$, where $[\![\mathsf{NID}^{(h)}]\!]$ and $[\![\mathsf{Label}^{(h)}]\!]$ are vectors of length $\min\{n, 2^h\}$.
Cost: $O(n \log n)$ non-free operations in $O(\log n)$ rounds.

1 $[\![\vec{c}_0]\!] \leftarrow \mathsf{GroupSum}([\![\vec{g}]\!], \neg[\![\vec{y}]\!])$ and $[\![\vec{c}_1]\!] \leftarrow \mathsf{GroupSum}([\![\vec{g}]\!], [\![\vec{y}]\!])$.
2 $[\![\vec{\mathsf{L}}]\!] \leftarrow ([\![\vec{c}_0]\!] \overset{?}{<} [\![\vec{c}_1]\!])$.
3 Compute $[\![\mathsf{NID}^{(h)}]\!]$ and $[\![\mathsf{Label}^{(h)}]\!]$ by applying $\mathsf{FormatLayer}([\![\vec{g}]\!], \cdot, \min\{2^h, n\})$ to $[\![\vec{\mathsf{N}}]\!]$ and $[\![\vec{\mathsf{L}}]\!]$, respectively.

Algorithm 10: Decision tree training.

Notation: $([\![\mathsf{Layer}^{(k)}]\!])_{k\in[0,h]} \leftarrow \mathsf{DecisionTreeTraining}((([\![\vec{x}_j^{(0)}]\!])_{j\in[1,m]}, [\![\vec{y}^{(0)}]\!], h)$
Input: m input attribute vectors $([\![\vec{x}_j^{(0)}]\!])_{j\in[1,m]}$ of length n, output attribute vector $[\![\vec{y}^{(0)}]\!]$ of length n, and maximum tree height h.
Output: A decision tree $([\![\mathsf{Layer}^{(k)}]\!])_{k\in[0,h]}$ of height h.
Cost: $O(hmn \log n)$ non-free operations in $O(h(\log n + \log m))$ rounds.

1 $[\![\vec{g}^{(0)}[1]]\!] \leftarrow 1$ and $[\![\vec{g}^{(0)}[i]]\!] \leftarrow 0$ for $i \in [2, n]$.
2 $[\![\vec{\mathsf{N}}^{(0)}[i]]\!] \leftarrow 1$ for $i \in [1, n]$.
3 **for** $k := 0$ **to** $h - 1$ **do**
4 $\quad [\![\mathsf{Layer}^{(k)}]\!], [\![\vec{b}^{(k)}]\!] \leftarrow \mathsf{TrainInternalNodes}(k, ([\![\vec{x}_j^{(k)}]\!])_j, [\![\vec{y}^{(k)}]\!], [\![\vec{g}^{(k)}]\!], [\![\vec{\mathsf{N}}^{(k)}]\!])$.
5 $\quad [\![\vec{\mathsf{N}}_+]\!] \leftarrow 2^k \times [\![\vec{b}^{(k)}]\!] + [\![\vec{\mathsf{N}}^{(k)}]\!]$.
6 $\quad [\![\vec{g}_+]\!] \leftarrow \mathsf{GroupFirstOne}([\![\vec{g}^{(k)}]\!], \neg[\![\vec{b}^{(k)}]\!]) +$ $\mathsf{GroupFirstOne}([\![\vec{g}^{(k)}]\!], [\![\vec{b}^{(k)}]\!])$, where $\mathsf{GroupFirstOne}([\![\vec{g}]\!], [\![\vec{b}]\!])$ is defined by $\mathsf{GroupPrefixSum}([\![\vec{g}]\!], [\![\vec{b}]\!]) \times [\![\vec{b}]\!] \overset{?}{=} 1$.
7 $\quad [\![\vec{x}_1^{(k+1)}]\!], \ldots, [\![\vec{x}_m^{(k+1)}]\!], [\![\vec{y}^{(k+1)}]\!], [\![\vec{g}^{(k+1)}]\!], [\![\vec{\mathsf{N}}^{(k+1)}]\!] \leftarrow \mathsf{Sort}([\![\vec{b}^{(k)}]\!]; [\![\vec{x}_1^{(k)}]\!], \ldots, [\![\vec{x}_m^{(k)}]\!], [\![\vec{y}^{(k)}]\!], [\![\vec{g}_+]\!], [\![\vec{\mathsf{N}}_+]\!])$.
8 $[\![\mathsf{Layer}^{(h)}]\!] \leftarrow \mathsf{TrainLeafNodes}(h, [\![g^{(h)}]\!], [\![\vec{y}^{(h)}]\!], [\![\vec{\mathsf{N}}^{(h)}]\!])$.

4.7 Our layer-by-layer training protocol

Finally, we describe our decision tree training algorithm, using the protocols presented so far as building blocks. We construct a decision tree by training nodes of the same height in a batch, layer by layer, while keeping the input and output secret. Training samples assigned to different nodes in the same layer are processed as internally separate groups using the protocols proposed in Section 3. This improves the 2^h factor of the communication complexity in [1] to h.

The main protocol of our decision tree training is shown in Algorithm 10. Samples and associated values are stored in our

Table 4: Running time for training a decision tree.

			Time [s]		
n	m	h	T1	T2	T3
2^{13}	11	4	10.333	4.798	541.054
2^{18}	11	4	349.515	82.399	969.883
2^{19}	11	4	701.331	178.831	1366.467
2^{20}	11	4	1466.253	403.804	2117.820
2^{13}	20	4	17.380	6.425	614.989
2^{13}	50	4	41.565	11.981	823.540
2^{13}	100	4	81.582	21.976	1166.079
2^{13}	11	10	23.981	11.130	1254.291
2^{13}	11	20	46.760	22.432	2443.332
2^{13}	11	50	116.641	55.234	6011.040

private grouping data structure as internally grouped vectors. At the loop in Line 3, it trains the decision tree layer by layer in order from the 0-th layer (the root node) to the h-th layer (the leaf nodes).

In each iteration, the k-th layer is trained with TrainInternalNodes in Line 4 and obtain the test results. Then the current data is stably sorted according to the test results.. Thanks to the stability of the sort, elements that were in the same group and had the same test result before sorting are guaranteed to appear consecutively after sorting, i.e., they are in the same group within our data structure. A small example is shown in Fig. 11.

Correctness. Throughout the training, a group flag vector $\vec{g}^{(k)}$ represents the grouping to each node of the k-th layer. $\mathsf{N}^{(k)}$,$\vec{x}_j^{(k)}$, and $\vec{y}^{(k)}$ are internally grouped vectors grouped by $\vec{g}^{(k)}$ that store the node numbers, input attribute values, and output attribute values, respectively. In the 0-th layer, all samples are initialized to be assigned to the root node whose node number is 1 (Lines 1 and 2). Then, each layer is trained iteratively (Line 3).

At each iteration, we first train the nodes at the k-th layer and compute test result $\vec{b}^{(k)}$ for each sample (Line 4). This is executed in the TrainInternalNodes protocol. Each $\vec{b}^{(k)}[i]$ represents the test result of the i-th sample. 0 and 1 denote false and true, respectively.

The node numbers $\vec{\mathsf{N}}_+$ and group flags $\vec{g}_+$ at the next layer are computed in Lines 5 and 6, respectively. Then, $\vec{g}^{(k+1)}$, $\vec{\mathsf{N}}^{(k+1)}$, $\vec{x}_j^{(k+1)}$, and $\vec{y}^{(k+1)}$ of the $(k+1)$-th layer are computed by stably sorting $\vec{g}_+$, $\vec{\mathsf{N}}_+$, $\vec{x}_j^{(k)}$, and $\vec{y}^{(k)}$ by $\vec{b}^{(k)}$ (Line 7). Thanks to the stability of sorting, both the correspondence between the values of each sample and the contiguity of elements in the same group are maintained.

In Line 8, we train the leaf nodes in a batch by invoking the TrainLeafNodes protocol and obtain the output vectors $\mathsf{Layer}^{(h)}$ for height h.

Let us verify the correctness of node numbers $\vec{\mathsf{N}}$ and group flags $\vec{g}$ for the next layer. Let V be a node at the k-th layer with node number d. The node number of a child node of V is d for a false child and $d + 2^k$ for a true child. Thus, $[\![\vec{\mathsf{N}}_+]\!]$, which is computed in Line 5, is the node number in the next layer of each sample. As for the group flags, since the splitting of the groups is stable, the first 0 and 1 in each group are the first elements of the group after the split. Since $\neg[\![\vec{b}^{(k-1)}]\!]$ and $[\![\vec{b}^{(k-1)}]\!]$ indicate the positions of 0's and 1's, respectively, the first elements of the new groups can be detected using GroupFirstOne, which detects first 1's for all groups.

Complexity. The decision tree training protocol uses $O(hmn \log n)$ non-free operations in $O(h(\log m + \log n))$ rounds.

Security. Algorithm 10 and its subprotocols consist of local computations except for $\mathcal{F}_{\mathrm{ABB}}$. Furthermore, during the execution of Algorithm 10, no value ever leaves the ABB: the Dec command (Fig. 1) is never invoked by any of the subprotocols. Therefore, the following theorem is immediate.

Theorem 4.1. *Algorithm 10 is secure in the $\mathcal{F}_{\mathrm{ABB}}$-hybrid model.*

4.8 Extension to other input/output types

Our decision tree training protocol is written only for the case where the input attributes are all numeric and the output attribute is binary. We discuss here how to extend it to categorical input attributes and output attribute attributes beyond binary.

To extend our protocol to the case where the output attribute is a c-valued categorical value, it is enough to extend Algorithm 4 so that $[\![\vec{y}_0]\!], [\![\vec{y}_1]\!], \ldots, [\![\vec{y}_{c-1}]\!]$ are used instead of $[\![\vec{y}_0]\!]$ and $[\![\vec{y}_1]\!]$.

To extend our protocol to the case where the output attribute is numeric, it is sufficient to replace Algorithm 4 with a protocol that computes another measure and modify Algorithm 9 to compute desired labels. For example, mean squared error, which is a often used measure in regression trees, can be constructed in the almost same way as Algorithm 4, since it can be written by group-wise prefix sum and group-wise sum in addition to element-by-element operations.

If the input attributes are changed to binary attributes, the only split for an input attribute is consistent with the numerical case, so it works just by giving the input as is. However, there is room for improving the efficiency of Algorithms 4 to 6, as they are designed to process sorted values.

If the input attributes are to be extended to c-valued, almost all protocols must be reworked. However, we can keep our idea of Algorithm 10 by setting the test result to c-valued, changing the 2 in Line 5 to c, and executing GroupFirstOne c times instead of 2 times.

5 DEMONSTRATION OF PRACTICALITY

In order to show the practicality of our decision tree training protocol, we implemented it and measured the running time.

5.1 Implementation methods

We implemented our protocols on a Shamir's secret-sharing [30] based three-party computation over a field $\mathbb{Z}_p$ by Chida et al. [7], where $p = 2^{61} - 1$. This 3PC scheme is secure against passive adversaries with a single static corruption. For the ABB implementation, we used the comparison protocols for EQ and LT by Kikuchi et al. [17] and protocols for other commands Enc, Dec, Add, Mul, SortPerm, Apply, and Unapply in $\mathcal{F}_{\mathrm{ABB}}$ by Chida et al. [7]. Since [17] requires a Mersenne prime for the field size, we chose $p = 2^{61}-1$ similar to 64-bit integer space.

In order to obtain the best performance, our implementation includes several optimizations. Instead of the original protocols

Table 5: Specifications of the machine used for our benchmark. T3 is the same as T1 except that a communication delay of 50 milliseconds per one-way trip is added by software.

	Testbed T1 and T3	Testbed T2
OS	CentOS Linux release 7.1.1503	CentOS Linux release 7.3.1611
CPU	Intel Core i7-6700 (3.40GHz 4 cores/16 threads)	Intel Xeon Gold 6144k (3.50GHz 8 cores/16 threads)×2
RAM / LAN / RTT	32 GB / 1 Gbps / 0.362 ms	768 GB / 10 Gbps / 0.136 ms

Table 6: Comparison of recall and precision with an implementation in the clear

trial	#1		#2		#3		#4		#5	
method	python	ours	python	ours	python	ours	python	ours	python	ours
Recall	0.911	0.933	0.860	0.883	0.913	0.913	0.784	0.784	0.840	0.840
Precision	1.000	1.000	0.880	0.883	0.857	0.913	0.975	0.975	0.954	0.976

GroupPrefixSum, GroupSum, and GroupMax, we implemented protocols GroupPrefixSum2, GroupSum2, and GroupMax2, respectively, which are alternative protocols optimized to be efficient in our ABB implementation with the same inputs and outputs. These alternative protocols are shown in appendix A.

The protocols were implemented with the C++ language. We measured on three testbeds named T1, T2, and T3. T1 is for comparison with existing studies, and T2 is to see the current best performance of training with MPC. T3 is the same as T1 except that a communication delay of 50 milliseconds, which is slightly larger than the delay between the eastern and western United States [24], per one-way trip is added by software. This is intended to simulate measurements in a WAN environment. Each testbed consists of three servers with the same configuration shown in Table 5.

5.2 Comparison with training in the clear

To see the accuracy of our protocol, we compared the performance of models trained with our protocol and an implementation in the clear. The comparison was performed using DecisionTreeClassifier in python version 3.6.9 and scikit learn version 0.23.1, with the options `criterion='gini'` and `max_depth = 4`. We call it python. We also used the same tree height of $h = 4$.

We used the Breast Cancer Wisconsin (Diagnostic) Data Set [33]. It consists of 30 numerical input attributes and a binary output attribute. The number of samples is 569. We randomly split the data and used 80% Our protocol and python were trained and validated with the same split data each time. Experiments were conducted five times using different random seeds for data split.

The results are shown in Table 6. It can be seen that all five trials yielded close predictions. This result supports that our protocol accurately trains decision trees. Possible reasons for the results not being exactly the same include errors in handling real values and differences in tie-breaking policies when the scores are the same.

5.3 Benchmarking results

To show the scalability of our protocol, we measured the running time for different parameters of n, m, and h. Based on the case where $n = 2^{13} = 8192$, $m = 11$, and $h = 4$, we measured the execution time of training by varying each of n, m, and h. Part of results are shown in Table 4. Each runtime is the average value of three measurements. The results show that the running time is approximately linear with respect to each of n, m, and h. More results are shown in Tables 7 to 9.

We give a brief comparison with the result by Abspoel et al. [1]. They estimated the running time of decision tree training to be 496.6 seconds for $n = 2^{13}$, $m = 11$, and $h = 4$ with passive security. In the same condition, our results in T1 and T2 were 10.333 seconds and 4.798 seconds, respectively. Even the running time at T1, which is almost equal to or less than the measurement environment in [1], was 48 times faster than that of Abspoel et al. [1].

Comparing the results on the two testbeds T1 and T2, for sufficiently large parameters, T2 was about 2 to 4 times faster. As expected from the round complexity of the protocol, the results in T1 and T3 show that the protocol running time increases in environments with high latency. In T3, as far as we have measured, all of the trainings have been completed within two hours. Thus, under the parameters we used in our benchmarks, our protocol is practical even for WAN.

6 CONCLUSION

In this paper, we proposed a decision tree training protocol for MPC. Thanks to our new data structure that enables computing group-wise aggregate values while hiding grouping information, our protocol requires only linear number of non-free operation calls in the tree height. In order to show the practicality of our protocol, we implemented it and measured the running time. Our protocol trained a decision tree with a height of 4 in 404 seconds for a dataset of 2^{20} rows and 11 attributes.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for many helpful comments, which greatly improved the article. This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

REFERENCES

[1] Mark Abspoel, Daniel Escudero, and Nikolaj Volgushev. 2021. Secure training of decision trees with continuous attributes. *Proc. Priv. Enhancing Technol.* 2021, 1 (2021), 167–187. https://doi.org/10.2478/popets-2021-0010

[2] Samuel Adams, Chaitali Choudhary, Martine De Cock, Rafael Dowsley, David Melanson, Anderson C. A. Nascimento, Davis Railsback, and Jianwei Shen. 2022. Privacy-preserving training of tree ensembles over continuous data. *Proc. Priv.*

Enhancing Technol. 2022, 2 (2022), 205–226. https://doi.org/10.2478/popets-2022-0042
[3] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. 2021. Parallel Privacy-Preserving Shortest Path Algorithms. *Cryptogr.* 5, 4 (2021), 27. https://doi.org/10.3390/cryptography5040027
[4] Mohammad Anagreh, Eero Vainikko, and Peeter Laud. 2021. Parallel Privacy-preserving Computation of Minimum Spanning Trees. In *Proceedings of the 7th International Conference on Information Systems Security and Privacy, ICISSP 2021, Online Streaming, February 11-13, 2021*, Paolo Mori, Gabriele Lenzini, and Steven Furnell (Eds.). SCITEPRESS, 181–190. https://doi.org/10.5220/0010255701810190
[5] Leo Breiman. 2001. Random Forests. *Mach. Learn.* 45, 1 (2001), 5–32. https://doi.org/10.1023/A:1010933404324
[6] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*. Wadsworth.
[7] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. 2019. An Efficient Secure Three-Party Sorting Protocol with an Honest Majority. *IACR Cryptol. ePrint Arch.* (2019), 695. https://eprint.iacr.org/2019/695
[8] Sebastiaan de Hoogh, Berry Schoenmakers, Ping Chen, and Harm op den Akker. 2014. Practical Secure Decision Tree Learning in a Teletreatment Application. In *FC 2014, March 3-7, 2014, Christ Church, Barbados (Lecture Notes in Computer Science, Vol. 8437)*, Nicolas Christin and Reihaneh Safavi-Naini (Eds.). Springer, 179–194. https://doi.org/10.1007/978-3-662-45472-5_12
[9] Jack Doerner, David Evans, and Abhi Shelat. 2016. Secure Stable Matching at Scale. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 1602–1613. https://doi.org/10.1145/2976749.2978373
[10] Brett Hemenway Falk and Rafail Ostrovsky. 2021. Secure Merge with O(n log log n) Secure Operations. In *ITC 2021, July 23-26, 2021, Virtual Conference (LIPIcs, Vol. 199)*, Stefano Tessaro (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:29. https://doi.org/10.4230/LIPIcs.ITC.2021.7
[11] Jerome H Friedman. 2001. Greedy function approximation: a gradient boosting machine. *Annals of statistics* (2001), 1189–1232.
[12] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473. https://doi.org/10.1145/233551.233553
[13] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2012. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms. In *ICISC (LNCS, Vol. 7839)*, Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon (Eds.). Springer, 202–216.
[14] Jiawei Han, Micheline Kamber, and Jian Pei. 2011. *Data Mining: Concepts and Techniques, 3rd edition*. Morgan Kaufmann. http://hanj.cs.illinois.edu/bk3/
[15] Laurent Hyafil and Ronald L. Rivest. 1976. Constructing Optimal Binary Decision Trees is NP-Complete. *Inf. Process. Lett.* 5, 1 (1976), 15–17. https://doi.org/10.1016/0020-0190(76)90095-8
[16] Marcel Keller and Peter Scholl. 2014. Efficient, Oblivious Data Structures for MPC. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 8874)*, Palash Sarkar and Tetsu Iwata (Eds.). Springer, 506–525. https://doi.org/10.1007/978-3-662-45608-8_27
[17] Ryo Kikuchi, Dai Ikarashi, Takahiro Matsuda, Koki Hamada, and Koji Chida. 2018. Efficient Bit-Decomposition and Modulus-Conversion Protocols with an Honest Majority. In *ACISP 2018, July 11-13, 2018, Wollongong, NSW, Australia (Lecture Notes in Computer Science, Vol. 10946)*, Willy Susilo and Guomin Yang (Eds.). Springer, 64–82. https://doi.org/10.1007/978-3-319-93638-3_5
[18] Richard E. Ladner and Michael J. Fischer. 1980. Parallel Prefix Computation. *J. ACM* 27, 4 (1980), 831–838. https://doi.org/10.1145/322217.322232
[19] Peeter Laud. 2015. Parallel Oblivious Array Access for Secure Multiparty Computation and Privacy-Preserving Minimum Spanning Trees. *Proc. Priv. Enhancing Technol.* 2015, 2 (2015), 188–205. https://doi.org/10.1515/popets-2015-0011
[20] Peeter Laud and Jan Willemson. 2014. Composable Oblivious Extended Permutations. In *FPS 2014, November 3-5, 2014, Montreal, QC, Canada (Lecture Notes in Computer Science, Vol. 8930)*, Frédéric Cuppens, Joaquín García-Alfaro, A. Nur Zincir-Heywood, and Philip W. L. Fong (Eds.). Springer, 294–310. https://doi.org/10.1007/978-3-319-17040-4_19
[21] Sven Laur, Riivo Talviste, and Jan Willemson. 2013. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7954)*, Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini (Eds.). Springer, 84–101. https://doi.org/10.1007/978-3-642-38980-1_6
[22] Sven Laur, Jan Willemson, and Bingsheng Zhang. 2011. Round-Efficient Oblivious Database Manipulation. In *Information Security, 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 7001)*, Xuejia Lai, Jianying Zhou, and Hui Li (Eds.). Springer, 262–277. https://doi.org/10.1007/978-3-642-24861-0_18
[23] Yehuda Lindell and Benny Pinkas. 2000. Privacy Preserving Data Mining. In *CRYPTO (LNCS, Vol. 1880)*, Mihir Bellare (Ed.). Springer, 36–54.
[24] Microsoft. 2022. Azure network round-trip latency statistics. Retrieved Aug 25, 2022 from https://docs.microsoft.com/en-us/azure/networking/azure-network-latency
[25] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *CCS 2018, October 15-19, 2018, Toronto, ON, Canada*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM, 35–52. https://doi.org/10.1145/3243734.3243760
[26] Philipp Probst, Anne-Laure Boulesteix, and Bernd Bischl. 2019. Tunability: Importance of Hyperparameters of Machine Learning Algorithms. *J. Mach. Learn. Res.* 20 (2019), 53:1–53:32. http://jmlr.org/papers/v20/18-444.html
[27] J. Ross Quinlan. 1986. Induction of Decision Trees. *Mach. Learn.* 1, 1 (1986), 81–106. https://doi.org/10.1023/A:1022643204877
[28] J. Ross Quinlan. 2014. *C4.5: programs for machine learning*. Elsevier.
[29] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *AsiaCCS 2018, June 04-08, 2018, Incheon, Republic of Korea*, Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim (Eds.). ACM, 707–721. https://doi.org/10.1145/3196494.3196522
[30] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (1979), 612–613. https://doi.org/10.1145/359168.359176
[31] Emil Stefanov, Marten van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *J. ACM* 65, 4 (2018), 18:1–18:26. https://doi.org/10.1145/3177872
[32] Sameer Wagh, Divya Gupta, and Nishanth Chandran. 2019. SecureNN: 3-Party Secure Computation for Neural Network Training. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 26–49. https://doi.org/10.2478/popets-2019-0035
[33] William H Wolberg, W Nick Street, and Olvi L Mangasarian. 1992. Breast cancer Wisconsin (diagnostic) data set. *UCI Machine Learning Repository [http://archive.ics.uci.edu/ml/]* (1992).
[34] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets (Extended Abstract). In *27th Annual Symposium on Foundations of Computer Science, 27–29 October 1986, Toronto, Canada*. IEEE Computer Society, 162–167. https://doi.org/10.1109/SFCS.1986.25
[35] Samee Zahur, Xiao Wang, Mariana Raykova, Adrià Gascón, Jack Doerner, David Evans, and Jonathan Katz. 2016. Revisiting Square-Root ORAM: Efficient Random Access in Multi-party Computation. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 218–234. https://doi.org/10.1109/SP.2016.21

A ALTERNATIVE GROUP-WISE AGGREGATION PROTOCOLS

We show alternative group-wise protocols that are used in our implementation in Section 5. These are asymptotically inefficient compared to the protocols proposed in Section 3 in the cost model of Section 2.2.2, but can be more efficient than the protocols of Section 3 in practical use when Apply, Unapply, and SortPerm with a single key bit are efficient, as in [7].

A.1 Alternative protocol for group-wise sum

Necessary operations. Before presenting our protocol, let us define some operations related to the computation of prefix sum. Given a vector $\vec{x}$ of length n, $\vec{z} \leftarrow \mathsf{PrefixSum}(\vec{x})$ computes a vector $\vec{z}$ of length n such that $\vec{z}[i] = \sum_{j=1}^{i} \vec{x}[j]$ for $i \in [1, n]$. We also define an inverse operation $\mathsf{PrefixSum}^{-1}$. Let $\vec{z} \leftarrow \mathsf{PrefixSum}^{-1}(\vec{x})$ denote an operation that computes $\vec{z}$ such that $\vec{x} = \mathsf{PrefixSum}(\vec{z})$. This can be easily computed as $\vec{z}[1] := \vec{x}[1]$ and $\vec{z}[i] := \vec{x}[i] - \vec{x}[i-1]$ for all $i \in [2, n]$. We further define reverse-ordered versions of these operations. Given a vector $\vec{x}$ of length n, $\vec{z} \leftarrow \mathsf{PrefixSumR}(\vec{x})$ computes a vector $\vec{z}$ of length n such that $\vec{z}[i] = \sum_{j=i}^{n} \vec{x}[j]$ for $i \in [1, n]$. Given a vector $\vec{x}$ of length n, $\vec{z} \leftarrow \mathsf{PrefixSumR}^{-1}(\vec{x})$ computes a vector $\vec{z}$ of length n such that $\vec{x} = \mathsf{PrefixSumR}(\vec{z})$. This is computed as $\vec{z}[n] := \vec{x}[n]$ and $\vec{z}[i] := \vec{x}[i] - \vec{x}[i+1]$ for all $i \in [1, n)$.

Algorithm 11: Alternative group-wise sum.

Notation: $[\![\vec{y}]\!] \leftarrow \mathsf{GroupSum2}([\![\vec{g}]\!], [\![\vec{x}]\!])$.
Input: A group flag vector $[\![\vec{g}]\!]$ of length n and a internally grouped vector $[\![\vec{x}]\!]$ of length n to be aggregated.
Output: An aggregation result vector $[\![\vec{y}]\!]$ of length n.
Cost: $O(n \log n)$ non-free operations in $O(\log n)$ rounds.

1 $[\![\vec{p}]\!] \leftarrow \mathsf{PrefixSumR}([\![\vec{x}]\!]) \times [\![\vec{g}]\!]$.
2 $[\![\pi]\!] \leftarrow \mathsf{SortPerm}(\neg[\![\vec{g}]\!])$.
3 $[\![\vec{p}_1]\!] \leftarrow \mathsf{Apply}([\![\pi]\!], [\![\vec{p}]\!])$.
4 $[\![\vec{s}_1]\!] \leftarrow \mathsf{PrefixSumR}^{-1}([\![\vec{p}_1]\!])$.
5 $[\![\vec{d}_1]\!] \leftarrow \mathsf{PrefixSum}^{-1}([\![\vec{s}_1]\!])$.
6 $[\![\vec{d}]\!] \leftarrow \mathsf{Unapply}([\![\pi]\!], [\![\vec{d}_1]\!]) \times [\![\vec{g}]\!]$.
7 $[\![\vec{y}]\!] \leftarrow \mathsf{PrefixSum}([\![\vec{d}]\!])$.

Intuition. First, let us illustrate our intuition using the case when $\vec{g} := (1, 0, 1, 1, 0, 0)$ and $\vec{x} := (3, 1, 2, 2, 3, 2)$. The vector we wish to compute is $\vec{y} := (4, 4, 2, 7, 7, 7)$. The protocol consists of two parts: (1) An intermediate vector $\vec{q} := (4, 0, 2, 7, 0, 0)$ such that the first element of each group is the sum of the elements of $\vec{x}$ in that group and all other elements are 0 is computed from $\vec{x}$. (2) $\vec{y}$ is computed by copying the value of the first element in each group of $\vec{q}$ to the other elements.

In (1), applying PrefixSumR to $\vec{x}$ followed by element-by-element multiplication with $\vec{g}$ seems to give a vector close to $\vec{q}$. In fact, it gives $\vec{p} := \mathsf{PrefixSumR}(\vec{x}) \times \vec{g} = (13, 0, 9, 7, 0, 0)$. Although this is not equal to $\vec{q}$, we can see that if we subtract the first element of the subsequent group from the first element of each group in $\vec{p}$, we obtain a vector that is equal to $\vec{q}$. That is, $\vec{q} = ((13-9), 0, (9-7), 7, 0, 0)$. This computation can be performed by applying π, subtracting the succeeding element from each element, and applying π^{-1}, in that order to $\vec{p}$, where π is a permutation representing a stable sort by $\neg\vec{g}$, which moves the first element of each group forward. Thus, $\vec{q}$ is computed by $\pi^{-1}(\mathsf{PrefixSumR}^{-1}(\pi(\mathsf{PrefixSumR}(\vec{x}) \times \vec{g})))$.

In (2), applying PrefixSum to $\vec{q}$ seems to give a value close to $\vec{y}$, but in fact $\mathsf{PrefixSum}(\vec{q}) = (4, 4, 6, 13, 13, 13)$. To compute $\vec{y}$, we need to apply PrefixSum to $\vec{d} := (4, 0, -2, 5, 0, 0)$ instead of $\vec{q}$. Comparing $\vec{q}$ and $\vec{d}$, we see that $\vec{d}$ is equal to a vector obtained by subtracting the first element of the preceding group from the first element of each group in $\vec{q}$. That is, $\vec{d} = (4, 0, (2-4), (7-2), 0, 0)$. This computation can be performed by applying π, subtracting the preceding element from each element, applying π^{-1}, and multiplying $\vec{g}$, in that order to $\vec{q}$. Thus, $\vec{y}$ is computed by $\mathsf{PrefixSum}(\pi^{-1}(\mathsf{PrefixSum}^{-1}(\pi(\vec{q}))) \times \vec{g})$.

Protocol. The protocol for group-wise sum is shown in Algorithm 11. It can be seen that the protocol is a straightforward concatenation of (1) and (2) above, except that $\pi(\pi^{-1}(\cdot))$ is removed. A small example appears in Fig. 2.

Correctness. Let r be the number of groups in the input. In Lines 1 to 3, we compute $[\![\vec{s}_1]\!]$ so that for each $j \in [1, r]$, $\vec{s}_1[j]$ is the sum in the j-th group in $\vec{x}$. This follows from the fact that the collection of the first elements of each group in $\vec{p}$ is equal to the reverse-ordered prefix sum of the sums in each group in $\vec{x}$. Next, in Lines 4 to 7, we copy each $\vec{s}_1[j]$, which is the sum in the j-th group of $\vec{x}$, to each element of the j-th group. To do this, we apply the technique used by Laud in his parallel reading protocol [19] as follows. We use the fact that when the prefix sum is computed, an element with a value of zero will be copied with the result of the preceding element. Specifically, in Line 5, the values are restored to their original order so that the first element of each group becomes the sum of each group and the other elements become zero, and in Line 7, the prefix sum of the entire vector is computed. However, this will copy the prefix sum of the sums instead of the sums. Therefore, the inverse operation for the prefix sum is preliminarily performed in Line 4.

Let the input values be as follows.

$\vec{g}$	1	0	1	1	0	0
$\vec{x}$	3	1	2	2	3	2

In Line 1, we compute $\vec{p}$.

$\vec{p}$	13	0	9	7	0	0

It is then stably sorted by $\neg\vec{g}$ and we obtain $\vec{p}_1$ in Line 3. In Lines 4 and 5, $\vec{s}_1$ and $\vec{d}_1$ are computed, respectively.

$\vec{p}_1$	13	9	7	0	0	0
$\vec{s}_1$	4	2	7	0	0	0
$\vec{d}_1$	4	-2	5	-7	0	0

Then, the order is reversed and we obtain $\vec{d}$ by multiplying $\vec{g}$. Finally, we apply PrefixSum and obtain $\vec{y}$ in Line 7.

$\vec{d}$	4	0	-2	5	0	0
$\vec{y}$	4	4	2	7	7	7

Figure 2: Example of group-wise sum according to Algorithm 11

Complexity. The protocol uses $O(n \log n)$ non-free operations in $O(\log n)$ rounds.

A.2 Alternative protocol for group-wise prefix sum

Algorithm 12: Alternative group-wise prefix sum.

Notation: $[\![\vec{y}]\!] \leftarrow \mathsf{GroupPrefixSum2}([\![\vec{g}]\!], [\![\vec{x}]\!])$.
Input: A group flag vector $[\![\vec{g}]\!]$ of length n and a internally grouped vector $[\![\vec{x}]\!]$ of length n to be aggregated.
Output: An aggregation result vector $[\![\vec{y}]\!]$ of length n.
Cost: $O(n \log n)$ non-free operations in $O(\log n)$ rounds.

1 $[\![\vec{s}]\!] \leftarrow \mathsf{PrefixSum}([\![\vec{x}]\!])$.
2 $[\![\vec{q}[1]]\!] \leftarrow 0$ and $[\![\vec{q}[i]]\!] \leftarrow [\![\vec{s}[i-1]]\!] \times [\![\vec{g}[i]]\!]$ for $i \in [2, n]$.
3 $[\![\vec{y}]\!] \leftarrow [\![\vec{s}]\!] - \mathsf{GroupSum2}([\![\vec{g}]\!], [\![\vec{q}]\!])$.

Intuition. The important observation is that the prefix sum of $\vec{x}$ is very close to the $\vec{y}$ that we want to compute. In fact, every element is greater than $\vec{y}$ by exactly the sum up to the preceding group in $\vec{x}$. Since this difference can be easily computed from $\vec{s}$ and $\vec{g}$, we can subtract it to obtain $\vec{y}$.

Protocol and its correctness. The protocol is shown in Algorithm 12. We first compute $[\![\vec{s}]\!]$, which is the prefix sum of $[\![\vec{x}]\!]$ (Line 1). This looks almost done, but each value $\vec{s}[i]$ exceeds the desired value by

Let the input values be as follows.

$\vec{g}$	1	0	1	1	0	0
$\vec{x}$	3	1	2	2	3	2

In Line 1, the prefix sum $\vec{s}$ is computed.

$\vec{s}$	3	4	6	8	11	13

In Line 2, we compute $\vec{q}$. Then, $\vec{a}$, which is the result of GroupSum2 in Line 3, is obtained by duplicating the value of the first element in each group of q.

$\vec{q}$	0	0	4	6	0	0
$\vec{a}$	0	0	4	6	6	6

We subtract $\vec{a}$ from $\vec{s}$ to obtain the output $\vec{y}$.

$\vec{y}$	3	4	2	2	5	7

Figure 3: Example of group-wise prefix sum according to Algorithm 12

a partial sum from the first element of $\vec{x}$ to the last element of the preceding group in $\vec{x}$. Therefore, we try to subtract these partial sums from $[\![\vec{s}]\!]$ and obtain the desired output. The predecessor of the first element in the j-th group in $\vec{s}$ is equal to the partial sum from the first element in $\vec{x}$ to the last element in the $(j-1)$st group of $\vec{x}$. Using this property, we construct a vector $[\![\vec{q}]\!]$ that contains such values as the first values of the groups (Line 2). We then copy the first elements of each group in $[\![\vec{q}]\!]$ to other elements by applying GroupSum2 protocol to $[\![\vec{q}]\!]$. Finally, we subtract this from $[\![\vec{s}]\!]$ to obtain the prefix sum for each group (Line 3). A small example appears in Fig. 3.

Complexity. The protocol uses $O(n \log n)$ non-free operations in $O(\log n)$ rounds.

A.3 Alternative protocol for group-wise max

Algorithm 13: Alternative group-wise max.

Notation: $[\![\vec{y}]\!] \leftarrow \mathsf{GroupMax2}([\![\vec{g}]\!], [\![\vec{x}]\!])$.
Input: A group flag vector $[\![\vec{g}]\!]$ of length n and a internally grouped vector $[\![\vec{x}]\!]$ of length n to be aggregated.
Output: An aggregation result vector $[\![\vec{y}]\!]$ of length n.
Cost: $O(n \log n)$ non-free operations in $O(\log n)$ rounds.

1 Let $[\![\vec{x}']\!]$ be the aggregation result vector obtained by applying Algorithm 2 to $[\![\vec{g}]\!]$ and $[\![\vec{x}]\!]$ with respect to Max.
2 $[\![\vec{t}[i]]\!] \leftarrow [\![\vec{g}[i+1]]\!]$ for $i \in [1, n)$ and $[\![\vec{t}[n]]\!] \leftarrow 1$.
3 $[\![\vec{y}]\!] \leftarrow \mathsf{GroupSum2}([\![\vec{g}]\!], [\![\vec{t}]\!] \times [\![\vec{x}']\!])$.

Intuition. First, we compute the group-wise prefix max by applying Algorithm 2 with respect to Max. Then, we convert it to the group-wise max in a manner similar to Algorithm 3, but without using GroupPrefixSum. Since the last element for each group is equal to the max value in the group, we can compute group-wise max by clearing all but the last elements and copying the last elements by GroupSum2.

Let the input values be as follows.

$\vec{g}$	1	0	1	1	0	0
$\vec{x}$	3	1	2	2	3	2

The group-wise prefix max $\vec{x}'$ is computed as follows.

$\vec{x}'$	3	3	2	2	3	3

In Line 2, $\vec{t}$ is computed from $\vec{g}$.

$\vec{t}$	0	1	1	0	0	1

Finally, $\vec{y}$ is computed by copying the last elements for each group in $\vec{x}'$.

$\vec{y}'$	3	3	2	3	3	3

Figure 4: Example of group-wise max according to Algorithm 13

Protocol and its correctness. The protocol is shown in Algorithm 13. The group-wise prefix max is computed in Line 1. Since the successor of the last element in a group is the first element in the succeeding group, $\vec{t}$ is a flags vector indicating whether it is the last element in a group. In Line 3, all but the last elements for each group are cleared to 0 by $\vec{t} \times \vec{x}'$. Since the last elements are equal to the max values in each group, GroupSum2 copies these values and the group-wise max is computed. A small example appears in Fig. 4.

Complexity. Algorithm 2 uses $O(n)$ operations in $O(\log n)$ rounds. Since GroupSum2 uses $O(n \log n)$ non-free operations in $O(\log n)$ rounds, the group-wise max protocol also uses $O(n \log n)$ non-free operations in $O(\log n)$ rounds.

B ADDITIONAL BENCHMARK RESULTS

Additional benchmark results for Algorithm 10 are shown in Tables 7 to 9.

C SUMMARY OF VARIABLES AND PROTOCOLS

For the reader's convenience, we summarized main variables and protocols in Tables 10 and 11, respectively.

D EXAMPLES OF OUR PROTOCOLS

We provide some examples to help understand the proposed protocols. Examples are shown in Figs. 5 to 11.

Table 7: Running time for training a decision tree of height $h = 4$ on different numbers of samples n with $m = 11$ input variables.

	Time [s]		
n	T1	T2	T3
$2^8 = 256$	1.733	2.762	406.369
$2^9 = 512$	2.014	3.019	432.122
$2^{10} = 1024$	2.594	3.252	458.022
$2^{11} = 2048$	3.438	3.614	484.189
$2^{12} = 4096$	5.762	4.171	512.702
$2^{13} = 8192$	10.333	4.798	541.054
$2^{14} = 16384$	18.902	6.463	572.145
$2^{15} = 32768$	36.385	9.679	610.332
$2^{16} = 65536$	71.467	16.954	663.163
$2^{17} = 131072$	174.855	37.952	780.339
$2^{18} = 262144$	349.515	82.399	969.883
$2^{19} = 524288$	701.331	178.831	1366.467
$2^{20} = 1048576$	1466.253	403.804	2117.820

Table 8: Running time for training a decision tree of height $h = 4$ on $n = 2^{13}$ samples with different numbers of input variables m.

	Time [s]		
m	T1	T2	T3
1	2.014	3.077	420.211
2	2.741	3.286	441.340
3	3.519	3.437	462.295
4	4.542	3.548	471.273
5	5.392	3.761	480.083
10	9.497	4.639	532.876
11	10.333	4.798	541.054
20	17.380	6.425	614.989
50	41.565	11.981	823.540
100	81.582	21.976	1166.079

Table 9: Running time for training a decision tree of different heights h on $n = 2^{13}$ samples with $m = 11$ input variables.

	Time [s]		
h	T1	T2	T3
1	3.412	1.479	184.366
2	5.716	2.546	303.245
3	8.006	3.650	422.134
4	10.333	4.798	541.054
5	12.651	5.911	659.810
6	14.942	7.148	778.787
7	17.267	8.203	897.683
8	19.403	9.141	1016.427
9	21.675	10.302	1135.328
10	23.981	11.130	1254.291
11	26.091	12.534	1373.173
20	46.760	22.432	2443.332
30	70.004	33.211	3632.579
40	93.310	43.612	4821.568
50	116.641	55.234	6011.040

Let the input values be as follows. Double lines are drawn to make it easier to identify group boundaries.

$\vec{g}$	1	0	1	1	0	0
$\vec{y}$	0	0	1	1	1	0

In Line 1, Algorithm 4 prepares the vectors $\vec{y}_0$ and $\vec{y}_1$.

$\vec{y}_0$	1	1	0	0	0	1
$\vec{y}_1$	0	0	1	1	1	0

In Lines 2 to 5, group-wise operations are used to compute the values required by Equation (1). Specifically, we count the number of 0's and 1's up to and after the i-th row of each group, respectively. They are the following.

$\vec{u}_0$	1	2	0	0	0	1
$\vec{u}_1$	0	0	1	1	2	2
$\vec{v}_0$	2	2	0	1	1	1
$\vec{v}_1$	0	0	1	2	2	2
$\vec{w}_0$	1	0	0	1	1	0
$\vec{w}_1$	0	0	0	1	0	0
$\vec{u}$	1	2	1	1	2	3
$\vec{w}$	1	0	0	2	1	0

In Lines 6 to 8, the modified Gini indices $\vec{\mathsf{S}}$ are computed element by element according to Equation (1), where $\perp$ represents an indefinite value by division by zero. They appear only in the last element of a group and are later overwritten by other values.

$\vec{p}$	2	0	0	4	6	0
$\vec{q}$	1	0	0	2	2	0
$\vec{\mathsf{S}}$	2	$\perp$	$\perp$	2	3	$\perp$

Figure 5: Example of modified Gini index computation according to Algorithm 4

Let the input values be as follows.

$\vec{g}$	1	0	1	1	0	0
$\vec{x}$	1	3	5	4	4	5
$\vec{y}$	0	0	1	1	1	0

In Line 1, Algorithm 5 computes the scores $\vec{\mathsf{S}}$, where $\perp$ represents an indefinite value by division by zero. In Line 2, it prepares the thresholds $\vec{\mathsf{T}}$. They are shown below. Note that some of the last values in each group are incorrect, but this is not a problem since they will be overwritten later.

$\vec{\mathsf{S}}$	2	$\perp$	$\perp$	2	3	$\perp$
$\vec{\mathsf{T}}$	4	8	9	8	9	$-\infty$

In Line 3, points that cannot be split immediately after themselves are detected as $\vec{p}$. Then, the corresponding elements in $\vec{\mathsf{S}}$ and $\vec{\mathsf{T}}$ are overwritten to $-\infty$ in Line 4.

$\vec{p}$	0	1	1	1	0	1
$\vec{\mathsf{S}}$	2	$-\infty$	$-\infty$	$-\infty$	3	$-\infty$
$\vec{\mathsf{T}}$	4	$-\infty$	$-\infty$	$-\infty$	9	$-\infty$

In Line 5, for each group, the values corresponding to the element with the largest $\vec{\mathsf{S}}$ are copied within the group.

$\vec{\mathsf{S}}$	2	2	$-\infty$	3	3	3
$\vec{\mathsf{T}}$	4	4	$-\infty$	9	9	9

Figure 6: Example of batch attribute-wise test selection according to Algorithm 5

Table 10: Summary of variables.

Notation	Description
n	Number of samples
m	Number of input attributes
h	Tree height
$\mathsf{Layer}^{(k)}$	Layer information at k-th layer
$\mathsf{NID}^{(k)}$	Formatted node numbers at k-th layer
$\mathsf{AID}^{(k)}$	Formatted attribute numbers at k-th layer
$\mathsf{Threshold}^{(k)}$	Formatted thresholds at k-th layer
$\mathsf{Label}^{(k)}$	Formatted leaf labels at k-th layer
$\vec{x}_i$	i-th input attribute
$\vec{y}$	Output attribute
$\vec{g}$	Group flag vector
$\vec{b}$	Test results
$\vec{\mathsf{N}}$	Node numbers
$\vec{\mathsf{A}}$	Attribute numbers
$\vec{\mathsf{T}}$	Thresholds
$\vec{\mathsf{L}}$	Leaf labels
$\vec{\mathsf{S}}$	Scores

Table 11: Summary of protocols.

Notation	Description	Reference
GroupSum	Group-wise sum	Instantiated from Algorithm 3
GroupPrefixSum	Group-wise prefix sum	Instantiated from Algorithm 2
GroupMax	Group-wise max	Instantiated from Algorithm 3
ModifiedGini	Modified Gini index	Algorithm 4
AttributewiseTestSelection	Batch attribute-wise test selection	Algorithm 5
GlobalTestSelection	Batch global test selection	Algorithm 6
FormatLayer	Summarizing a privately grouped vector	Algorithm 7
TrainInternalNodes	Training internal nodes	Algorithm 8
TrainLeafNodes	Training leaf nodes	Algorithm 9
DecisionTreeTraining	Decision tree training	Algorithm 10
GroupSum2	Alternative group-wise sum	Algorithm 11
GroupPrefixSum2	Alternative group-wise prefix sum	Algorithm 12
GroupMax2	Alternative group-wise max	Algorithm 13

Suppose we are given the following input attribute vectors $\vec{x}_j$ $(j = 1, 2)$ and an output attribute vector $\vec{y}$ that are privately grouped by a group flag vector $\vec{g}$.

$\vec{x}_1$	4	3	4	4	5	5
$\vec{x}_2$	3	1	5	2	5	3
$\vec{y}$	0	0	1	1	0	1
$\vec{g}$	1	0	1	1	0	0

First, let's look at the first input attribute $\vec{x}_1$. In Line 2, $\vec{x}_1$ and $\vec{y}$ are sorted within each group with respect to $\vec{x}_1$, and the resulting vectors are denoted $\vec{u}_1$ and $\vec{v}_1$, respectively. Then, in Line 3, AttributewiseTestSelection computes the optimal thresholds $\vec{\mathsf{T}}_1$ and the scores $\vec{\mathsf{S}}_1$ for the test with the first input attribute.

$\vec{u}_1$	3	4	4	4	5	5
$\vec{v}_1$	0	0	1	1	0	1
$\vec{\mathsf{T}}_1$	7	7	$-\infty$	9	9	9
$\vec{\mathsf{S}}_1$	2	2	$-\infty$	2	2	2

In addition, the optimal thresholds $\vec{\mathsf{T}}_2$ and the scores $\vec{\mathsf{S}}_2$ are computed for the second input attribute in parallel as well.

$\vec{u}_2$	1	3	5	2	3	5
$\vec{v}_2$	0	0	1	1	1	0
$\vec{\mathsf{T}}_2$	4	4	$-\infty$	8	8	8
$\vec{\mathsf{S}}_2$	2	2	$-\infty$	3	3	3

Finally, in Lines 5 and 6, we select the test with the highest score for each group.

$\vec{\mathsf{A}}$	1	1	1	2	2	2
$\vec{\mathsf{T}}$	7	7	$-\infty$	8	8	8

Figure 7: Example of batch global test selection according to Algorithm 6

Let $w = 4$. Let other input values be as follows.

$\vec{g}$	1	0	1	1	0	0
$\vec{u}$	7	7	3	8	8	8

In Line 1, we overwrite all but the first element of each group in $\vec{u}$ with NULL to obtain $\vec{r}$ and further sort in descending order by $\vec{g}$ to obtain $\vec{v}$.

$\vec{r}$	7	NULL	3	8	NULL	NULL
$\vec{v}$	7	3	8	NULL	NULL	NULL

In Line 3, the first w elements in $\vec{v}$ are output as $\vec{z}$.

$\vec{z}$	7	3	8	NULL

Figure 8: Example of summarizing according to Algorithm 7

Let the input values be as follows.

$\vec{x}_1$	4	3	4	4	5	5
$\vec{x}_2$	3	1	5	2	5	3
$\vec{y}$	0	0	1	1	0	1
$\vec{g}$	1	0	1	1	0	0
$\vec{\mathsf{N}}$	1	1	2	3	3	3

In Line 1, the best tests $\vec{\mathsf{A}}$ and $\vec{\mathsf{T}}$ are computed considering only the stopping criterion (2).

$\vec{\mathsf{A}}$	1	1	1	2	2	2
$\vec{\mathsf{T}}$	7	7	$-\infty$	8	8	8

In Lines 2 to 4, nodes satisfying the stopping criterion (2) are detected as $\vec{f}$.

$\vec{c}$	2	2	1	3	3	3
$\vec{c}_0$	2	2	0	1	1	1
$\vec{c}_1$	0	0	1	2	2	2
$\vec{f}$	1	1	1	0	0	0

In Line 5, for each element $\vec{f}[i]$ such that $\vec{f}[i] = 1$, we overwrite $\vec{\mathsf{A}}[i]$ and $\vec{\mathsf{T}}[i]$ with 1 and $-\infty$. They represent the test $2X_1 < \mathsf{MIN\_VALUE}$ which is never satisfied.

$\vec{\mathsf{A}}$	1	1	1	2	2	2
$\vec{\mathsf{T}}$	$-\infty$	$-\infty$	$-\infty$	8	8	8

In Line 6, $\vec{\mathsf{N}}$, $\vec{\mathsf{A}}$, and $\vec{\mathsf{T}}$ are summarized by FormatLayer and output as $[\![\mathsf{Layer}^{(k)}]\!]$ = (NID, AID, Threshold).

NID	1	2	3	NULL
AID	1	1	2	NULL
Threshold	$-\infty$	$-\infty$	8	NULL

In Lines 7 and 8, the input attribute values specified by attribute numbers $\vec{\mathsf{A}}$ is computed and set to $\vec{x}_*$. Then, the test result $\vec{b}$ is computed by comparing $2 \times \vec{x}_*$ and $\vec{\mathsf{T}}$ in Line 9.

$\vec{e}_1$	1	1	1	0	0	0
$\vec{e}_2$	0	0	0	1	1	1
$\vec{x}_*$	4	3	4	2	5	3
$\vec{b}$	0	0	0	1	0	1

Figure 9: Example of internal node training according to Algorithm 8

Let the input values be as follows.

$\vec{g}$	1	0	0	1	0	1
$\vec{y}$	1	0	1	0	1	0
$\vec{\mathsf{N}}$	1	1	1	2	2	3

In Lines 1 and 2, Algorithm 7 computes the most frequent values of $\vec{y}$ for each group and as $\vec{\mathsf{L}}$.

$\vec{c}_0$	1	1	1	1	1	1
$\vec{c}_1$	2	2	2	1	1	0
$\vec{\mathsf{L}}$	1	1	1	0	0	0

In Line 3, $\vec{\mathsf{N}}$ and $\vec{\mathsf{L}}$ are summarized by FormatLayer and output as $[\![\mathsf{Layer}^{(h)}]\!]$ = $([\![\mathsf{NID}]\!]^{(h)}, [\![\mathsf{Label}]\!]^{(h)})$.

$\mathsf{NID}^{(h)}$	1	2	3	NULL
$\mathsf{Label}^{(h)}$	1	0	0	NULL

Figure 10: Example of leaf node training according to Algorithm 9

Let $h = 2$. Let the input values and initialized values be as follows. These are the input to the 0-th layer.

$\vec{x}_1^{(0)}$	3	5	2	5	4	4
$\vec{x}_2^{(0)}$	3	2	5	4	5	1
$\vec{y}^{(0)}$	0	1	0	0	1	0
$\vec{g}^{(0)}$	1	0	0	0	0	0
$\vec{\mathsf{N}}^{(0)}$	1	1	1	1	1	1

At the first iteration, the layer information and test results are computed in Line 4.

$\mathsf{NID}^{(0)}$	1
$\mathsf{AID}^{(0)}$	1
$\mathsf{Threshold}^{(0)}$	7

$\vec{b}$	1	0	1	0	0	0

Then, $\vec{\mathsf{N}}$ and $\vec{g}$ are updated using $\vec{b}$ in Lines 5 and 6.

$\vec{\mathsf{N}}_+$	2	1	2	1	1	1
$\vec{g}_+$	1	1	0	0	0	0

In Line 7, current dataset is sorted by $\vec{b}$ and we obtain the inputs for the 1-th layer.

$\vec{x}_1^{(1)}$	5	5	4	4	3	2
$\vec{x}_2^{(1)}$	2	4	5	1	3	5
$\vec{y}^{(1)}$	1	0	1	0	0	0
$\vec{g}^{(1)}$	1	0	0	0	1	0
$\vec{\mathsf{N}}^{(1)}$	1	1	1	1	2	2

The same computation is repeated at the second iteration.

$\mathsf{NID}^{(1)}$	1	2
$\mathsf{AID}^{(1)}$	2	1
$\mathsf{Threshold}^{(1)}$	3	$-\infty$

$\vec{b}$	0	0	0	1	0	0
$\vec{\mathsf{N}}_+$	1	1	1	3	2	2
$\vec{g}_+$	1	0	0	1	1	0

$\vec{x}_1^{(2)}$	5	5	4	3	2	4
$\vec{x}_2^{(2)}$	2	4	5	3	5	1
$\vec{y}^{(2)}$	1	0	1	0	0	0
$\vec{g}^{(2)}$	1	0	0	1	0	1
$\vec{\mathsf{N}}^{(2)}$	1	1	1	2	2	3

Finally, the layer information for h-th layer is computed in Line 8.

$\mathsf{NID}^{(2)}$	1	2	3	-1
$\mathsf{Label}^{(2)}$	1	0	0	-1

Figure 11: Example of decision tree training according to Algorithm 10