Greenwoods: A Practical Random Forest Framework for Privacy Preserving Training and Prediction

Harsh Chaudhari
Indian Institute of Science
Bangalore, Kanrantaka, India
chaudharim@iisc.ac.in

Peter Rindal
Visa Research
Palo Alto, CA 94306, United States
PeterRindal@gmail.com

Abstract

In this work we propose two prediction protocols for a random forest model. The first takes a traditional approach and requires the trees in the forest to be complete in order to hide sensitive information. Our second protocol takes a novel approach which allows the servers to obliviously evaluate only the "active path" of the trees. This approach can easily support trees with large depth while revealing no sensitive information to the servers. We then present a distributed framework for privacy preserving training which circumvents the expensive procedure of privately training the random forest on a combine dataset and propose an alternate efficient collaborative approach with the help of users participating in the training phase.

1 Introduction

Random Forest is one of the most widely deployed machine learning (ML) models with numerous applications ranging from healthcare, finance, spam filtering and many more. At its core a random forest is a collection of decision trees where each tree is a function that takes a vector of *features* as input and outputs a *label* which predicts which class this vector of features belongs to. In the context of machine learning a decision tree is constructed by taking many examples of feature-label pairs and learning the correlation between the two. This process is also referred to as training phase of the decision tree. Later, the prediction phase is performed where the trained decision trees in the random forest are used to predict the correct label for a new feature vector.

This work considers the problem of evaluating a random forest in a privacy preserving manner using secure multiparty computation (MPC) [Yao82, BGW88, GMW87, IKNP03, DPSZ12]. Our protocols are in the outsourced setting with three servers and at most one corruption [BGW88, MRZ15, ABF$^+$16, BJPR18, CCPS19]. That is, we consider the setting where a *model owner* submits encrypted decisions trees to the servers. Later, a *client* may submit his/her query (an unlabeled feature vector). The server should then evaluate the encrypted trees on this query and output the most common class label among the trees. In the end, the client should only learn the label to his/her query and the servers should learn no information about the decision trees or the query.

Given such a functionality several applications are possible. For example, the model itself may contain proprietary data which they are unwilling to share. Similarly, the client may also consider their query as sensitive information. For example, consider a diagnostic model [TASF09] which predicts the presence of a disease. This model could have been trained by some third party using proprietary techniques and likely used a large dataset of sensitive patient data which makes sharing the model difficult. A patient and/or their doctor may wish to evaluate the patient symptoms to the model but do not desire to disclose his/her data. In this case the functionality described above immediately allows all parties to maintain their desired privacy. The servers which will hold the encrypted data could be hosted by well regarded medical institutions and/or on various cloud providers.

1.1 Our Contribution

We propose two approaches for secure prediction of decision trees. The first approach is a traditional one, where we evaluate the entire tree to get a final prediction, which requires only $\mathcal{O}(\log d)$ rounds but with an exponential communication of $\mathcal{O}(2^d)$ bits, where d is the depth of the tree. Thus the traditional approach is ideal for the case of short depth trees but quickly becomes impractical for larger values of d, where many of the real world datasets operate. As a consequence, our second approach takes a more practical outlook where a single prediction takes $\mathcal{O}(d)$ rounds with only $\mathcal{O}(n+m)$ bits of communication, where n is number of features and m is number of internal nodes. In our empirical tests, this protocol is able to evaluate 400 trees each with depth 80 in 185 milliseconds and 111KB of communication per tree.

Training random forest privately in a MPC setting has long been an expensive procedure even with [WFNL16, LP00] proposing various approximations. Thus, we propose a more efficient collaborative approach of training the random forest in our setting. Specifically all users participating in the training phase of the framework train an ensemble of decision trees on their individual datasets and share the resulting trees and the training data among the servers which then are used to form the random forest. We also propose approaches to tackle edge case scenarios where users have drastically different data distributions from one another. Our approach works for various real world scenarios with accuracy drop ranging $0\% - 3\%$ compared to plain-text training, across multiple datasets.

1.2 ABY3 Framework & Corruption Model

Our protocols can be viewed as an application of the ABY3 framework[MR18]. This work inherits their semi-honest security model. We denote the three servers as P_1, P_2, P_3. In addition, we consider the setting with one or more users which provide secret shared inputs among the three servers. All parties are assumed to be semi-honest. The users can collude with any single server. That is, the joint views of the corrupt users and any single server along with the output can be simulated given their inputs and output.

2 Random Forest Prediction

2.1 Dense: Full Tree Evaluation

This approach evaluates the entire tree to get the final prediction. First we will loosely describe the flow of the protocol without concern to how each step is securely implemented. The detailed secure version is provided in Appendix B.1. The decision tree can be visualized as an array with the first entry being the root node with index $i = 1$. A node at position $i \in [n]$ has a left and right child at $2i$ and $2i + 1$. Notice that, there is an unique path from the root node to each leaf node, where each leaf node represents an output class label. For each leaf node there is a specific set of comparison values along this path which means this leaf holds the output label. Thus given a new query $\mathbf{z}$, we simultaneously compute all the paths from the root to each leaf node in parallel and exactly one out of all possible paths is active giving us the output label corresponding to $\mathbf{z}$. One requirement of this technique is that the tree needs to be complete. This is due to the structure of the binary tree being considered private information as it is a function of the training data. In the case that the decision tree is not complete the model owner adds "dummy" nodes [WFNL16, JPVE07] until it is. This leads a round optimal protocol with $\mathcal{O}(\log d)$ rounds but the communication cost explodes with an exponential communication of $\mathcal{O}(2^d)$ bits, where d is the depth of the tree.

2.2 Sparse: Active Path Evaluation:

Approach II circumvents the need to pad the tree to be complete by obliviously evaluating only the "active path" of the tree. A tree is represented using a "linked list" type of data structure. Internal node i will have children nodes with indexes r_i, l_i, a threshold t_i, and a feature index $j_i \in [m]$ denoting that input feature f_{j_i} should be compared with t_i. When evaluating, the parties first randomly shuffle the order of the nodes and the order of the input features. Through some oblivious process we also update the r_i, l_j to point to the new locations of these child nodes. Similarly, all j_i values are updated to index the new shuffled location of the desired input feature. Evaluation proceeds by identifying the root node i and revealing the *shuffled* feature index $j_i \in [m]$. All parties can then compare t_i with

f_{j_i} within an MPC and reveal the *shuffled* index of the left or right child. This process is repeated a total of d times at which point a leaf node has been reached. This leaf node i will contain a label ℓ_i which is output. This approach becomes more practical for large depth trees requiring $\mathcal{O}(d)$ rounds and with only $\mathcal{O}(n+m)$ bits of communication, where n is number of features and m is number of internal nodes. The secure version of the above protocol is deferred to Appendix B.2. The outputs of these decision trees are then combined to get the final output of the random forest.

With respect to security, the main concern is if information is leaked when the shuffled position of the input feature j_i is revealed or when the shuffle position of the next node $x_i \in \{r_i, l_i\}$ is revealed. This turns out not to be an issue. First, no feature or node is ever used twice on a single root to leaf path. Secondly, only the shuffled position is revealed and not the actual logical feature/node index.

3 Random Forest Training

3.1 Traditional Approach of Training

We consider a set of users $\mathrm{u}_1, \ldots, \mathrm{u}_m$ who want to train a joint random forest model $\mathcal{M}$ by pooling in their local datasets $D_1, \ldots, D_m$ respectively. The users secret share their respective datasets to the servers who then train a random forest model on the combined dataset $\mathcal{D} = \{D_1, \ldots, D_m\}$. Recall that a random forest is a ensemble of decision trees. Thus the cost of training a random forest comes down to the cost of training a single decision tree times the number of decision trees in the forest. Training a decision tree involves computing the entropy of each attribute/ feature to create a candidate split. In our setting, the calculation of entropy for a single candidate split in the tree requires computation of enormous number of division and logarithmic circuits followed by a huge circuit to to obliviously split the training set across the split. The aforementioned steps are required to just compute one candidate split in a tree. Thus to train a random forest model having multiple trees with this approach will make it almost impractical in our setting.

3.2 Our Approach of Training

We propose an alternate efficient solution which exploits the properties of a random forest. A Random Forest is built by training multiple decision trees on bootstrapped datasets. The crucial idea behind our approach is to view the datasets $\{D_1, \ldots, D_m\}$ owned by the users themselves as bootstrapped datasets coming from $\mathcal{D}$. In other words this can also be viewed as sampling *without replacement* from $\mathcal{D}$ which in turn create our bootstrapped datasets $\{D_1, \ldots, D_m\}$. Thus each user can locally train a decision tree (or a group of trees) on his own dataset which when pooled together with other trees (trained by the remaining users) forms our trained random forest model $\mathcal{M}$. Note that, this is an approximation to the original technique of training the random forest model directly on $\mathcal{D}$ and we are likely to observe some accuracy drop when shifting from the traditional approach. In our experiments, we show that the drop in accuracy (across multiple datasets) is quite reasonable given the efficiency we achieve. Also Note that, our proposed approach works well for most of the real world scenarios, where the underlying distribution of the participating users datasets are similar to one another.

For instance, say multiple hospitals together wanted to train a random forest model on their combined patient dataset to predict if a person has lung cancer or not; Using our aforementioned approach each hospital will secret share its locally trained DTs to our three-server setting. The underlying assumption for our approach to work is that each hospital requires at least some samples associated with both patients with and without lung cancer. It is not necessary for each hospital to have identical number of samples. Generally we assume each users' dataset has its data distribution similar to other users. However, in some cases this does not hold, e.g. Say one subset of hospitals only have samples associated with patients having lung cancer and the remaining subset have patient data with no lung cancer. Thus the DTs trained on each indiviual hospital's dataset would have only seen samples of one class and hence would be incapable of predicting any new query belonging to the other unseen class. More formally, the random forest model, constructed by combining the decision trees secret shared by the users, becomes incompatible when the data distribution of each participating user is drastically different from one another. For such cases we require additional tools to make our approach compatible. The details for tackling such edge case scenarios are given in Appendix C.

4 Experimental Results

Experimental Setup: We measure the performance of our prediction protocols on a single commodity laptop with an i5-8365U and 16GB of memory. The network latency was measured to be 0.2 milliseconds with 8Gbps throughput. Given that the benchmark machine is a constrained device we would expect several times speedup in throughput one server grade hardware. In the training phase, the collection of trees for each individual user is trained using SkLearn Library [PVG$^+$11]. Additionally the non-identical behavior among the users are simulated using Dirchlet distribution on the class labels with the help of Numpy Library [VCV11]. Our implementation will be made publicly available.

4.1 Our Prediction Protocols

We now discuss the relative performance of our two protocols, sparse and dense evaluations. We report the running time in milliseconds and total communication overhead in bytes. Each servers sends approximately one third of the reported communication. We evaluate $t = 400$ trees with $n \in \{100, 1000\}$ features, feature bit count $\sigma\{1, 16\}$, depth $d \in \{5, 10, 40, 80\}$, nodes per tree $m \in \{200, 1000, 4000\}$. Table 2 (Appendix D.2) contains an abridged table of performance results comparing our two protocols. The main observation is that when a tree of depth d has approximately $m > 2^d/4$ real nodes then the dense protocol out performs the sparse protocol in terms of running time. This result is relatively intuitive given that the dense protocol requires less work per node in the tree. For example, with depth $d = 10$ and one thousand nodes per tree, the sparse protocol takes 154 milliseconds to evaluate 400 trees. On the other hand, the dense protocol performs the same computation in just 52 milliseconds. Moreover, the communication of the dense protocol is about $25\times$ less than the sparse protocol. When the decision tree is relatively sparse our sparse protocol continues to perform at a high level. For example, if we consider a deep tree with depth $d = 80$ we observe that the sparse protocol almost same level of performance as a shallow tree with the same number of node. For example, if we compare depth $d \in \{10, 80\}$ evaluations with $m = 1000$, we observe that the $d = 80$ evaluation is only $1.2\times$ slower while requiring effectively the same amount of communication.

We conclude that the optimal protocol to use will depend on how sparse the tree is and whether the number of features is large enough to make the superior asymptotics of the sparse protocol have a practical impact.

4.2 Our Training Approach

We test our proposed approach of training a random forest model on various real world datasets. Table 1 shows how the test accuracy is affected as more users secret share their locally trained trees. We begin by testing for the case when the random forest $\mathcal{M}$ is built using only a single users' local trees which acts as the baseline for our comparison. As we increase the number of users from 1 to 10, the test accuracies increase marginally with improvements ranging from $1.10\% - 7.25\%$ across datasets. This strengthens our claim that a collection of locally trained trees when pooled in together by multiple users perform better. As expected the overall accuracy further increases as the number of users are increased from 10 to 40.

# Users	MNIST	F-MNIST	Bank Marketing	Nomao	Breast Cancer
1	86.11%	78.81%	88.86%	92.60%	96.89%
10	93.36%	82.87%	90.02%	95.06%	97.99%
40	93.62%	83.53%	90.39%	95.08%	98.10%

Table 1: Test accuracy of RF trained using Our Approach

We now compare our approach against the traditional approach of training. The average drop in test accuracy across datasets ranges from 0.07% to 3.10% when we shift from the traditional approach of training a random forest $\mathcal{M}$ using the combined dataset $\mathcal{D}$ to our collaborative approach of users directly secret sharing their locally trained trees. We are also able to handle edge case scenarios using our additionally proposed tools (Appendix C) at the cost of a reasonable overhead. The detailed experimental results for the same are given in Appendix D.1

References

[ABF$^+$16] T. Araki, A. Barak, J. Furukawa, Y. Lindell, A. Nof, and K. Ohara. DEMO: high-throughput secure three-party computation of kerberos ticket generation. In *ACM CCS*, 2016.

[ARS$^+$15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 430–454. Springer, 2015.

[BGW88] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *ACM STOC*, 1988.

[BJPR18] M. Byali, A. Joseph, A. Patra, and D. Ravi. Fast secure computation for small population over the internet. *ACM CCS*, 2018.

[CCPS19] H. Chaudhari, A. Choudhury, A. Patra, and A. Suresh. ASTRA: High-throughput 3PC over Rings with Application to Secure Prediction. In *IACR Cryptology ePrint Archive*, 2019.

[DEF$^+$19] I. Damgård, D. Escudero, T. Frederiksen, M. Keller, P. Scholl, and N. Volgushev. New primitives for actively-secure mpc over rings with applications to private machine learning. IEEE SP, 2019.

[DPSZ12] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*, 2012.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, 1987.

[HHM19] T.M.Harry Hsu, H.Qi, and M.Brown. Measuring the effects of non-identical data distribution for federated visual classification, 2019.

[IKNP03] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending Oblivious Transfers Efficiently. In *CRYPTO*, 2003.

[JPVE07] J.Brickell, D.E. Porter, V.Shmatikov, and E.Witchel. Privacy-preserving remote diagnostics. In *Proceedings of the 2007 ACM CCS 2007*, 2007.

[LP00] Y. Lindel and B. Pinkas. Privacy preserving data mining. In *Proceedings of the 20th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '00, 2000.

[MR18] P. Mohassel and P. Rindal. ABY^3: A Mixed Protocol Framework for Machine Learning. In *ACM CCS*, 2018.

[MRR19] P. Mohassel, P. Rindal, and M. Rosulek. Fast database joins for secret shared data. *IACR Cryptology ePrint Archive*, 2019.

[MRZ15] P. Mohassel, M. Rosulek, and Y. Zhang. Fast and Secure Three-party Computation: Garbled Circuit Approach. In *CCS*, 2015.

[PVG$^+$11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 2011.

[TASF09] Ajay Kumar Tanwani, Jamal Afridi, M. Zubair Shafiq, and Muddassar Farooq. Guidelines to select machine learning scheme for classification of biomedical datasets. In Clara Pizzuti, Marylyn D. Ritchie, and Mario Giacobini, editors, *Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics*, pages 128–139, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[VCV11] S. VanDerWalt, S.C. Colbert, and G. Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 2011.

[WFNL16] D.J. Wu, T. Feng, M. Naehrig, and K. Lauter. Privately evaluating decision trees and random forests. PoPETS, 2016.

[Yao82] A. C. Yao. Protocols for Secure Computations. In *FOCS*, 1982.

A Building Blocks

A.1 Notations

We use $[\![x]\!]$ to denote that a given value x is secret shared among the servers. The exact structure/type of the secret share is context dependent. The binary sharing $[\![x]\!]^{\mathbf{B}}$ denotes that a value x is replicated secret shared among the three servers over a boolean ring $\mathbb{Z}_2$. That is, there exists (random) values $x_1, x_2, x_3 \in \mathbb{Z}_2$ such that P_1 holds (x_1, x_3), P_2 holds (x_2, x_1), P_3 holds (x_3, x_2) and $x = x_1 \oplus x_2 \oplus x_3$. We extend this notation to include the case when $x \in \mathbb{Z}_{2^\sigma}$ is a bit string. Similar, we use $[\![x]\!]^{\mathbf{A}}$ to denote an arithmetic secret sharing of $x \in \mathbb{Z}_{2^\sigma}$ such that $x = x_1 + x_2 + x_3 \mod 2^\sigma$. If $f(x, y)$ is a binary or arithmetic circuit with inputs x and y, then $f([\![x]\!], [\![y]\!])$ denotes the servers performing the ABY3 protocol[MR18] which computes $[\![f(x, y)]\!]$ on the shared inputs $[\![x]\!]$ and $[\![y]\!]$. We use the notation $[\![x]\!] \in X$ to denote that $x \in X$. Let $[m, n] = \{m, m+1, ..., n\}$ and the shorthand $[n] = [1, n]$. When assigning a variable x to y, we use the notation $x := y$ while $x = y$ denotes the statement that the two are of equal value. $x \leftarrow X$ denotes that x is uniformly sampled from the set X.

A.2 Oblivious Permutations $\mathcal{F}_{\mathsf{perm}}$

An oblivious permutation is a functionality $\mathcal{F}_{\mathsf{perm}}$ that takes a secret shared permutation $[\![\pi]\!]$ where $\pi : [m] \to [m]$ and a vector $[\![\mathbf{u}]\!]^{\mathbf{B}} \in \mathbb{Z}_{2^\sigma}^m$ as input. It outputs a shared vector $[\![\mathbf{v}]\!]^{\mathbf{B}}$ such that $\mathbf{v}_i = \mathbf{u}_{\pi(i)} \in \mathbb{Z}_{2^\sigma}$ for all i. We also extend this functionality to the setting where the output vector $\mathbf{v}$ is shorter than $\mathbf{u}$, i.e. $\pi : [n] \to [m]$ for some $n \leq m$.

We provide the protocol Π_{perm} that realize this functionality. $[\![\pi]\!]$ is represented using two shares $\pi_1 : [m] \to [m], \pi_2 : [n] \to [m]$ where π_1 is uniformly sampled from all length m permutations and π_2 is chosen such that $\pi = \pi_1 \circ \pi_2$, i.e. $\pi(i) = \pi_2(\pi_1(i))$ for $i \in [n]$. Our protocol will provide π_1 to one party, say P_1, while π_2 is provided to a different one, say P_2.

The work [MRR19] then provides a protocol $\Pi_{\mathsf{perm}'}$ which realizes a similar functionality as $\mathcal{F}_{\mathsf{perm}}$ except that π is the private input of one of the parties. Π_{perm} then realizes the functionality $\mathcal{F}_{\mathsf{perm}}$ by invoking $\Pi_{\mathsf{perm}'}$ twice. Once for the π_1 permutation and then π_2 on the result.

We also define the functionality $\mathcal{F}_{\mathsf{r-perm}}$ which takes as input the vector $[\![\mathbf{u}]\!]^{\mathbf{B}}$ and uniformly samples $\pi : [n] \to [m]$. It outputs the vector $[\![\mathbf{v}]\!]^{\mathbf{B}}$ such that $\mathbf{v}_i = \mathbf{u}_{\pi(i)} \in \mathbb{Z}_{2^\sigma}$. Protocol $\Pi_{\mathsf{r-perm}}$ realizes this by having one party uniformly sample $\pi_1 : [m] \to [m]$ while a different party uniformly samples $\pi_2 : [n] \to [m]$. The parties then invoke the protocol $\Pi_{\mathsf{perm}'}$ twice as described above.

Both protocols require $O(m\sigma)$ bits of communications and can be realized with information theoretic security[MRR19].

A.3 Oblivious Maps $\mathcal{F}_{\mathsf{omap}}$

Our protocols will utilize a functionality $\mathcal{F}_{\mathsf{omap}}$ which takes as input a secret share vector $[\![\mathbf{u}]\!]^{\mathbf{B}} \in \mathbb{Z}_{2^\sigma}^m$ and a secret shared function $[\![\theta]\!]$ where $\theta : [n] \to [m]$ for some n. The functionality outputs the shared vector $[\![\mathbf{v}]\!]^{\mathbf{B}} \in \mathbb{Z}_{2^\sigma}^n$ where $v_i = u_{\theta(i)}$ for $i \in [n]$.

Our protocol $\Pi_{\mathsf{ip-map}}$ represents θ as a matrix $\mathbf{M} \in \{0,1\}^{m \times n}$ where the ith column vector has weight one with $m_{\theta(i),i} = 1$. Then it holds that $\mathbf{Mu} = \mathbf{v}$. Given that $\mathbf{M}$ is shared as $[\![\mathbf{M}]\!]^{\mathbf{B}}$, directly computing $[\![\mathbf{v}]\!]^{\mathbf{B}} := \left([\![\mathbf{M}]\!]^{\mathbf{B}}[\![\mathbf{u}^{\mathrm{T}}]\!]^{\mathbf{B}}\right)^{\mathrm{T}}$ requires just $O(n\sigma)$ bits of communication and $O(nm\sigma)$

computation. This follows from the fact that in our setting inner products of two secret shared length vectors in $\mathbb{Z}_{2^\sigma}^n$ can be computed with $O(\sigma)$ bits of communication[MR18].

A.4 Arg Min/Max Protocol

Protocol Π_{ami} takes input as $[\![\mathbf{u}]\!]^{\mathbf{B}} \in \mathbb{Z}_{2^\sigma}^m$ and outputs $[\![\mathbf{b}]\!]^{\mathbf{B}} \in \{0,1\}^m$ such that $\mathbf{b}$ is a weight one vector where $u_i \in \mathbb{Z}_{2^\sigma}$ is a minimum element in $\mathbf{u}$ and $b_i = 1$. For simplicity let us assume $m = 2^d$ for some integer d. First the minimum element is computed in a binary tree fashion. The tree will have $2^d - 1$ internal nodes indexed by $1, 2, ..., 2^d - 1$ and leaf nodes index by $2^d, ..., 2^{d+1}$. Each leaf node will be assigned the share $[\![v_{2^d-1+i}]\!]^{\mathbf{B}} := [\![u_i]\!]^{\mathbf{B}}$ for $i \in [m]$. We will then define internal node shares as $[\![c_i]\!]^{\mathbf{B}} := \mathsf{LessThan}([\![v_{2i}]\!]^{\mathbf{B}}, [\![v_{2i+1}]\!]^{\mathbf{B}})$ and $[\![v_i]\!]^{\mathbf{B}} := [\![v_{2i}]\!]^{\mathbf{B}} \oplus [\![c_i]\!]^{\mathbf{B}}([\![v_{2i}]\!]^{\mathbf{B}} \oplus [\![v_{2i+1}]\!]^{\mathbf{B}})$. Note that this circuit results in $v_i = \mathsf{min}(v_{2i}, v_{2i+1})$. Then, for each $i \in [m]$, let $n_1, ..., n_d$ denote the node indices from the root to leaf $j = (2^d - 1 + i)$, i.e. $n_i = \lfloor j/2^{i-d} \rfloor$. b_j can then be computed as $[\![b_j]\!]^{\mathbf{B}} := \prod_{i=1}^{d-1} [\![c_{n_i}]\!]^{\mathbf{B}}$. To reduce the total number of multiplications all b_j can be computed together which results in a total of $O(m)$ overhead. Also note that the arg-max protocol Π_{amx} can be computed in a similar way by replacing $\mathsf{LessThan}(...)$ with $\mathsf{GreaterThan}(...)$ function.

B Random Forest Prediction

We describe two approaches on private evaluation of a single decision tree : i) Evaluation of Full Tree, ii) Evaluation of Active Path. The remainder of the section deals with extending our approaches and making them compatible to built our private random forest framework.

B.1 Dense: Full Tree Evaluation

Inspired from [DEF$^+$19] we propose a prediction protocol in our 3PC setting which requires us to evaluate the full tree to get the final predicted label. Prediction in DTs begin with mapping each feature in the query vector $\mathbf{z}$ to the corresponding node in the decision tree using $\mathcal{F}_{\mathsf{omap}}$. At a given decision node i, the mapped feature is compared to a threshold and a comparison bit $[\![c_i]\!]^{\mathbf{B}}$ is obtained. There is unique path from the root node to each leaf node, where each leaf node represents an output class label. For each leaf node there is a specific set of comparison values along this path which means this leaf hold the output label. As such, for each node the protocol computes if this node is "active" and if so we output that label. For an incomplete tree, we use the technique of adding dummy nodes as in [WFNL16, JPVE07] to have a complete one.

B.1.1 Input Sharing

In this phase, user $\mathsf{P_u}$ generates $[\![\mathbf{z}]\!]^{\mathbf{B}}$ of his features $\mathbf{z} \in \mathbb{Z}_{2^\sigma}^m$ and shares it among the servers. The model owner $\mathsf{P_m}$ shares the following among the servers: i) threshold vector $\mathbf{t} \in \mathbb{Z}_{2^\sigma}^n$, where n is the total number of decision nodes ($n = 2^d - 1$) and ii) vector $\ell \in \mathbb{Z}_{2^\sigma}^{n+1}$, where each ℓ_j represents the class of the j^{th} leaf node of the tree. We require $\mathbf{t}$ to be ordered such that values for the root node is stored at position 1. Then, for the i^{th} nodes, the left and right children must be stored at $2i, 2i+1$ respectively. iii) a mapping function $\theta : [m] \rightarrow [n]$, which maps $\mathsf{P_u}$'s query $\mathbf{z}$ to a vector (of decision nodes) $\mathbf{u}$ of size n. The role of the mapping function θ is to map the features to the nodes. In particular, for the i^{th} node which requires feature z_j, we define $\theta(i) = j$. Thus, after applying θ, the i^{th} node will consist of a threshold t_i and a mapped feature $b_i = z_{\theta(i)}$. The sharing of θ is done as mentioned earlier in the $\mathcal{F}_{\mathsf{omap}}$ section.

B.1.2 Evaluation

The decision tree can be visualized as an array with the first entry being the root node with index $i = 1$. A node at position $i \in [n]$ has a left and right child at $2i$ and $2i+1$. Evaluation is done as follows: First, the servers send $([\![\theta]\!], [\![\mathbf{z}]\!]^{\mathbf{B}})$ to $\mathcal{F}_{\mathsf{omap}}$ which returns $[\![\mathbf{b}]\!]^{\mathbf{B}}$ such that $b_i = z_{\theta(i)}$.

For each node $i \in [n]$, the parties evaluate the comparison circuit $[\![b_i]\!]^{\mathbf{B}} := \mathsf{cmp}([\![t_i]\!]^{\mathbf{B}}, [\![b_i]\!]^{\mathbf{B}})$. For each leaf node $i \in [n+1]$ observe that the root to leaf node indexes are $j_1, ..., j_d$ where $j_k =$

$\lfloor(2^d-1+i)/2^{k-d}\rfloor$. Moreover, this leaf node is the output node if $[\![a_i]\!]^{\mathbf{B}} := \prod_{k=1}^{d}([\![c_{j_k}]\!]^{\mathbf{B}} \oplus i_k)$ where i_k is the kth bit of i. The final output label can then be computed as $[\![y]\!]^{\mathbf{B}} := \bigoplus_{i\in[n+1]} [\![a_i]\!]^{\mathbf{B}}[\![\ell_i]\!]^{\mathbf{B}}$.

The computation of $[\![a_i]\!]^{\mathbf{B}}$ can be achieved in two ways. The first is to compute $[\![a_i]\!]^{\mathbf{B}} := \prod_{k=1}^{d}([\![c_{j_k}]\!]^{\mathbf{B}} \oplus i_k)$ as described above for each $i \in [n+1]$. This requires $O(nd)$ binary multiplications and can be performed in $O(\log d) = O(\log\log n)$ rounds. Alternatively, for $i \in [2, 2^{2d}]$ we can compute a bit $[\![a'_i]\!] := [\![a'_{i/2}]\!]^{\mathbf{B}}([\![c_{i/2}]\!]^{\mathbf{B}} \oplus \overline{i_1})$ where $i_1 := i \mod 2$. This bit encodes whether node i is on the “active” root to leaf path. Then we can define $[\![a_i]\!]^{\mathbf{B}} := [\![a'_{n+i}]\!]^{\mathbf{B}}$. This approach requires $O(d)$ rounds of communication with a total of $O(n)$ bits of communication. This second approach can be superior when the bandwidth is limited.

B.1.3 Complete Tree

One requirement with this technique is that the tree needs to be complete. This is due to the structure of the binary tree being considered private information as it is a function of the training data. In the case that the decision tree is not complete the model owner adds "dummy" nodes [WFNL16, JPVE07] until it is. This process of adding dummy nodes can in some cases can be quite expensive. Consider the pathological case where the tree has depth d and the number of real nodes is $O(d)$. Then the number of dummy nodes that need to be added is $O(2^d)$, which leads to an exponential overhead of evaluating such nodes.

Although this increase can be significant, in many cases adding dummy nodes does not substantially impact performance. Consider cases, where the trees in the random forest have depth d to be small. Alternatively, for some other applications it may be acceptable to omit some or all of these dummy nodes. This would leak the structure of the model but such information may not convey significant information.

B.2 Sparse: Active Path Evaluation

B.2.1 Input Sharing

In this phase, user $\mathsf{P_u}$ generates $[\![\mathbf{z}]\!]^{\mathbf{B}}$ of his features $\mathbf{z} \in \mathbb{Z}_{2^\sigma}^m$ and shares it among the servers. The model owner $\mathsf{P_m}$ describes the decision tree in a sparse data structure. Each decision node in the tree is defined as a tuple. The i^{th} decision node of the tree is represented as $N_i := (i, t_i, f_i, l_i, r_i, \ell_i, w_i)$, where t_i is the threshold corresponding to the i^{th} decision node, $f_i \in [n]$ indexes the feature to be compared with, $l_i \in [m]$ indexes the left child node and $r_i \in [m]$ indexes the right child node. That is, $z_{f_i} < t_i$ then the next node to be evaluated should be indexed by l_i and index r_i otherwise. If this is a leaf node then ℓ_i is the label and $w_i = 1, l_i = r_i = 0$. Otherwise $\ell_i = w_i = 0$. The bit w_i is used to denote that the current node is an leaf node.

For now let us assume that the tree is complete. $\mathsf{P_m}$ samples a permutation $\pi : [2n] \to [2n]$ such that $\pi(2i) = l_i, \pi(2i+1) = r_i$ for all i which are not leaf nodes. In addition, the model owner defines the mapping function $\theta : [m] \to [n]$ in the same way as the previous section.

$\mathsf{P_m}$ shares $[\![\mathbf{t}]\!]^{\mathbf{B}}, [\![\mathbf{w}]\!]^{\mathbf{B}}, [\![\ell]\!]^{\mathbf{B}}, [\![\pi]\!], [\![\theta]\!]$ among the servers. $[\![\pi]\!]$ is share according to Section A.2 and $[\![\theta]\!]$ according to Section A.3.

B.2.2 Construction

Our core approach is as follows. Each node i is assigned a random value a_i as its “random node index”. That is, the servers jointly sample $[\![\mathbf{a}]\!]^{\mathbf{B}} \leftarrow \mathbb{Z}_{2^\kappa}^{2m}$. Using the shared permutation $[\![\pi]\!]$ we map the child random index to the parent. That is, we will create two vectors $[\![\mathbf{L}]\!]^{\mathbf{B}}, [\![\mathbf{R}]\!]^{\mathbf{B}}$ such that for parent node i it holds that $L_i = a_{l_i}$ and $R_i = a_{r_i}$. These vectors are created by sending $([\![\pi]\!], [\![\mathbf{a}]\!]^{\mathbf{B}})$ to $\mathcal{F}_{\mathsf{perm}}$ and receiving back $[\![\alpha]\!]^{\mathbf{B}}$. The servers define $[\![L_i]\!]^{\mathbf{B}} := [\![\alpha_{2i}]\!]^{\mathbf{B}}, [\![R_i]\!]^{\mathbf{B}} := [\![\alpha_{2i+1}]\!]^{\mathbf{B}}$. Similar to the previous protocol the servers send map the features by sending $([\![\theta]\!], [\![\mathbf{z}]\!]^{\mathbf{B}})$ to $\mathcal{F}_{\mathsf{omap}}$ and receive back $[\![\mathbf{F}]\!]^{\mathbf{B}}$.

The parties define $[\![\mathbf{N}]\!]^{\mathbf{B}}$ such that $[\![N_i]\!]^{\mathbf{B}} = ([\![a_i]\!]^{\mathbf{B}}||[\![t_i]\!]^{\mathbf{B}}||[\![F_i]\!]^{\mathbf{B}}||\ [\![L_i]\!]^{\mathbf{B}}||[\![R_i]\!]^{\mathbf{B}}||[\![w_i]\!]^{\mathbf{B}})$. The servers then send $[\![\mathbf{N}]\!]^{\mathbf{B}}$ to $\mathcal{F}_{\mathsf{r-perm}}$ which randomly permutes the nodes & features and returns $[\![\mathbf{N}']\!]^{\mathbf{B}}$.

The servers then parse these back into the individual shares as $([\![a'_i]\!]^{\mathbf{B}}||[\![t'_i]\!]^{\mathbf{B}}||[\![F'_i]\!]^{\mathbf{B}}||[\![L'_i]\!]^{\mathbf{B}}||[\![R'_i]\!]^{\mathbf{B}}||[\![w'_i]\!]^{\mathbf{B}}) := [\![N'_i]\!]^{\mathbf{B}}$ and $([\![b'_i]\!]^{\mathbf{B}}||\ [\![z'_i]\!]^{\mathbf{B}}) := [\![Z'_i]\!]^{\mathbf{B}}$. Next the servers reveal $\mathbf{a}' := \mathsf{reveal}([\![\mathbf{a}']\!]^{\mathbf{B}})$ and $\mathbf{b}' := \mathsf{reveal}([\![\mathbf{b}']\!]^{\mathbf{B}})$ along with $a_1 := \mathsf{reveal}([\![a_1]\!]^{\mathbf{B}})$. The parties define i such that $a'_i = a_1$.

For d iterations the servers do the following. The servers compute $[\![c_i]\!]^{\mathbf{B}} := \mathsf{cmp}([\![t'_i]\!]^{\mathbf{B}}, [\![F'_i]\!]^{\mathbf{B}})$ and $[\![x_i]\!]^{\mathbf{B}} := [\![L'_i]\!]^{\mathbf{B}} \oplus [\![c_i]\!]^{\mathbf{B}}([\![L'_i]\!]^{\mathbf{B}} \oplus [\![R'_i]\!]^{\mathbf{B}})$. Note that if $c_i = 1$ the $x_i = R'_i$ and otherwise $x_i = L'_i$. The servers reveal $x_i := \mathsf{reveal}([\![x_i]\!]^{\mathbf{B}})$ and redefine $i := i'$ such that $a_{i'} = x_i$.

After d iterations the protocol outputs the label value $[\![\ell'_i]\!]^{\mathbf{B}}$ of the current node i.

B.2.3 Incomplete Tree

Now consider the case when the tree is not complete. The core challenge is to hide when we reach a leaf node i that is at level $j < d$. Our basic solution is to add a special path of d dummy nodes. Each dummy node δ_j sets both of its children nodes as the next dummy node δ_{j+1}. Similar, leaf nodes at level $j < d$ will set their children indices as δ_j. When a leaf node i is reached, it assigns its label ℓ_i to be the output label y. Then the child dummy node δ_j is evaluated which does not assign to y and will instruct its child dummy node δ_{j+1} to be evaluated next. This is repeated until a total of d nodes have been evaluated. The overhead of this modification is adding d dummy nodes and $O(d\sigma)$ AND gates to implement the required logic.

In more detail, at the start of the protocol the servers will augment the shared vectors $[\![\mathbf{t}]\!]^{\mathbf{B}}, [\![\mathbf{w}]\!]^{\mathbf{B}}, [\![\ell]\!]^{\mathbf{B}}$ to have length $m' := m + d$. These extra nodes shares are added to the end of each vector and the underlying values are defined to be zero. In addition, we require the model owner to share $[\![\pi]\!]$ such that $\pi : [2m'] \to [2m']$.

The servers the run same protocol as before up to evaluating the active path. Here we now do the following. They define $[\![y]\!]^{\mathbf{B}} := 0, [\![w']\!]^{\mathbf{B}} := 0$ and for the k^{th} iteration, the servers compute $[\![c_i]\!]^{\mathbf{B}}, [\![x_i]\!]^{\mathbf{B}}$ as before. Then they redefine $[\![x_i]\!]^{\mathbf{B}} := [\![x_i]\!]^{\mathbf{B}} \oplus [\![w']\!]^{\mathbf{B}}([\![x_i]\!]^{\mathbf{B}} \oplus [\![a_{m+k}]\!]^{\mathbf{B}})$ and then $[\![w']\!]^{\mathbf{B}} := [\![w']\!]^{\mathbf{B}} \oplus [\![w_i]\!]^{\mathbf{B}}$ and $[\![y]\!]^{\mathbf{B}} := [\![y]\!]^{\mathbf{B}} \oplus [\![\ell_i]\!]^{\mathbf{B}}$. After the d iterations the servers output $[\![y]\!]^{\mathbf{B}}$ as the label.

B.2.4 Optimizations

We present the protocol as is for clarity but several additional optimizations can be applied. The first allows us to improve the performance of the protocol Π_{perm} which realizes $\mathcal{F}_{\mathsf{perm}}$. Instead of sampling $[\![\mathbf{a}]\!]^{\mathbf{B}}$, we allow one server, say P_j, to know it. In this case a more efficient permutation protocol can be know. However, it is then important that whenever an a_i, a'_i, x'_i value is revealed, that the party who samples $\mathbf{a}$ does not see the revealed value as this would reveal the permutation. It is possible to hide these values from P_j due to the other two server possessing enough information to define the relevant shares. In addition, this P_j can now define $\mathbf{a}$ as a random permutation of $[m']$ which allows each a_i value to be represented with $\log_2(m')$ bits as opposed to κ.

In addition, it is safe to assume that the first three nodes of the tree are not leaf nodes. As such they need not be permuted. More generally, one can consider a hybrid approach where the first d' levels of the tree are padded to be complete and evaluated with the first approach. At level $d' + 1$ the protocol switching to the second approach.

Observe that the protocol $\Pi_{\mathsf{ip-map}}$ realizing $\mathcal{F}_{\mathsf{omap}}$ requires $O(nm)$ communication to share a tree among the servers. In addition, the computational overhead of evaluating $\Pi_{\mathsf{ip-map}}$ is $O(nm\sigma)$ where σ is the length of the strings being mapped. For many practical settings this quadratic overhead in n, m is not too significant given the small constants. However, we address this by providing an alternative construction for the sparse protocol which achieves $O(n + m)$ complexity.

First consider the case where each feature is used at most once along any root to leaf path of the tree. Building on this assumption we present a new approach for mapping features to node. The servers sample a PRF key $[\![k]\!]$ within the MPC protocol and compute $[\![\mathbf{F}]\!]^{\mathbf{B}}$ where $[\![F_i]\!]^{\mathbf{B}} := \mathcal{F}_{[\![k]\!]}([\![f_i]\!]^{\mathbf{B}})$ where f_i is the feature *index* that should be used with node i and $[\![\mathbf{f}]\!]^{\mathbf{B}}$ is provided by the model owner. In addition, the vector $[\![\mathbf{b}]\!]^{\mathbf{B}}$ is computed as $[\![b_i]\!]^{\mathbf{B}} := \mathcal{F}_{[\![k]\!]}([\![i]\!]^{\mathbf{B}})$.

When shuffling the nodes $[\![N]\!]^{\mathbf{B}}$, it is now the case that F'_i is the "pseudorandom feature index" instead of the feature itself. The servers then additionally shuffle $[\![\mathbf{Z}]\!]^{\mathbf{B}}$ via $\mathcal{F}_{\mathsf{r-perm}}$ where $[\![Z_i]\!]^{\mathbf{B}} := ([\![b_i]\!], [\![z_i]\!]^{\mathbf{B}})$ to obtain $[\![\mathbf{Z}']\!]^{\mathbf{B}}$ such that $([\![b'_i]\!]^{\mathbf{B}}, [\![z'_i]\!]^{\mathbf{B}}) := [\![Z'_i]\!]^{\mathbf{B}}$. The servers then compute $\mathbf{b}' := \mathsf{reveal}([\![\mathbf{b}']\!]^{\mathbf{B}})$. When evaluating the active path the servers first need to obtain the actual feature value. This is achieved by revealing the pseudorandom feature index of the current node. This in turn allows the servers to identify the which shuffled feature $[\![z'_j]\!]^{\mathbf{B}}$ should be use to compute the comparison bit $[\![c_i]\!]^{\mathbf{B}}$.

This protocol is straight forward to prove secure. Each f'_i value that is revealed will be unique and has not been previously revealed due to the assumption each feature is used once. Then, given that the feature vector has also been permuted no information about which feature this corresponds to is leaked to the servers.

When $\mathbf{F}$ is implemented as the LowMC circuit[ARS+15], it has been shown[MRR19] that 1 million PRF evaluation can be performed per second in the [MR18] framework. In particular, each evaluation requires computing approximately 500 AND gates in the [MR18] framework. Compared to the $\Pi_{\mathsf{ip-map}}$ approach, this alternative construction increases the communication complexity by a constant of approximately 500 while making the computational overhead $O(n+m)$ as opposed in $O(nm)$.

We have two approaches to lift the restriction that a feature can be used at most once. The first is to make all possible threshold values t_i public and have the user compute all possible feature-threshold predicates $c_{i,j} := \mathsf{cmp}(z_i, t_j)$ and input this as their feature vector. This works well when the set of thresholds can be made small. Alternatively, each node can contain a special (secret shared) value which encodes that this node should use the feature from a previous level. The servers can then use this to obliviously reuse the a previous feature. This latter approach results in a added overhead of $O(d^2)$ which is acceptable since d is typically ≈ 60 in practice.

C Edge Case Scenarios

We propose a heuristic approach of dealing with scenarios where the users' data distributions are drastically different from one another. Recall from our above example that all the trees in the random forest may not be compatible to correctly predict a given query. Thus our idea revolves around choosing only a subset of trees which show compatibility with the given query. This newly constructed subset of trees are then used during the prediction phase. The construction of the aforementioned subset, given a new query $\mathbf{q}$ is as follows:

i) *Constructing a Similarity vector:*
We compute a similarity vector $\mathbf{u}$ by comparing $\mathbf{q}$ with training samples in $\mathcal{D}$, based on a pre-agreed similarity measure. Most often the similarity measure used is a simple Euclidean Distance (ED) measure, but any other measure can also be used given the type of dataset. Let the length of $\mathbf{u}$ be m, where m is the total training samples in $\mathcal{D}$.

ii) *2-Opinion Search*:
Given the similarity vector $\mathbf{u}$, we search the two most similar training samples to $\mathbf{q}$. The intuition is training samples similar to $\mathbf{q}$ will give an opinion in which region of the output class, vector $\mathbf{q}$ belongs to. The 2-opinion search boils down to finding the indices of the two smallest elements in $\mathbf{u}$ (if ED is used as a measure). A straight-forward way of computing the indices of the two smallest elements would be to invoke the Π_{ami} protocol once, find the index of the smallest element, set the min value in $\mathbf{u}$ to max possible value and re-invoke Π_{ami} protocol to find the index of the second smallest element. The communication and round complexity of this method would be $\approx 3m\ell$ bits and $\approx 2\log(m)$ rounds respectively. Instead, we propose an alternate approach with $(\frac{2}{3})^{\text{rd}}$ the above communication and with half the number of rounds. We modify our Π_{ami} protocol to simultaneously output both the first and second smallest element in an array. Both the smallest and the second smallest element are computed in a binary tree fashion. For simplicity let us assume $m = 2^d$ for some integer d.

The first tree to compute the smallest element and its corresponding index follows the same procedure as Π_{ami} protocol, where the internal node shares are given as $[\![c_i]\!]^{\mathbf{B}} := \mathsf{LessThan}([\![v_{2i}]\!]^{\mathbf{A}}, [\![v_{2i+1}]\!]^{\mathbf{A}})$ and $[\![v_i]\!]^{\mathbf{A}} := [\![v_{2i}]\!]^{\mathbf{A}} \oplus [\![c_i]\!]^{\mathbf{B}}([\![v_{2i}]\!]^{\mathbf{A}} + [\![v_{2i+1}]\!]^{\mathbf{A}})$.
The tree for the second smallest element, also has a similar structure as the previous tree. Each leaf node is assigned the share $[\![w_{2^d-1+i}]\!]^{\mathbf{A}} := [\![u_i]\!]^{\mathbf{A}}$ for $i \in [m]$.
At the penultimate level $(d-1)$, for $i \in [m/2]$ and $j := [2^{d-1} + i - 1]$, w_j is assigned the greater of the two values w_{2j} and w_{2j+1}.
At every other level (arbitrary depth k), for $i \in [m/2^{d-k}]$ and $j \in [2^k + i - 1]$, w_j is updated by first removing the largest and the smallest element from the set $\{v_{2j},\ v_{2j} + 1,\ w_{2j},\ w_{2j} + 1\}$, followed by selecting the smaller out of the two remaining elements.
The index associated with the second smallest element is also computed in a similar fashion, where the output $\mathbf{b}'$ is a bit vector with the corresponding position of the element, set to 1.
Note that to accommodate cases where the largest and the second largest value in $\mathbf{u}$ corresponds to the two most similar samples, we replace $\mathsf{LessThan}(...)$ with $\mathsf{GreaterThan}(...)$ function to make our protocol compatible.

iii) *Decision Tree Compatibility:*
Given the two most similar training samples to $\mathbf{q}$, for each decision tree in the random forest ($\mathrm{dt}_i \in \mathcal{M}$), we check if dt_i is able to correctly classify any one of the two samples. Observe that the prediction on the training samples are needed to be computed only once before the prediction phase begins and then can be reused for all new incoming prediction queries. Thus the computation boils down to comparing if the prediction (a one-hot vector) for the two training samples matches with their corresponding true-labels. The output of the result (1 if a match else 0) is then multiplied to the prediction of dt_i on $\mathbf{q}$, which decides if dt_i in the random forest is eligible to take part in the final voting process or not.

Later in section D.1 we experimentally simulate non-identical behavior among participating users and show our initially proposed collaborative approach works for most of the real world scenarios and the remaining edge/pathological cases is effectively handled by our above modified approach.

D Experimental Results

D.1 Edgecase Scenarios

We simulate edge scenarios by distributing the training data amongst the participating users using Dirichlet distribution on the class labels [HHM19]. On a high level, the number of samples with respect to each class of the training data is non-uniformly distributed among the users. It is also possible for a subset of users to not have even a single sample of a given class in their locally generated dataset. Concretely, to generate a population of non-identical users, we sample $q \sim Dir(\alpha p)$ from a Dirichlet distribution, where p characterizes a prior class distribution over all distinct classes, and $\alpha > 0$ is a concentration parameter which controls the amount of identicalness among the users. Figure 1 illustrates the samples drawn from the Dirichlet distribution for different values of α, where as $\alpha \to \infty$, all users have identical distributions, whereas as $\alpha \to 0$, on the other extreme, each user holds samples of only one class chosen at random. As observed in Figure 1, a valid argument can be made that for most of the real world scenarios the underlying data distribution of the participating users would lie somewhere in the range of $\alpha \approx 10 - 100$.

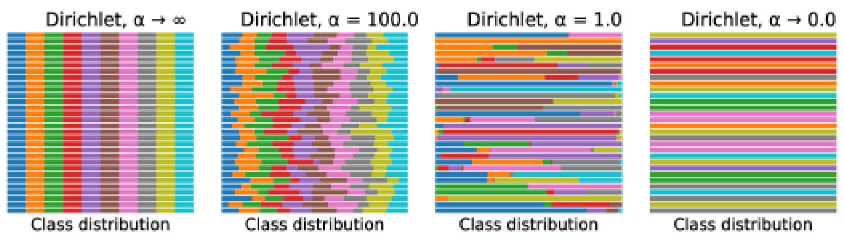

Figure 1: Local dataset generated for each user from Dirichlet distribution for varying α values. Each row represents a user and each color in the row represents a class label.

We test our approach on multiple datasets by varying the parameter α to generate varying degree of non-identicalness among the users. Figure 2 gives a comprehensive view of how the test accuracy of our approach varies for different values of α.The training data is divided among 20 users based on the value of α. Euclidean distance is used as the pre-agreed similarity measure to compute our similarity vector $\mathbf{u}$. Additionally, we use our sparse protocol to compute the prediction of a decision tree on a new query, as the depth of the decision trees for the above datasets lie in the range $d \approxeq 60 - 80$.

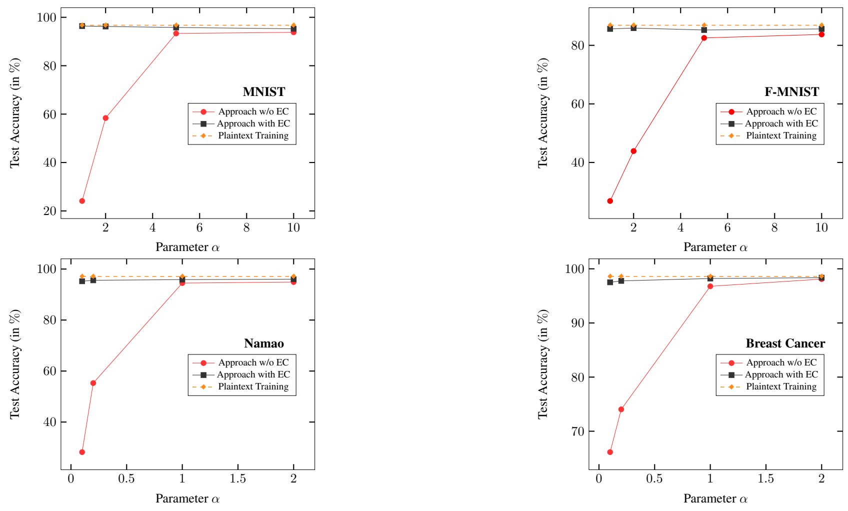

Figure 2: Test Accuracy of our collaborative approach with and without handling Edge Case (EC) scenarios over multiple datasets for varying concentration parameter α

We observe that our original collaborative approach (without EC) has a test accuracy comparable to Plaintext training for $\alpha > 5$ for MNIST and F-MNIST datasets. Similarly, for Namao and Breast Cancer datasets we observe a similar behavior for $\alpha > 1$. Thus, the above experiments further strengthens our earlier argument that our collaborative approach without EC works quite well for most of the real world scenarios. On the other hand, our modified approach is also able to effectively handle edge cases like $\alpha < 1$ in Namao and Breast Cancer dataset. The only drawback of the modified approach is the overhead to compute the three additional steps, in order to create a new subset of trees that are compatible with respect to a given query. The asymptotic communication complexity to compute the aforementioned steps is dependent on the size of the dataset. Concretely, for the given datasets, we observe that the communication overhead is $1.98 - 2.76\times$ more when we use our modified approach instead of our original collaborative approach. Amongst the three aforementioned steps, the second step of finding the two most similar training samples for a given query, consumes $\approx 40\% - 60\%$ of the additional communication overhead. Similarly, the asymptotic round complexity is also in the order of $\log$ of the dataset size. Concretely, across the given datasets, $\approx 94 - 100$ rounds are required to complete those three steps. Note that, these rounds can be clubbed with the rounds required by our sparse protocol to compute the active path of the tree as both the computations are independent.

Thus, our modified approach is able to handle all values of α including the edge cases but at the cost of a reasonable overhead, whereas our earlier proposed collaborative approach is able to efficiently handle a large range of α values, where most of the real world use cases would lie. Thus, both approaches have their own pros and cons and can be viewed as a design choice which the participating users can make at the beginning of the training phase.

D.2 Prediction Protocol

Protocol	n	d	m	Time	Total Comm	Comm per Tree
Sparse	100	5	32	37	1,780,323	4,450
		10	200	95	10,796,329	26,991
			1000	154	44,836,329	112,091
		40	200	122	11,482,669	28,707
			1000	159	45,522,669	113,807
			4000	326	173,172,669	432,932
		80	200	148	12,397,789	30,994
			1000	185	46,437,789	116,094
			4000	343	174,087,776	435,219
	1000	5	32	55	5,661,939	14,154
		10	200	130	28,076,316	70,191
			1000	215	62,116,329	155,291
		40	200	148	28,762,669	71,907
			1000	230	62,802,669	157,007
			4000	514	190,452,669	476,132
		80	200	181	29,677,789	74,194
			1000	258	63,717,789	159,294
			4000	563	191,367,789	478,419
dense	100	5	32	7	203,553	509
		10	1024	52	1,691,613	4,229
		12	4096	138	6,299,637	15,749
	1000	5	32	7	203,553	509
		10	1024	89	1,691,613	4,229
		12	4096	324	6,299,637	15,749

Table 2: Running time (ms) and communication (bytes) overhead of our approaches for various parameters. 400 trees were evaluated. f denotes the number of features, each of size 1 bit. d denotes the depth of each tree. m denotes the number of trees. There are 8 output labels.