Extremely Randomized Trees with Multiparty Computation

by

David Melanson

Supervised by Dr. Martine De Cock

A senior thesis submitted in partial fulfillment of the departmental honors requirements for the degree of

Bachelor of Science
Computer Science & Systems
University of Washington Tacoma

June 2020

Presentation of work given on June 5, 2020

The student has satisfactorily completed the Senior Thesis, presentation and senior elective course requirements for CSS Departmental Honors.

Faculty advisor: [illegible] Date JUNE 12, 2020

CSS Program Chair: [illegible] Date June 12, 2020

Extremely Randomized Trees with Multiparty Computation

David Melanson

An honors thesis
submitted in partial fulfillment of the
requirements for

Computer Science and Systems Honors

University of Washington Tacoma

2020

Reading Committee:

Dr. Martine De Cock, Chair

Program Authorized to Offer Degree:
B.S. in Computer Science and Systems

University of Washington Tacoma

Abstract

Extremely Randomized Trees with
Multiparty Computation

David Melanson

Chair of the Supervisory Committee:
Dr. Martine De Cock
School of Engineering and Technology

Machine learning (ML) is a prominent field in the study of computer science. The usefulness of ML algorithms comes from its ability to process large amounts of data and recognize patterns in the data, which allows to classify new, previously unseen instances of data. A limiting factor in ML is sometimes data accessibility. In the modern age, there often exists more data than we would know what to do with, but some data must be kept private, and thus some databanks become inaccessible. In this thesis, we use Privacy-Preserving Machine Learning (PPML) with Multi-Party Computation (MPC) to allow two parties to jointly train an ML model on their combined data without having to reveal their data to each other. The ML algorithm we've chosen to implement into a secure MPC framework is the Extremely Randomized Trees (extra trees) algorithm. The extra trees algorithm outputs an ensembles of decision trees, which are state-of-the-art for many supervised ML tasks. Extra trees classifiers can be trained on data with continuous attribute values without the need to sort the data which would be an expensive operation in the context of MPC. Our MPC protocol for training extra trees classifiers is fast enough to use in practice, and could, for example, allow two representatives of hospitals to combine their data regarding symptoms in patients to train an ML model for medical diagnosis, all in a privacy-preserving manner.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

The author wishes to express sincere appreciation to Dr. Martine De Cock who has guided me through this entire process, allowing me to succeed in my academic endeavours. I also wish to thank Davis Railsback and Dr. Anderson C.A. Nascimento who have patiently taught me much of what I know about MPC. Additionally, I extend my thanks to Rafael Dowsley, who assisted me in writing pseudo-code in a concise, and clear manner. I would also like to thank Jianwei Shen, and Davis Railsback again, for helping me understand how to implement the secure extra trees algorithm in Rust. Lastly, I would like to thank Samuel Adams, who tirelessly implemented the secure extra trees source code with me, splitting much of the workload in half.

Chapter 1
INTRODUCTION

Machine learning (ML) has enabled important and impactful applications in data science over the past few decades. With the vast increase in data availability, and the increased capabilities of modern hardware, we've been able to process large amounts of data and make meaningful predictions from that data. ML has a wide variety of applications, from determining who would be most susceptible to an advertisement, to assisting doctors in uncovering if a patient has cancer or not. With its wide range of applications, ML has become a valuable asset.

In principle, the more data you have available, the more reliable you can make your ML model. The issue that we sometimes come across in ML is that we do not have enough data at our disposal to train our models. In some cases, the data may exist in massive amounts, but we may not have access to it because it is private. Let us consider the case where we have two parties in the medical field representing their hospitals, Alice and Bob. Alice and Bob both have data regarding features of tumors in patients. These tumors can be classified as malignant, or benign. The two parties can use their own data to train a ML model that can help doctors diagnose patients in their respective hospitals. While each party may have enough data to make meaningful ML models, it could be beneficial if the parties somehow combined their data in order to make a more sophisticated model. However, this can endanger the parties of violating patient confidentiality. The data hospitals keep on their patients tend to be highly private, and illegal to distribute. These sort of dilemmas can make it difficult to provide ML models with the best possible accuracy. Considering how valuable some of these ML tasks can be, we want to find a way to circumvent these issues.

Scenarios akin to the issue in the previous paragraph are what we aim to tackle in this thesis. We present a Privacy-Preserving Machine Learning (PPML) algorithm that combines ML with cryptographic techniques to conceal data. Our PPML algorithm is constructed with Secure Multi-Party Computation (MPC) protocols [4]. This will allow multiple parties to aggregate their data to train a common ML model without ever revealing information.

While deep learning is state-of-the-art for tasks that relate to perception, such as computer vision and natural language processing, in domains with structured information, the best results are often obtained with tree ensemble methods, such as random forests and boosted decision trees [7]. The latter also have the advantages of being faster to train and being easier to interpret. The PPML training algorithm we introduce in this thesis is modeled after the extremely randomized trees (extra trees) classifier presented by Geurts et. al [9]. Extra trees process continuous valued data and produce an ensemble of decision trees to classify said data. Each decision tree (DT) is trained by picking a random subset of attributes in the dataset, and then partitioning the dataset based on random splits in those attributes, and selecting the best split. This algorithm is a suitable choice for PPML because by choosing splits randomly, we avoid the need to sort the data in order to find the optimal split, and sorting data is an expensive operation in the context of MPC. Currently, there appear to exist few proposals for training DTs on continuous attributes [2, 12, 13]. These proposals are based off of Quinlan's C4.5 algorithm [11], which does require sorting the data. To the best of our knowledge, the privacy-preserving extra trees algorithm presented in this thesis, which we will refer to as π_{XT} for its remainder, is the first and only of its kind. Our π_{XT} algorithm provides the accuracy of extra trees, and the security guaranteed by MPC protocols. In this thesis, we focus on scenarios with two parties Alice and Bob (2PC), where the values of the input attributes are continuous, and whose target values are discrete. This scenario for instance arises when Alice and Bob both have tissue samples, characterized through gene expression values, that need to be classified as cancer or not. For other real world datasets that fit this scenario, we refer to Chapter 6.

The rest of this thesis is structured as follows: In Chapter 2, we will describe the extra trees algorithm from Geurts et. al [9] which acts as the precursor to our own. In Chapter

3, we recall cryptographic preliminaries that are necessary to understand how the secure ML algorithm functions. This includes the additive sharing scheme we use to keep each party's data hidden from the other, and cryptographic primitives we use to construct certain functionality. Chapter 4 describes a *pre-processing* phase, which consists of descriptions about how we format our data, and the operations we perform on the data before we enter the training algorithm. In Chapter 5 we then discuss how we actually train our model with the pre-processed data. In Chapter 6, we examine the runtime and accuracy results of our algorithm on real world data. Finally, in Chapter 7 we provide a summary of the thesis, and future work.

The results from this work, along with proofs of correctness and security of the protocols that we developed, have been submitted for publication as part of a paper:

Secure Training of Tree based Models over Continuous Data
Samuel Adams, Chaitali Choudhary, Martine De Cock, Rafael Dowsley, **David Melanson**, Anderson Nascimento, Davis Railsback, and Jianwei Shen
Under review, 2020

Chapter 2

EXTRA TREES IN THE CLEAR

2.1 Extra Trees Algorithm

A proposal and implementation of the extra trees algorithm was given by Pierre Geurts, Damien Ernst, and Louis Wehenkel in their paper *Extremely randomized trees*, published in 2006 [9]. The algorithm gets its name from the random way it constructs each DT, which reduces the chance of overfitting the data. Although the algorithm could be adapted to process categorical attributes, the main focus of the paper was to process numerically valued attributes. Since its introduction, the exra trees algorithm has become popular among data scientists. It is for instance included in well known ML software libraries such as *sklearn*[1] and frequently adopted in successful solutions in data science competition.

In PPML, we say that an algorithm is "in-the-clear" if we make no attempt to hide the data, in other words, when all computations are done on plaintext instead of on encrypted data. The original extra trees algorithm is an example of an in-the-clear algorithm. The extra trees algorithm works by training an ensemble of binary DTs, where each DT in the ensemble has its own idea of how to classify new data.

2.1.1 Training an Extra Tree

Apart of what makes extra trees random is the way it processes the data set. Throughout the training of DTs, the algorithm will randomly pick several columns of data without replacement, and try to determine which column best describes the data. Each column corresponds to an attribute which describes a feature of the data set. Let us take the

[1] `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html`

following data set as an example. Suppose we have 9 training examples (instances) for a predictive maintenance system. The input attributes $A_1, ..., A_4$ are reading from 4 sensors, and the classification (label) indicates whether the machine needs maintenance: "b" (blue) means "everything is ok", "r" (red) means "needs urgent maintenance", and "y" (yellow) means "may need maintenance in the near future".

A_1	A_2	A_3	A_4	label
1.5	1.2	1.0	1.0	b
1.7	1.5	2.2	1.4	b
2.1	1.2	3.0	1.6	b
2.6	2.2	1.4	2.0	r
2.7	2.7	2.0	2.4	r
4.8	3.9	3.4	2.6	y
2.2	1.0	1.2	3.0	b
2.4	2.6	2.4	3.4	b
3.8	4.1	3.6	3.6	y

Let us suppose the algorithm randomly chooses 2 columns, A_1 and A_3, from the data set. The algorithm will then pick a random threshold between the maximum and minimum values in each column. Lets say 2.7 for A_1, and 2.35 for A_3. Based off of the threshold chosen for A_1, we can horizontally partition the data set as follows:

A_1	A_2	A_3	A_4	label
2.7	2.7	2.0	2.4	r
4.8	3.9	3.4	2.6	y
3.8	4.1	3.6	3.6	y

A_1	A_2	A_3	A_4	label
1.5	1.2	1.0	1.0	b
1.7	1.5	2.2	1.4	b
2.1	1.2	3.0	1.6	b
2.6	2.2	1.4	2.0	r
2.2	1.0	1.2	3.0	b
2.4	2.6	2.4	3.4	b

Notice that the first subset only consists of data where the attribute A_1 has values ≥ 2.7, and the second subset has values < 2.7. Doing the same thing for the split in A_3 yields:

A_1	A_2	A_3	A_4	label
2.1	1.2	3.0	1.6	b
4.8	3.9	3.4	2.6	y
2.4	2.6	2.4	3.4	b
3.8	4.1	3.6	3.6	y

A_1	A_2	A_3	A_4	label
1.5	1.2	1.0	1.0	b
1.7	1.5	2.2	1.4	b
2.6	2.2	1.4	2.0	r
3.1	2.7	2.0	2.4	r
2.2	1.0	1.2	3.0	b

Now we have two subsets of the data for each split column, A_1 and A_3. We now need a quantifiable way to determine which attributes split best describes the data. The metric used in the original extra trees algorithm is a variant of *Shannon information gain*, which scores how meaningful the split in our data is. In π_{XT}, we use a variant of a different scoring method, the *Gini index*. For consistency's sake, we will use the Gini index to score the data sets in this example. The Gini index is a number between 0 and 1 that describes the probability that we incorrectly classify the data if we randomly pick a classification relative to the distribution of the current classifications in the data. Given this definition, we naturally wish to find the split that minimizes the Gini index. We define the Gini index for a particular split as $G(A_j) = 1 - \sum_{i=1}^{2} \left(\frac{|S_i|}{|S|} (\sum_{j=1}^{C} (p_{i,j})^2) \right)$ where S is the current data set, S_i is one of the two data sets created by the split, C is the number of classifications, and $p_{i,j}$ is the proportion of the j'th classification in the i'th subset of data.

The first subset created by the split in A_1 had 3 training examples, with a proportion of 1/3 red, and 2/3 yellow. The second data set has 6 training examples, with a proportion of 5/6 blue, and 1/6 red. With this information, we can calculate its Gini index as

$$G(A_1) = 1 - \left(\frac{3}{9} \left((\frac{1}{3})^2 + (\frac{2}{3})^2 \right) + \frac{6}{9} \left((\frac{5}{6})^2 + (\frac{1}{6})^2 \right) \right) \approx 0.33.$$

Performing similar calculations for the split from A_3 gives

$$G(A_3) = 1 - \left(\frac{4}{9} \left((\frac{2}{4})^2 + (\frac{2}{4})^2 \right) + \frac{5}{9} \left((\frac{3}{5})^2 + (\frac{2}{5})^2 \right) \right) \approx 0.49.$$

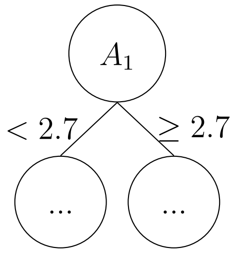
Because the split in A_1 has a Gini index smaller than A_3, the extra trees algorithm will choose to split on the threshold 2.7 from A_1. We will then take the subsets created from the split with the columns A_1 and A_3 removed, and create a right child trained on the set

A_2	A_4	label
2.7	2.4	r
3.9	2.6	y
4.1	3.6	y

and a left child trained on the set

A_2	A_4	label
1.2	1.0	b
1.5	1.4	b
1.2	1.6	b
2.2	2.0	r
1.0	3.0	b
2.6	3.4	b

To classify an unseen instance on the DT we have started to create, we would first test the value in attribute A_1 to see whether it is less than the threshold, or greater than or equal to the threshold. The traversal of the instance on this DT is dependant on the values of the instances attributes. The DT we have trained up to this point looks like the following:



The tree is then recursively built by the same rules until we hit one of the following 3 stopping conditions: (1) all of the class labels in the nodes' subset of data are the same, (2) the amount of data (rows) that has reached the node is under a certain threshold, or (3) each column of data only has one distinct value remaining. When a stopping condition is met, the frequencies of each class label in the data set are returned, i.e. probability distribution over the class labels. This allows us to weight the vote of each tree in the ensemble. To train m trees, we repeat the same process m times. High level pseudo code to train one DT is given in Algorithm 1.

Algorithm 1: Algorithm for Training an Extra-Trees Classifier.

```
Input  : A training set $S$ with continuous data and $n$ samples (each sample has $f$
         attributes (features)), the number $k$ of features to consider in each node,
         the number $m$ of trees in the ensemble, the minimum number $n_\epsilon$ of values
         to split on.
Output: An ensemble of trees $E = t_1, ..., t_m$
1  Function Build_extra_tree($S, k, m, n_\epsilon$)
2      if $|S| \leq n_\epsilon$ or all classes or attribute values in $S$ are constant then
3          return class frequencies
4      else
5          Randomly select $k$ (non-constant valued) attributes, $\{a_1, ..., a_k\}$, without
            replacement, among all candidate attributes in $S$.
6          Generate $k$ splits, $\{s_1, ..., s_k\}$, where each split $s_i$ is a value in the range of
            the attribute $k_i$ in $S$
7          Select $s'$ such that $Score(s', S) = max_{i \in \{1,...,k\}} Score(s_i, S)$, where the scoring
            function outputs a number indicating how good the split is
8          From $s'$, split $S$ into two sets, $S_l$ and $S_r$
9          Let $t_l = Build\_extra\_tree(S_l)$ and $t_r = Build\_extra\_tree(S_r)$
10         Create a node $N$ with the split $s'$, attach $t_l$ and $t_r$ as left and right subtrees
11     end
12     return $N$
```

2.2 Variations to Extra Trees

The extra trees algorithm is designed to maximize accuracy by making an ensemble of DTs that are resilient to noisy data. There are a few rules in the algorithm that do not lend themselves to a MPC setting. For example, removing attributes without replacement. This operation would require us to maintain information about what attributes are available to us at any given moment. While this should be no issue in-the-clear, keeping these results in a secure setting, and then using them, could add significant computational expenses. In general, every aspect or rule in the extra trees algorithm whose removal does not make a large impact on the accuracy should be removed. We'll see in Chapter 6 that any loss in accuracy due to the removal of a rule should be regainable through tweaking the parameters. In chapters 4 and 5 we will construct a nearly equivalent algorithm to extra trees that will be ideal in the MPC setting.

Chapter 3
CRYPTOGRAPHIC PRELIMINARIES

3.1 Fixed Point Representation in a Ring

The protocol π_{XT} that we propose in this thesis is designed for training of extra trees classifiers on data with continuous attribute values (real numbers). The training data consists of a set S of training examples $\langle(x_1, x_2, \ldots, x_f), y\rangle$. The attribute (or feature) values $x_1, x_2, \ldots, x_f$ are real numbers, while the class label y is categorical. The goal is to learn a function from the data that maps previously unseen feature values to a corresponding class label, e.g. to determine whether a patient has a disease or not based on blood pressure, temperature etc. The kind of functions that we consider in this thesis are ensembles of DTs. Our assumption is that, instead of residing in one place, the data set S is distributed across multiple data owners.

During the execution of the protocol π_{XT} and its subprotocols, operations are performed on additive shares in the ring $\mathbb{Z}_q$, where q is some appropriate integer. In this thesis, we mainly use $q = 2^\lambda$. Before execution of any protocol, data owners must map any continuous attribute value x into the ring. To do so, they use the function $Q : \mathbb{R} \to \mathbb{Z}_{2^\lambda}$, where Q is defined as (see [3])

$$Q(x) = \begin{cases} 2^\lambda - \lfloor 2^a \cdot |x| \rfloor & \text{if } x < 0 \\ \lfloor 2^a \cdot x \rfloor & \text{if } x \geq 0 \end{cases} \tag{3.1}$$

When converting $Q(x)$ into its bit representation, it consists of λ bits in total. The first a bits represent the fractional part of x, the next b bits represent the non-negative integer part of x, and the most significant bit represents the sign in a two's compliment scheme. For finite λ, Q defines a bijection between the reals that can be represented by λ bits, and the integers in $\mathbb{Z}_q$. This allows us to easily combine shares and transition from the ring to the reals if the parties wish to reveal the actual values at any point. When choosing a λ value, we must choose one large enough to be able to represent the largest numbers produced during protocols. Multiplication of fixed point numbers doubles the fractional bits, and so products must be truncated to remain in the proper range. The product can also double the non-negative integer bits (which do not become truncated), and as such, λ must be at least $2(a + b)$. It is also important to choose a b large enough to represent the largest value possible for the integer part of all x's. The value of λ, a, and b largely depend on the type of data we are processing. In our work, $\lambda = 64$, $b = 20$, and $a = 10$ suffices.

3.2 Trusted Initializer

For π_{XT}, we use a trusted initializer (TI) which pre-distributes correlated randomness to the parties participating in the protocol. This correlated randomness allows us to apply a one time pad (OTP) to each piece of data. This process of using OTPs on each piece of data allow us to keep data secure. Among other things, the TI also distributes *multiplication triples*, which allows parties to securely multiply their secure values together [1]. After the TI makes its distributions, it terminates. This ensures that it has no access to any of the protocols. If the TI is unavailable or undesirable, the two parties can simulate the process of the TI at the cost of efficiency during the distribution phase.

3.3 Secure Multiparty Computation

3.3.1 Security Setting

When constructing MPC protocols, we need to define the type of adversaries we intend to protect data-holders from. In this thesis, we exclusively consider "honest-but-curious adversaries". This setting is typical for MPC based PPML (see e.g. [5, 6]). This assumes that the parties involved will not deviate from the protocols, but will try to learn information regarding the other parties' data during execution. The protocols we implement prevent parties from learning this information.

3.3.2 Shares

To achieve security of parties' data in 2PC, we use an *additive sharing* scheme. Additive sharing applies a OTP of correlated randomness to each piece of data. After each party maps their values into $\mathbb{Z}_q$, they secret share attribute values on their end and send it to the other party. Any value z in $\mathbb{Z}_q$ is split into two shares $z_0, z_1 \in \mathbb{Z}_q$ uniformly at random subject to the constraint $z = z_0 + z_1 \mod q$. From the perspective of the parties, the secret shared data looks like random noise. It is only when parties add their shares together do they retain their true value in the ring. We denote each party's secret sharing of a value z as $[\![z]\!]_q$. All computations in our work are assumed to be modulo q unless otherwise specified, and so we omit modular notation with q from the remainder of this thesis, and will refer to $[\![z]\!]_q$ as $[\![z]\!]$.

The following operations can be performed locally (no communication required) with secret shared values between the parties Alice and Bob in 2PC:

- Addition ($z = x + y$): Alice and Bob just add their local shares of x and y. This operation will be denoted by $[\![z]\!] \leftarrow [\![x]\!] + [\![y]\!]$.

- Subtraction ($z = x - y$): Alice and Bob subtract their local shares of y from that of x. This operation will be denoted by $[\![z]\!] \leftarrow [\![x]\!] - [\![y]\!]$.

- Multiplication by a constant ($z = c \cdot x$): Alice and Bob multiply their local shares of x by c. This operation will be denoted by $[\![z]\!] \leftarrow c[\![x]\!]$

- Addition of a constant ($z = x + c$): Alice adds c to her share x, while Bob keeps the same share of x. This operation will be denoted by $[\![z]\!] \leftarrow [\![x]\!] + c$.

The way we decide which party adds c to their share can be done arbitrarily. For the sake of consistency, we use an *asymmetric bit* which indicates which party does one operation, while the other party omits the operation. The parties asymmetric bit is agreed upon by the parties before any computations occur.

3.3.3 Secure Multiplication and Multiplication Triples

A non-trivial operation is secure multiplication of secret shared numbers. There are occasions in π_{XT} where we securely perform a dot product, or square a shared value. Among a few others, these operations require Alice and Bob to compute secret shares of products $[\![x \cdot y]\!]$, starting from secret shares of $[\![x]\!]$ and $[\![y]\!]$. To this end, Alice and Bob use multiplication triples, consisting of three secret shared values provided by the TI, a, b and c. Values a and b are chosen at random, while $c = a \cdot b$. The use of these triples allow Alice and Bob to securely perform multiplication in an efficient manner, as we explain next (see also [8]).

Consider numbers x and y that are secret shared between Alice and Bob. Alice and Bob can locally compute $[\![d]\!] = [\![x]\!] - [\![a]\!]$ and $[\![e]\!] = [\![y]\!] - [\![b]\!]$ with their respective shares. Because Alice and Bob used their shares of a, b as OTPs, they can make their shares of d and e public without revealing anything about their shares of x, y, a, or b. Now, consider the following:

$$\begin{aligned} x \cdot y &= ((x - a) + a)((y - b) + b) \\ &= (d + a)(e + b) \\ &= de + db + ae + ab \\ &= de + db + ae + c \end{aligned} \tag{3.3}$$

This identity allows the parties to compute shares of the product $x \cdot y$. Specifically, one party computes $d[\![b]\!] + e[\![a]\!] + [\![c]\!]$, and another computes $de + d[\![b]\!] + e[\![a]\!] + [\![c]\!]$. Which party retains the de value on their end will depend on the asymmetric bit.

All protocols we perform can be broken down into secure multiplications and additions of secret shared values. By breaking our protocols down to these atomic operations, we can guarantee security of each party's data, assuming they do not deviate from these protocols.

3.3.4 Comparison of Secret Shared Values

An essential operation in π_{XT} is calculating relative ordering between values. Securely ordering elements allows us to determine what the best split for each node is, and whether or not we satisfy an early stopping condition. We define the operator $\geq^?$ on the operands $[\![x]\!]$, $[\![y]\!]$ in the following way:

$$[\![x]\!] \geq^? [\![y]\!] = [\![1]\!]_2 \quad \textbf{if} \quad x \geq y, \;\; \textbf{else} \;\; [\![x]\!] \geq^? [\![y]\!] = [\![0]\!]_2.$$

Note that the outputs of the $\geq^?$ operator are modulo 2, representing a boolean result. The $\geq^?$ operator is not functionality that is innate to additive sharing, but instead requires a somewhat complicated protocol to function. The protocol that simulates the result of $\geq^?$ will be referred to as π_{GEQ}. At a high level, π_{GEQ} takes the difference between two values x, y, and then observes the most significant bit. Because we use a two's compliment scheme, if the most significant bit of $y - x$ is 1, then $x \geq y$, otherwise $y > x$. As parameters, π_{GEQ} takes the parties secret shared values $[\![x]\!]$ and $[\![y]\!]$, and then outputs a single secret share bit modulo 2. The details regarding the protocol are outside of the scope of this paper, and as such, we will use it for the remainder of this paper as a black box.

Because π_{GEQ} returns a $[\![0]\!]_2$ or a $[\![1]\!]_2$, we can not immediately use the results. Recall that most additive shares are in the ring $\mathbb{Z}_q$, where $q = 2^\lambda$. Thus, we must convert the result of π_{GEQ} from $\mathbb{Z}_2$ to $\mathbb{Z}_q$. Unfortunately, because the rings zeros do not match up, we can't just declare our values in $\mathbb{Z}_2$ to be in the ring $\mathbb{Z}_q$. Suppose we did try to seamlessly go from $\mathbb{Z}_2$ to $\mathbb{Z}_q$. Further suppose the parties share a bit $[\![\mathsf{b}]\!]_2 \;:=\; (\mathsf{b}_0,\; \mathsf{b}_1)$, where the notation $[\![\mathsf{x}]\!] \;:=\; (\mathsf{x}_0,\; \mathsf{x}_1)$ indicate that party 0 has the share of x as x_0, and party 1 has their share as x_1. Consider the following cases:

Case 1. $b_0 = b_1 = 0$: Given that both shares are 0, taking the sum $b_0 + b_1$ in $\mathbb{Z}_q$ retains the original value in $\mathbb{Z}_2$.

Case 2. $b_0 = 1, b_1 = 0$ *or* $b_0 = 0, b_1 = 1$: Similar to case 1, if the parties are to take the sum of their shares in $\mathbb{Z}_q$, the original value from $\mathbb{Z}_2$ holds.

Case 3. $b_0 = b_1 = 1$: Taking the sum of the parties' shares in $\mathbb{Z}_q$ would result in a value of 2; however, the sum of the values in $\mathbb{Z}_2$ should be 0.

Case 3 tells us that we can't go from one ring to another so easily. Thus, we introduce π_{2toQ}, which is defined in protocol 2.

Protocol 2: Secure Protocol π_{2toQ} converts a secret bit from a $\mathbb{Z}_2$ sharing to a $\mathbb{Z}_q$ sharing.

```
Input  : $[\![\mathsf{b}]\!]_2 := (\mathsf{b}_0, \mathsf{b}_1)$
Output: $[\![\mathsf{b}]\!]_q$
1 Alice creates the sharing $[\![\mathsf{b}_0]\!]_q = (\mathsf{b}_0, 0)$
2 Bob creates the sharing $[\![\mathsf{b}_1]\!]_q = (0, \mathsf{b}_1)$
3 $[\![\mathsf{b}]\!]_q \leftarrow [\![\mathsf{b}_0]\!]_q + [\![\mathsf{b}_1]\!]_q - 2 \cdot [\![\mathsf{b}_0]\!]_q \cdot [\![\mathsf{b}_1]\!]_q$
4 return $[\![\mathsf{b}]\!]_q$
```

Note that for case 1, the expression $[\![\mathsf{b}_0]\!]_q + [\![\mathsf{b}_1]\!]_q - 2 \cdot [\![\mathsf{b}_0]\!]_q \cdot [\![\mathsf{b}_1]\!]_q$ turns into $0+0-2\cdot 0\cdot 0 = 0$, which is in fact the correct value. For case 2, we get $1+0-2\cdot 1\cdot 0 = 1$, which is also correct. Finally, case 3 is also satisfied, giving us $1+1-2\cdot 1\cdot 1 = 0$. Since the protocol has been shown to output the correct value for all three cases, we have demonstrated correctness. By feeding our results from π_{GEQ} to π_{2toQ}, we will be able to take the secure product between our comparison results and additive shared values from $\mathbb{Z}_q$, which will be important in the pre-processing and training phase.

3.3.5 Simple MPC protocols

There are two simple protocols that we have not gone over yet, but are important to π_{XT}. The first protocol is the secure element wise product between vectors, π_{H}, also known as the Hadamard product. Since we have already developed secure multiplication, π_{H} follows naturally. To preform π_{H}, we take the secure product of parallel elements of two arrays, and return the shares of the product in a third array. The second protocol is the truncation protocol, π_{trunc}. Recall that after the product of two secret shared values, we have to preform truncation because the fractional bits double from a to $2a$. Fortunately, π_{trunc} can be done locally by right-shifting the secret shared value by a bits.

3.3.6 Dummy Operations to Protect against Side Channel Attacks

In chapters 4 and 5, we will see many MPC protocols that appear to be doing redundant work. For example, while training a DT, if we satisfy an early stopping condition, we will have to continue the training as if a stopping condition has not yet been met. An in-the-clear algorithm would not have to do this; however, this is a vital process in an MPC setting. If π_{XT} were to stop training a tree when an early stopping condition was met, it would inform the parties about the uniformity of the data at that iteration of the training algorithm, and thus could leak information about the distribution of the data. In an honest-but-curious setting, nothing should be revealed except for what is agreed to be public knowledge, and what one can extrapolate about the data given the classification of an unseen instance. Typically speaking, anything in the algorithm that could reveal information about the flow of logic in the program should be avoided.

3.4 Communication Between Parties

3.4.1 Optimizing MPC Protocols

In MPC, operations are typically distinguished between an online and an offline phase. Online phases require communication between the parties, whereas offline phases just require computation that is local to each party's machine. Operations such as multiplication of secret shared values require parties to communicate. Communication tends to be the most time consuming activity in a MPC algorithm. Because of this, when we discuss the efficiency of an algorithm, we will typically do so in terms of communication complexity.

3.4.2 Batching Operations

In attempt to reduce communication complexity, we batch operations that require communication into a single function call. This allows us to significantly reduce the communication complexity by a linear factor in most cases. Although not specified in the upcoming pseudocode, all protocols in this thesis that require communication can be batched in some way, and are batched in the implementation of π_{XT}.

Chapter 4

PRE-PROCESSING

4.1 Overview

In this chapter, we present the first part of our π_{XT} protocol for secure training of extra trees classifiers, the pre-processing phase. The pre-processing phase is made up from several protocols that allow us to take the burden off of the actual tree training phase that will be discussed in Chapter 5.

4.1.1 Adaptation from Original Extra Trees Algorithm

We altered some aspects of the original extra trees algorithm for efficiency. Possibly the most substantial change from the extra trees algorithm in-the-clear to the MPC based protocol π_{XT} are the random splits at each node. In the extra trees algorithm, while growing a decision tree (DT) one dynamically selects a random pool of attributes to split on at each node without replacement. In π_{XT}, not only do we not remove attributes, we do not change the pool of attributes we consider at each node within a tree. Instead, each node in a single DT has to choose its split from the same set of attributes chosen for that DT during the pre-procssing phase, i.e. we select a static random pool of attributes to split on *per tree* instead of *per node*. This change drastically decreases the time π_{XT} takes. Securely constructing sets of attributes to split on for a single node has linear communication complexity in respect to the number of attributes per split, so constructing such sets for every node in the ensemble is an expensive operation. We'll see in Chapter 6 that this choice can reduce our accuracy for ensembles where we do not consider many attributes for each split, but we can regain our accuracy by increasing the number of attributes at each split while still maintaining competitive runtimes.

4.2 Pre-Processing with No Security

To get intuition in regards to the pre-processing phase, we will first discuss the altered algorithm without any security. To start, we need to process all of the columns (attributes) of data and find their minimum and maximum values. Continuing with the example from Chapter 2, we can find the minimum and maximum value of each column in the data set.

A_1	A_2	A_3	A_4	label
1.5	1.2	1.0	1.0	b
1.7	1.5	2.2	1.4	b
2.1	1.2	3.0	1.6	b
2.6	2.2	1.4	2.0	r
2.7	2.7	2.0	2.4	r
4.8	3.9	3.4	2.6	y
2.2	1.0	1.2	3.0	b
2.4	2.6	2.4	3.4	b
3.8	4.1	3.6	3.6	y

Define the 2-tuple $(min, max)_i$ to be the minimum and maximum values of the attribute A_i. We have $(1.5, 4.8)_1$, $(1.0, 4.1)_2$, $(1.0, 3.6)_3$, and $(1.0, 3.6)_4$. We then randomly select k attributes which will act as our candidate attributes to split on. Suppose we select columns 1 and 4, giving us

A_1	A_4
1.5	1.0
1.7	1.4
2.1	1.6
2.6	2.0
2.7	2.4
4.8	2.6
2.2	3.0
2.4	3.4
3.8	3.6

For each candidate attribute a_j, we will create a random threshold α_j between the minimum and maximum values in a_j. For our example, a_1 corresponds to A_1, and a_2 corresponds to A_4. Let us suppose that $\alpha_1 = 3.6$, and $\alpha_2 = 2.2$. We can use α_j to effectively turn the numerical attribute a_j into a binary categorical attribute. Numerical values in a_j that are less than α_j are replaced with a zero, and values greater than or equal to α_j are replaced with a one. In our example, a_1 and a_2 would turn into the following:

a_1	a_2
0	0
0	0
0	0
0	0
0	1
1	1
0	1
0	1
1	1

We can now use the binarized columns to train a DT using Quinlan's ID3 algorithm [10], a well-known algorithm to train a DT with categorical attributes (of which binarized attributes are a special case). Detailed pseudo-code for the pre-processing in-the-clear is provided in Algorithm 3. We repeat the steps mentioned here for as many trees as we want in our ensemble.

4.3 Secure Pre-Processing in the Honest-but-Curious Setting

4.3.1 Secure MinMax Protocol

Before we present π_{XT}, we will first go over a protocol that will allow us to find the minimum and maximum values for each attribute in a privacy-preserving manner. In the previous section, finding the range of values of an attribute was simple. Here, however, all values are secret shared between the two parties, Alice and Bob. Thus, we will have to use a protocol with communication in order to discern the minimum and maximum values in each column. To this end, we introduce the π_{minmax} protocol, described in Protocol 4. The protocol describes a naive way of computing the min and max values in a single column. Lines 1, 2, 3, 5, 6 and 8 of π_{minmax} require communication between the parties, giving us linear communication complexity in terms of the number of values of elements in the list.

An optimization can be made in the communication rounds by comparing secret shares pairwise in a batching π_{GEQ} protocol. For instance, suppose we have n values to process. Instead of comparing a value $[\![d_i]\!]$ against the proposed min and max like in lines 7 and 8, we compare all ordered pairs $([\![d_1]\!], [\![d_2]\!])...([\![d_{n-1}]\!], [\![d_n]\!])$ in a single batch. The half of the values that were $\geq$ to their counterpart would then all be compared to each other in the proceeding batching comparison, and the other half would all be compared together in the same batching scheme. For n of odd parity, we will simply omit the last element from comparisons until adding it back in would make for an even number of values to compare. This proposed batching scheme would allow us to find the minimum and maximum values for each individual attribute in the data set with logarithmic communication complexity in

Algorithm 3: Adapted Algorithm for Training an Extra-Trees Classifier, with Detailed Pre-processing Steps

Input : A training set S with continuous data and n samples (each sample has f features), the number k of features to consider in each tree, the number m of trees in the ensemble, the depth d of each tree.

Output: An ensemble of trees $XT = t_1, \ldots, t_m$.

```
1  For each feature $j$, find its minimum and maximum values, $\min_j$ and $\max_j$.
2  for $i \leftarrow 1$ to $m$ do
3      Select $k$ random, distinct indexes $j_1, \ldots, j_k \in \{1, \ldots, f\}$.
4      for $\ell \leftarrow 1$ to $k$ do
5          For a uniformly random $r \in (0,1)$, $\alpha_\ell \leftarrow r \cdot (\max_{j_\ell} - \min_{j_\ell}) + \min_{j_\ell}$.
6          for $s \leftarrow 1$ to $n$ do
7              if $S[s, j_\ell] \geq \alpha_\ell$ then
8                  $S'[s, \ell] \leftarrow 1$
9              else
10                 $S'[s, \ell] \leftarrow 0$
11             end
12         end
13     end
14     Train a decision tree $t_i$ of depth $d$ on $S'$ using ID3.
15 end
16 return $XT = t_1, \ldots, t_m$.
```

Protocol 4: Secure Min/Max-Finding Protocol π_{minmax}

```
Input  : $[\![D]\!]$, number $n$ of elements in $[\![D]\!]$
Output: $[\![d_{\mathsf{min}}]\!]$, $[\![d_{\mathsf{max}}]\!]$
1  Let $[\![\geq^?]\!] \leftarrow \pi_{\mathsf{2toQ}}(\pi_{\mathsf{GEQ}}([\![d_2]\!], [\![d_1]\!]))$
2  Let $[\![d_{\mathsf{min}}]\!] \leftarrow [\![\geq^?]\!] \cdot [\![d_1]\!] + (1 - [\![\geq^?]\!]) \cdot [\![d_2]\!]$
3  Let $[\![d_{\mathsf{max}}]\!] \leftarrow [\![\geq^?]\!] \cdot [\![d_2]\!] + (1 - [\![\geq^?]\!]) \cdot [\![d_1]\!]$
4  for $i \leftarrow 3$ to $n$ do
5      $[\![\geq^?_{\mathsf{min}}]\!] \leftarrow \pi_{\mathsf{2toQ}}(\pi_{\mathsf{GEQ}}([\![d_i]\!], [\![d_{\mathsf{min}}]\!]))$
6      $[\![\geq^?_{\mathsf{max}}]\!] \leftarrow \pi_{\mathsf{2toQ}}(\pi_{\mathsf{GEQ}}([\![d_i]\!], [\![d_{\mathsf{max}}]\!]))$
7      $[\![d_{\mathsf{min}}]\!] \leftarrow [\![\geq^?_{\mathsf{min}}]\!] \cdot [\![d_{\mathsf{min}}]\!] + (1 - [\![\geq^?_{\mathsf{min}}]\!]) \cdot [\![d_i]\!]$
8      $[\![d_{\mathsf{max}}]\!] \leftarrow [\![\geq^?_{\mathsf{max}}]\!] \cdot [\![d_i]\!] + (1 - [\![\geq^?_{\mathsf{max}}]\!]) \cdot [\![d_{\mathsf{max}}]\!]$
9  end
10 return $[\![d_{\mathsf{min}}]\!]$, $[\![d_{\mathsf{max}}]\!]$
```

terms of the total number of attributes. Further optimizations can be performed by batching every attribute into π_{minmax}. All optimizations mentioned are implemented in the π_{XT} source code.

4.3.2 Secure Pre-Processing Protocol

The π_{XT} algorithm is given in pseudo-code as Protocol 5. Lines 1-12 of the pseudo-code describe the pre-processing phase, while line 13 involves the training phase. Note that despite introducing a pre-processing phase, which the original extra trees algorithm did not have, the π_{XT} protocol is logically equivalent to the original extra trees algorithm; barring the few modifications we have made for efficiency, which can easily be re-implemented into π_{XT} if desired. Line 1 of the protocol starts us off with an offline phase, where the TI produces feature selection matrices (FSMs), where the word feature is used synonymously with attribute. These matrices consist of $[\![0]\!]$'s and $[\![1]\!]$'s, and among other things, will allow us to securely extract certain columns from the data set. In line 4, the multiplication is in reference to matrix multiplication between the secret shared data set $[\![S]\!]$ of size $n \times f$, and a FSM of size $f \times k$, where n is the number of instances in the data set, f is the number of attribute or feature columns in the data set, and k is the number of features we consider at each split. So for example, suppose that Alice and Bob have shares of the data set from Chapter 2. If we define Q to be the function from equation 3.1 that maps real values r to $\mathbb{Z}_q$, the parties' sharings of the attribute values would look like the following.

A_1	A_2	A_3	A_4
$[\![\mathrm{Q}(1.5)]\!]$	$[\![\mathrm{Q}(1.2)]\!]$	$[\![\mathrm{Q}(1.0)]\!]$	$[\![\mathrm{Q}(1.0)]\!]$
$[\![Q(1.7)]\!]$	$[\![Q(1.5)]\!]$	$[\![Q(2.2)]\!]$	$[\![Q(1.4)]\!]$
$[\![Q(2.1)]\!]$	$[\![Q(1.2)]\!]$	$[\![Q(3.0)]\!]$	$[\![Q(1.6)]\!]$
$[\![Q(2.6)]\!]$	$[\![Q(2.2)]\!]$	$[\![Q(1.4)]\!]$	$[\![Q(2.0)]\!]$
$[\![Q(2.7)]\!]$	$[\![Q(2.7)]\!]$	$[\![Q(2.0)]\!]$	$[\![Q(2.4)]\!]$
$[\![Q(4.8)]\!]$	$[\![Q(3.9)]\!]$	$[\![Q(3.4)]\!]$	$[\![Q(2.6)]\!]$
$[\![Q(2.2)]\!]$	$[\![Q(1.0)]\!]$	$[\![Q(1.2)]\!]$	$[\![Q(3.0)]\!]$
$[\![Q(2.4)]\!]$	$[\![Q(2.6)]\!]$	$[\![Q(2.4)]\!]$	$[\![Q(3.4)]\!]$
$[\![Q(3.8)]\!]$	$[\![Q(4.1)]\!]$	$[\![Q(3.6)]\!]$	$[\![Q(3.6)]\!]$

Then, let us suppose that the TI from line 1 of the protocol randomly selected the first and the third column to extract to create a subset of data for the $i'th$ tree in the ensemble. The parties would have a sharing of the following FSM, denoted as $[\![FS^{(i)}]\!]$.

$$[\![FS^{(i)}]\!] = \begin{bmatrix} [\![1]\!] & [\![0]\!] \\ [\![0]\!] & [\![0]\!] \\ [\![0]\!] & [\![1]\!] \\ [\![0]\!] & [\![0]\!] \end{bmatrix}$$

Let the attribute values above describe a secret shared data set $[\![S]\!]$. The product $[\![S]\!] \cdot [\![FS^{(i)}]\!]$ would result in a vertically partitioned secret shared subset of the data set of size $n \times k$ as demonstrated below.

$$\begin{bmatrix} [\![Q(1.5)]\!] & [\![Q(1.2)]\!] & [\![Q(1.0)]\!] & [\![Q(1.0)]\!] \\ [\![\mathrm{Q}(1.7)]\!] & [\![\mathrm{Q}(1.5)]\!] & [\![\mathrm{Q}(2.2)]\!] & [\![\mathrm{Q}(1.4)]\!] \\ [\![\mathrm{Q}(2.1)]\!] & [\![\mathrm{Q}(1.2)]\!] & [\![\mathrm{Q}(3.0)]\!] & [\![\mathrm{Q}(1.6)]\!] \\ [\![\mathrm{Q}(2.6)]\!] & [\![\mathrm{Q}(2.2)]\!] & [\![\mathrm{Q}(1.4)]\!] & [\![\mathrm{Q}(2.0)]\!] \\ [\![\mathrm{Q}(2.7)]\!] & [\![\mathrm{Q}(2.7)]\!] & [\![\mathrm{Q}(2.0)]\!] & [\![\mathrm{Q}(2.4)]\!] \\ [\![\mathrm{Q}(4.8)]\!] & [\![\mathrm{Q}(3.9)]\!] & [\![\mathrm{Q}(3.4)]\!] & [\![\mathrm{Q}(2.6)]\!] \\ [\![\mathrm{Q}(2.2)]\!] & [\![\mathrm{Q}(1.0)]\!] & [\![\mathrm{Q}(1.2)]\!] & [\![\mathrm{Q}(3.0)]\!] \\ [\![\mathrm{Q}(2.4)]\!] & [\![\mathrm{Q}(2.6)]\!] & [\![\mathrm{Q}(2.4)]\!] & [\![\mathrm{Q}(3.4)]\!] \\ [\![\mathrm{Q}(3.8)]\!] & [\![\mathrm{Q}(4.1)]\!] & [\![\mathrm{Q}(3.6)]\!] & [\![\mathrm{Q}(3.6)]\!] \end{bmatrix} \cdot \begin{bmatrix} [\![1]\!] & [\![0]\!] \\ [\![0]\!] & [\![0]\!] \\ [\![0]\!] & [\![1]\!] \\ [\![0]\!] & [\![0]\!] \end{bmatrix} = \begin{bmatrix} [\![Q(1.5)]\!] & [\![Q(1.0)]\!] \\ [\![\mathrm{Q}(1.7)]\!] & [\![\mathrm{Q}(2.2)]\!] \\ [\![\mathrm{Q}(2.1)]\!] & [\![\mathrm{Q}(3.0)]\!] \\ [\![\mathrm{Q}(2.6)]\!] & [\![\mathrm{Q}(1.4)]\!] \\ [\![\mathrm{Q}(2.7)]\!] & [\![\mathrm{Q}(2.0)]\!] \\ [\![\mathrm{Q}(4.8)]\!] & [\![\mathrm{Q}(3.4)]\!] \\ [\![\mathrm{Q}(2.2)]\!] & [\![\mathrm{Q}(1.2)]\!] \\ [\![\mathrm{Q}(2.4)]\!] & [\![\mathrm{Q}(2.4)]\!] \\ [\![\mathrm{Q}(3.8)]\!] & [\![\mathrm{Q}(3.6)]\!] \end{bmatrix}$$

We will later binarize this subset of the data as we did in Section 4.2. By executing line 2, the parties Alice and Bob calculate the minimum and maximum values of each column and stores the values into the vectors $[\![min]\!]$ and $[\![max]\!]$. In our running example, we'd have

$$[\![min]\!] = ([\![Q(1.5)]\!], [\![Q(1.2)]\!], [\![Q(1.0)]\!], [\![Q(1.0)]\!])\,,$$

$$[\![max]\!] = ([\![Q(4.8)]\!], [\![Q(3.9)]\!], [\![Q(3.6)]\!], [\![Q(3.6)]\!])\,.$$

Line 5 grabs a slice of random ratios generated by the TI which are approximately equivalent to $r \in (0, 1)$ in $\mathbb{R}$. We approximate r with $[1, 2^{a-1}]$ in the ring. Recall that the value a is in reference to the quantity of bits we reserve for the decimal portion of our ring values. Thus, $2^a - 1$ is the largest value in the ring less than $1 \in \mathbb{R}$. Line 6 then uses these random ratios to construct a vector $[\![\alpha]\!]$ consisting of values that will allow us to later split the columns into two categories, values $<$ than the split, and values $\geq$ than the split. To calculate $[\![\alpha]\!]$, Alice and Bob start by taking the difference between the maxiumum and minimum values of each column. In our example, this would result in

$$[\![max]\!] - [\![min]\!] = [\![range]\!] = ([\![Q(3.3)]\!], [\![Q(2.7)]\!], [\![Q(2.6)]\!], [\![Q(2.6)]\!])\,.$$

Note that depending on our value for a, the difference between two values such as $[\![Q(4.8)]\!] - [\![Q(1.5)]\!]$, may not exactly result in $[\![Q(3.3)]\!]$. This is because there may not be enough fractional bits reserved for the true value of the difference. However, the previously mentioned difference should sufficiently approximate $[\![Q(3.3)]\!]$ if it is not directly equal. We can then use $FS^{(i)}$ to securely extract the values from $[\![range]\!]$ that correspond to the columns of our subset of data. In other words, the parties calculate $[\![range]\!] \cdot [\![FS^{(i)}]\!]$, which gives us $([\![Q(3.3)]\!], [\![Q(2.6)]\!])$, the first and third values in $[\![range]\!]$. We then use the Hadamard product π_{H} to take the product between the random ratios and the $[\![range]\!]$ values as dictated by $FS^{(i)}$, after which we will have to truncate the products with a truncation protocol, π_{trunc}. Truncation is necessary because the bits dedicated to the decimals would have doubled from a to $2a$. Finally, having the parties add $[\![min]\!]$ to the product as indicated by $FS^{(i)}$ guarantees that there are k values in the parties share of $[\![\alpha]\!]$, all between the minimum and maximum values from the columns in which they were derived from. Lines 9 and 10 of Protocol 5 are logically equivalent to lines 8 and 10 from Algorithm 3 in that they distribute values that binarize the data set to allow us to train DTs. The primary difference is in line 10 of the protocol: we also construct the negation of the binarized set. We do this because having the negation of the set available is helpful during the training phase. Note that every **for** loop in the protocol can be replaced by batched operations, significantly reducing communication complexity. We train our DTs with a variant of a secure version of the ID3 algorithm, π_{SID3T}, which we will discuss in the next chapter. By the end of the pre-processing phase, each party will have shares of the initial state of several binarized column-wise subsets of $[\![S]\!]$, where each subset is dependent on the random features selected for that tree by the TI.

Protocol 5: Protocol for Securely Training an Extra-Trees Classifier π_{XT}

```
Input  : A secret shared training set $[\![S]\!]$ with continuous data and $n$ samples (each
         sample has $f$ features), the number $k$ of features to consider in each tree,
         the number $m$ of trees in the ensemble, the depth $d$ of each tree.
Output: A secret shared ensemble of trees $[\![XT]\!] = [\![t_1]\!], \dots, [\![t_m]\!]$.
1  (Offline Phase) The TI secret shares $m$ random 0/1-valued feature selection matrices
    $[\![FS^{(1)}]\!], \dots, [\![FS^{(m)}]\!]$ of size $f \times k$, where each column contains a single 1, and no
    two rows have more than a single 1. The TI also distributes $k \cdot m$ uniformly
    random ratios $[\![r^{(1)}]\!], \dots, [\![r^{(k \cdot m)}]\!] \in [1, 2^a - 1]$ (which approximates $r \in (0,1)$ in $\mathbb{R}$).
2  Compute the vectors $[\![min]\!]$ and $[\![max]\!]$, by using $([\![min_j]\!], [\![max_j]\!]) \leftarrow \pi_{\mathsf{minmax}}([\![S_j]\!], n)$
    for each column $S_j$ $(j = 1, \dots, f)$ of $S$.
3  for $i \leftarrow 1$ to $m$ do
4      $[\![S_{\mathsf{FS}}]\!] \leftarrow [\![S]\!] \cdot [\![FS^{(i)}]\!]$
5      $[\![\boldsymbol{r}]\!] \leftarrow ([\![r^{((i-1)k+1)}]\!], \dots, [\![r^{(ik)}]\!])$
6      $[\![\boldsymbol{\alpha}]\!] \leftarrow \pi_{\mathsf{trunc}}(\pi_{\mathsf{H}}([\![\boldsymbol{r}]\!], ([\![max]\!] - [\![min]\!]) \cdot [\![FS^{(i)}]\!])) + [\![min]\!] \cdot [\![FS^{(i)}]\!]$
7      for $\ell \leftarrow 1$ to $k$ do
8          for $p \leftarrow 1$ to $n$ do
9              $[\![D[p, \ell, 1]]\!]_2 \leftarrow \pi_{\mathsf{GEQ}}([\![S_{\mathsf{FS}}[p, \ell]]\!], [\![\alpha[\ell]]\!])$
10             $[\![D[p, \ell, 0]]\!]_2 \leftarrow 1 - [\![D[p, \ell, 1]]\!]_2$
11         end
12     end
13     Let $[\![t_i]\!]$ be the decision tree of depth $d$ trained using $\pi_{\mathsf{SID3T}}$ with the data set
        $[\![D]\!]_2$.
14 end
15 return $[\![XT]\!] = [\![t_1]\!], \dots, [\![t_m]\!]$.
```

Chapter 5

TRAINING THE ENSEMBLE

After the pre-processing phase, each tree we wish to train in the ensemble will be trained on a column-wise subset of the original data set, where the attribute columns purely consist of $[\![0]\!]'s$ and $[\![1]\!]'s$, effectively turning each column into a binary categorical column. Given this representation of the data, we can efficiently train each tree in the ensemble with π_{SID3T} (see below).

5.1 Training a Decision Tree

π_{SID3T} was developed by Sebastiaan de Hoogh, Berry Schoenmakers, Ping Chen, and Harm op den Akker in their paper, *Practical Secure Decision Tree Learning in a Teletreatment Application* (see [6]). As the title suggests, the protocol was constructed to be a practical MPC protocol based off of the popular ID3 algorithm for training DTs in-the-clear. The protocol works by taking data with categorical attributes and determining at each step which attribute will maximize the protocol's scoring metric. In determining the best attribute to split on, we have to consider each unique value in each attribute. Fortunately, our columns only have two different values, namely 0 and 1. Furthermore, we've reduced the amount of features each tree is trained on from the original number f, to the amount of features we consider at each split k, making the attribute scoring phase fairly inexpensive. The way in which we score our splits is with a variant of the Gini index constructed by Hoogh et. al., which is described in the authors' paper.

5.2 Modification of π_{SID3T}

5.2.1 Concealing the shape of the tree

In our honest-but-curious setting, the original π_{SID3T} protocol has one aspect which violates the assumption, that is, it can reveal the shape of the DT. This is a violation because it can reveal which nodes split on which attributes. Since each branch in a DT will correspond to a single value in an attribute, it can be easily inferred by observing how many branches come from each node which attribute the node splits on. This allows parties to infer the distribution of their combined data, which is meant to be kept secret. The original authors did this for the sake of practicality. To stay true to the original ID3 algorithm, the π_{SID3T} protocol needs to create a single branch per unique value in each column of data. To truly hide the structure of the tree in this scenario, one would likely have to add redundant branches to each non-leaf node, greater than or equal to the maximum amount of unique values among all categorical attributes. Lets say that, for example, we have 4 categorical attributes, 3 of which have 5 unique values associated to them, and 1 which has 50 unique values associated to it. Then, to truly hide which node splits on what attribute, we would have to let each node break off into at least 50 branches. For the three attributes that only had 5 unique values, the node that splits off of them would have no less than 45 redundant branches leading to dummy nodes. We would have to increase the number of branches from 50 to a higher number if we also wanted to conceal how many unique values for each attribute the parties have. Since the trees are recursively built from each node, introducing these dummy nodes would create a giant tree structure that could take weeks to train depending on the data set.

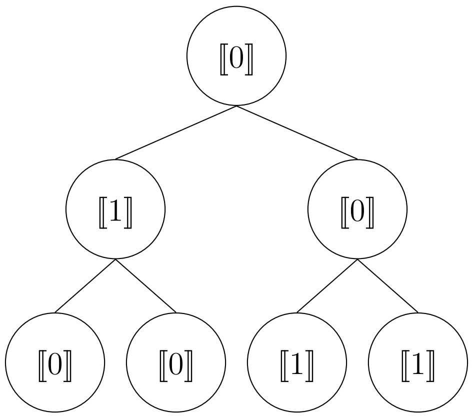
The issue that the original π_{SID3T} algorithm has does not affect us to the same degree. Because each column of our data only has two unique values, we exclusively construct binary trees. Furthermore, in practice, ensembles of many shallow trees tend to provide similar accuracy as ensembles with deep trees, hence it is common to force the trees in an ensemble to be shallow by making sure they do not grow past a certain depth. This gives us the opportunity to hide the structure of our tree by adding dummy nodes without creating giant tree structures. Thus, we make the minor modification/improvement to π_{SID3T} that makes it exclusively produce full trees that grow to a predetermined depth d which is agreed upon by the parties before training. This allows us to remain in a honest-but-curious setting without sacrificing too much time efficiency.

5.2.2 Stopping conditions

Although we always grow our trees to a certain maximum depth, each tree still may hit an early stopping condition, in which case the node that hit the condition should classify the data, and dummy nodes should be created under it to conceal the shape of the tree. The original extra trees algorithm stops growing a tree if one of the following three things happens: (1) all of the class labels in the nodes' subset of data are the same, (2) the amount of data (rows) that has reached the node is under a certain threshold, or (3) each column of data only has one distinct value remaining. The first two stopping conditions are kept in our training algorithm, but the last stopping condition was removed. Given our preference for shorter DTs, we do not typically split the data very frequently, making it unlikely that the last stopping condition is satisfied. As such, the stopping condition was removed for the sake of saving time during the training phase.

5.2.3 Determining which nodes classify

The introduction of growing our DTs to some full depth d, all the while having early stopping conditions, presents a new problem, namely when Alice and Bob want to classify new instances with the trained DT in a privacy-preserving manner. If the DT was trained with the original π_{SID3T} protocol, all of the classifications would reside in the leaf nodes; however, in our rendition of π_{SID3T}, many of the leaf nodes will be dummy nodes which should not label the data. To remedy this, a secret shared classification bit is created for each node. A classification bit of 1 tells us that the node should label, a value of 0 tells us not to label. Take the following DT with its classification bits shown in each node as an example.



The tree above tells us that the left child of the root labels the data, and its children simply act as dummy nodes. In contrast, the right child of the root does not label the data, but its children do. Every node in the tree attempts to label the data based on the most frequent label of the training examples that have reached the node during training; Alice and Bob determine this information and store the most frequent label as additive secret shares during the training process. With that being said, we only care about the nodes with classification bits of 1, as this indicates that the node should actually label the data. For Alice and Bob to securely classify unseen instances of data with our ensemble, we may use methods similar to the one described in the paper *Efficient and Private Scoring of Decision Trees, Support Vector Machines and Logistic Regression Models based on Pre-Computation* (see [5]) along with our classification bits.

Chapter 6
RESULTS

We implemented our secure extra trees algorithm in Rust [1]. With our implementation, we performed a series of experiments to evaluate the accuracy of π_{XT} compared to the original extra trees algorithm (see Table 6.2), and its runtime (see Table 6.1). The runtime results were obtained on AWS c5.9xlarge machines with 36 vCPUs, 72.0 GiB Memory. Each of the parties ran on a separate machine (connected with a Gigabit Ethernet network), which means that the results in Table 6.1 cover communication time in addition to computation time.

Table 6.1 contains accuracy and runtime results for π_{XT} and accuracy for extra trees in-the-clear on three different datasets that vary in terms of the number of instances and the total number of features. The results in Table 6.1 were obtained with an ensemble of 100 trees all of depth 1. Since the ensemble is very simple, it gives us a good idea of how our algorithm performs in general. DTs of depth 1 sometimes have a hard time classifying the data. In fact, the ensembles obtained for the ECG and lower back pain data always voted for the most frequent label, giving us majority baseline results for accuracy. Those two datasets can be tough to classify correctly, even for the original extra trees algorithm that is implemented in sklearn[2]. Notice that the accuracy from the secure π_{XT} algorithm, and the original in-the-clear algorithm have a 2% discrepancy in accuracy. This difference can be attributed to the fact that the way our nodes classify data is by identifying the most frequent class label, and setting its vote to that. The algorithm in-the-clear simply returns the frequency of each class label, which allows us to weight votes. The latter sometimes

[1]https://bitbucket.org/uwtppml

[2]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html

preforms slightly better than the one we currently have implemented. Our protocol π_{XT} can easily be switched to returning class frequencies if desired.

			π_{XT}		in-the-clear
data set	# instances	# features	Acc	Time	Acc
BC: Breast Cancer [1]	569	30	0.84	1.1	0.86
ECG: ECG Heartbeat [2]	14,552	187	0.72	63.0	0.72
BACK: Lower Back Pain Symptoms [3]	310	12	0.68	0.5	0.68

Table 6.1: Accuracy obtained with 5-fold cross-validation, and runtime results (in min) for secure training of extra trees classifiers. The max tree depth d is set to 1 with a total of $m = 100$ trees in the ensemble, and the number features per split is $k \approx \sqrt{f}$, where f is the total number of features

Recall that, for efficiency, instead of creating a subset of data for each node in the ensemble to be trained on, we create a single subset of data for each tree to be trained on. In the case where our depth d is equal to 1, these approaches are equivalent since we only ever make one split in the data. For $d \geq 2$, this change has an increased chance to effect our accuracy, but in all of our experiments, we've been able to regain this accuracy by adjusting the hyperparameters. Table 6.2 demonstrates this idea. In the Table, we see a large discrepancy in accuracy between making subsets for each node, or using the original extra-trees classifier, and making subsets for each tree when we only consider 5 features per split. This accuracy is regained by increasing the number of features. The initial drop in accuracy is likely because for small feature counts, we run the risk of randomly selecting subsets of data that do not describe the rest of the data in a meaningful way. When we create subsets per node, this is not a big issue since it is only one node that has a poor subset. We do not have this luxury when we create a single subset of data for each tree to

[1]https://www.kaggle.com/uciml/breast-cancer-wisconsin-data

[2]https://www.kaggle.com/shayanfazeli/heartbeat

[3]https://www.kaggle.com/sammy123/lower-back-pain-symptoms-dataset

	π_{XT} with discr. per tree		π_{XT} with discr. per node		in-the-clear
k	Acc	Time	Acc	Time	Acc
5	82.3	0.6	92.5	3.1	93.54
10	93.54	1.3	94	8	94.69
15	94.69	1.8	94.69	11.5	94

Table 6.2: Accuracy and runtime results (in min) of π_{XT} protocol for secure training of an extra-trees classifier and accuracy for the in-the-clear algorithm on the **Breast Cancer data set** with a max depth of 3, and 50 trees in the ensemble

be trained on. In this case, if the subset contains poor features, the tree is stuck with the subset for its entire duration of training, giving us a relatively high probability of having trees that do not classify the data well. By increasing the number of features in each subset, we reduce the chance of this happening, until eventually there is no noticeable difference between discretizing per tree, discretizing per node, and the original extra-trees algorithms results.

An alternative look into the accuracy between discretizing per tree and per node is shown in Figure 6.1. As seen in Figure 6.1, we can obtain the same accuracy by discretizing per node when we discretize per tree. Figure 6.2 shows the time we save by discretizing per tree as opposed to per node.

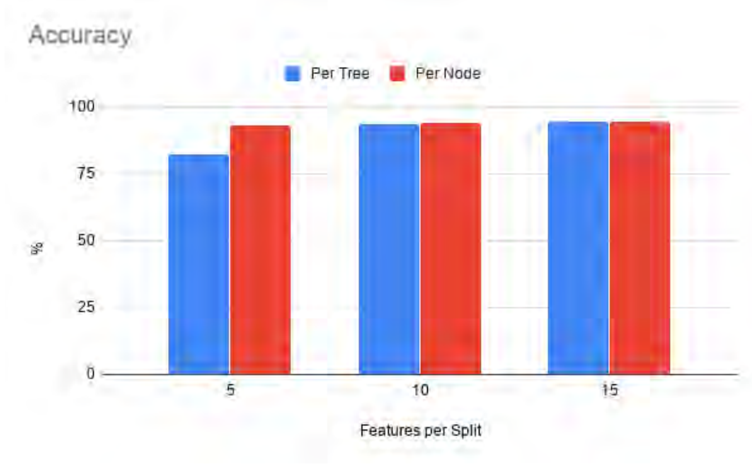


Figure 6.1: Comparing accuracy of discretizing per tree and discretizing per node on the **Breast Cancer data set** with a max depth of 3, 50 trees in the ensemble, and varying features per split

Given Figure 6.2, it is clear that we save alot of time by discretizing per tree as opposed to per node. When we discretize per node, we have to make a subset of the data for each node in the ensemble. The amount of nodes in the tree are exponentially dependent on d. Thus, creating these additional subsets requires the parties to jointly binarize an exponential amount of datasets whose amount of columns are dependent on the features per split. Alternatively, discretizing per tree only requires us to binarize as many subsets of data as there are trees in the ensemble. Thus, it is ideal to discretize per tree, unless it is highly suspected that the dataset our algorithm is trained on will not allow for a meaningful classifier when discretizing per tree.

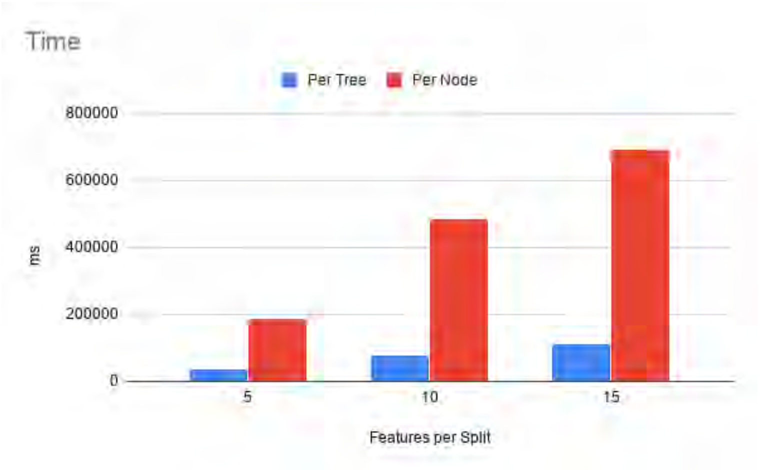


Figure 6.2: Comparing accuracy of discretizing per tree and discretizing per node on the **Breast Cancer data set** with a max depth of 3, 50 trees in the ensemble, and varying features per split

Chapter 7
CONCLUSIONS

In this thesis, we proposed π_{XT}, the first Multiparty Computation (MPC) based protocol for training Extra Trees classifiers in a privacy-preserving manner. π_{XT} is composed of MPC sub-protocols that aggregate data between two parties to jointly compute a machine learning (ML) model without requiring the parties to divulge their data to one another. This has important implications, as there exist many real life scenarios when two data owners may want to combine their data together to train a state-of-the art ML model, but do not wish to reveal their data to each other.

Extra Trees classifiers are a kind of decision tree ensembles. Most existing work on privacy-preserving training of decision trees with MPC has been done for datasets with *categorical* input attributes. In many data science applications, data is continuous instead of categorical, or a mix of both. Prior to our work, the few existing proposals for MPC based training of decision trees on *continuous* attributes were based off of Quinlan's C4.5 algorithm, which requires sorting the data, a very expensive operation in the context of MPC. Thanks to its random nature, the Extremely Randomized Trees algorithm allows to create decision tree ensembles on data with continuous input attributes *without the need to sort the data*. This makes the Extra Trees algorithm an "MPC-friendly" technique to train decision tree ensembles on continuous valued data, inspiring us to develop an MPC protocol to train Extra Trees classifiers in a fully privacy-preserving manner.

To this end, we used an additive sharing scheme which allows parties to securely add and multiply shares of data together. We then broke up our π_{XT} protocol into a pre-processing phase which we constructed ourselves (Chapter 4), and an already existing secure tree training algorithm, π_{SID3T}, which we only had to make a single improvement to in order to remain in our honest-but-curious setting (Chapter 5). The purpose of the pre-processing phase was to create vertically partitioned subsets of the data, and then binarize the subsets according to some random splits. By binarizing the subsets of data, we transformed the subsets from continuous data to binary categorical data which was used to train a DT with π_{SID3T}. The modification made to π_{SID3T} was to hide the true structure of the tree by making each tree *complete*. We did this by creating dummy nodes under any node that truly classifies the data. This modification in practice does not have much effect on the runtimes for training and inference, since trees in ensembles tend to be shallow anyway, and all of our trees are binary trees. This allowed us to create a privacy-preserving machine learning (PPML) algorithm that is fast enough to be used in practice (Chapter 6), and makes zero sacrifices regarding information leakage.

There are many avenues for future work. One obvious way to extend π_{XT} is to make it work for an arbitrary number of parties. Some of our protocols, such as the comparison protocol π_{GEQ}, are only designed for a two party setting, and extending this to work with an arbitrary number of parties would require a different approach. Overcoming these hurdles could allow our protocol to be viable in more scenarios. More future work could be to extend π_{XT} from an honest-but-curious setting to a malicious adversarial setting. Recall that in the honest-but-curious setting, the parties follow the protocols, but try to learn any additional information from the data that they could. In the malicious adversary setting, parties actively deviate from protocols in order to gain as much knowledge about the other parties' data as they can and/or to cause the protocol to produce wrong results. By guarding against these sort of attacks, we could make our algorithm far safer to use at the cost of time efficiency.

BIBLIOGRAPHY

[1] Donald Beaver. Commodity-based cryptography. In *STOC*, volume 97, pages 446–455, 1997.

[2] Gopal Behera. Privacy preserving C4.5 using Gini index. In *2nd National Conference on Emerging Trends and Applications in Computer Science*, pages 1–4, 2011.

[3] Martine De Cock, Rafael Dowsley, Anderson C. A. Nascimento, Davis Railsback, Jianwei Shen, and Ariel Todoki. High performance logistic regression for privacy-preserving genome analysis. *CoRR*, abs/2002.05377, 2020.

[4] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015.

[5] Martine De Cock, Rafael Dowsley, Caleb Horst, Raj Katti, Anderson Nascimento, Wing-Sea Poon, and Stacey Truex. Efficient and private scoring of decision trees, support vector machines and logistic regression models based on pre-computation. *IEEE Transactions on Dependable and Secure Computing*, 16(2):217–230, 2019.

[6] Sebastiaan de Hoogh, Berry Schoenmakers, Ping Chen, and Harm op den Akker. Practical secure decision tree learning in a teletreatment application. In *International Conference on Financial Cryptography and Data Security*, pages 179–194. Springer, 2014.

[7] Thomas G Dietterich. Ensemble methods in machine learning. In *International Workshop on Multiple Classifier Systems*, volume 1857 of *LNCS*, pages 1–15. Springer, 2000.

[8] David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. A pragmatic introduction to secure multi-party computation. *Foundations and Trends in Privacy and Security*, 2(2-3):70–246, 2018.

[9] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. In *Machine Learning*, pages 3–42. Machine Learning, 2006.

[10] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[11] J Ross Quinlan. *C4.5: programs for machine learning*. Elsevier, 2014.

[12] Yanguang Shen, Hui Shao, and Li Yang. Privacy preserving C4.5 algorithm over vertically distributed datasets. In *2009 International Conference on Networks Security, Wireless Communications and Trusted Computing*, volume 2, pages 446–448. IEEE, 2009.

[13] Ming-Jun Xiao, Kai Han, Liu-Sheng Huang, and Jing-Yuan Li. Privacy preserving C4.5 algorithm over horizontally partitioned data. In *2006 Fifth International Conference on Grid and Cooperative Computing (GCC'06)*, pages 78–85. IEEE, 2006.