CS261 Scribe Notes: Secure Learning

Scriber: Ryan Deng

October 17, 2018

1 Introduction

Machine learning classification is prevalent in many real-world applications. However, in many use cases, the input to the classification model and the model itself must be kept secret for privacy concerns. In these notes, we will explore privacy-preserving machine learning classification, as described in [1]. Generally, the protocol for supervised machine learning is as follows:

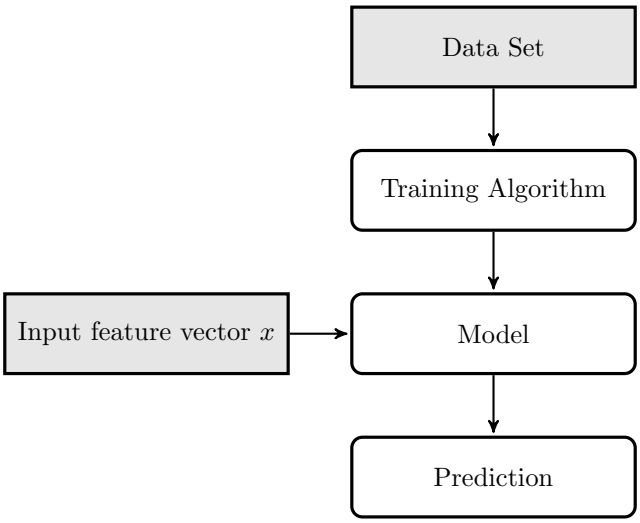

Figure 1: Supervised Machine Learning Protocol

We use a training algorithm to generate a model from a dataset. Given an input feature vector x, the model generates a prediction as an output. In the case of classification that we will focus on, the prediction is a classification class that the input vector belongs to.

2 Classification Background

Three approaches to machine learning classification are discussed in [1], and privacy-preserving methods for each are described in detail. The three methods are:

1. Hyperplane decision
2. Naïve Bayes
3. Decision Trees

We will briefly go over each in the following sections, and then go over the privacy-preserving methods for each in section 4.

2.1 Hyperplane Decision

The classification model w for hyperplane decision consists of k vectors in $\mathbb{R}^d$ ($w = \{w_i\}_{i=1}^k$). Given an input vector $x \in \mathbb{R}^d$, the class that it belongs to is identified as follows:

$$k^* = \arg\max_{i \in [k]} \langle w_i, x \rangle$$

2.2 Naïve Bayes

The classification model w for Naïve Bayes consists a series of probabilities. First, it contains the probabilities that each of the k classification class occurs. We use the notation $\{P(C = c_i)\}_{i=1}^k$ to represent this where the c_i's are the different classification classes. Additionally, the model also contains the probabilities that each component of the input vector x belongs to a classification class. More concretely, if the input vectors are in $\mathbb{R}^d$, then the model also contains the probabilities:

$$\{\{\{P(X_j = v | C = c_i)\}_{v \in D_j}\}_{j=1}^d\}_{i=1}^k$$

These denote the probability that when x belongs to a class c_i, the jth component of X is equal to value v in its domain D_j. Given an input vector $x \in R^d$, the classification class outputted by Naïve Bayes is given by:

$$\begin{aligned} k^* &= \arg\max_{i \in [k]} P(C = c_i | X = x) \\ &= \arg\max_{i \in [k]} \frac{P(C = c_i, X = x)}{P(X = x)} \\ &= \arg\max_{i \in [k]} P(C = c_i, X = x) \\ &= \arg\max_{i \in [k]} P(C = c_i, X_1 = x_1, X_2 = x_2, ..., X_d = x_d) \end{aligned} \tag{1}$$

The third equality holds since the constant multiplicative term $\frac{1}{P(X=x)}$ does not affect the arg max. In the Naïve Bayes setting, we assume that the d features of the input vector x are conditionally independent given the class it belongs to. Therefore, we use this assumption to get that:

$$\begin{aligned} k^* &= \arg\max_{i \in [k]} P(C = c_i, X_1 = x_1, ..., X_d = x_d) \\ &= \arg\max_{i \in [k]} P(C = c_i) \prod_{j=1}^{d} P(X_j = x_j | C = c_i) \\ &= \arg\max_{i \in [k]} \log(P(C = c_i)) + \sum_{i=1}^{d} \log(P(X_j = x_j | C = c_i)) \end{aligned} \tag{2}$$

The final equality holds since taking the log does not affect the arg max of a sequence of values. The reason for doing is that taking the log turns a product into a summation, which allows us to use additive homomorphic encryption which we will go over in section 3.

2.3 Decision Trees

Suppose that we have a set of data points that look like this:

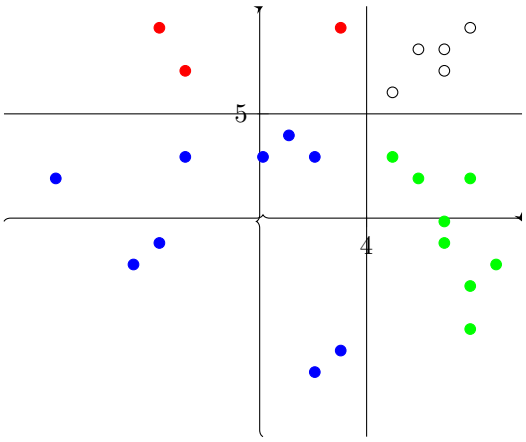

Figure 2: Graph of data points

We can see that we can separate the data by a set of boundaries. In this case, the lines $y = 5$ and $x = 4$ create a boundary within which points of the same color reside. Therefore, we can classify points based on a series of decision rules based on the boundaries we constructed. These decision rules can be formed as a binary tree as follows:

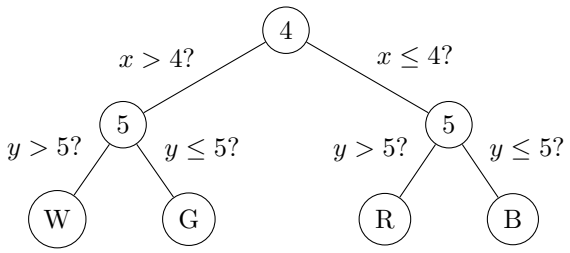

Figure 3: Binary tree representation of Figure 2

Given a data point (x, y), we can classify what color it belongs by traversing the binary tree. We traverse left if the input value is greater than the value at the node and right otherwise. We stop our traversal when we hit a leaf. The leaf nodes consist of the different classes that a data point can belong to. In this case, the classes are the different colors a dot might be. The values in the inner nodes which we compare the inputs to are called the threshold values.

3 Building Blocks

This section, we will go over some of the basic cryptographic building blocks that will be fundamental for the privacy-preserving classification protocols the paper [1] introduces.

3.1 Threat Model

We first go over the threat model that the paper assumes when constructing these building blocks. The threat model assumes that the adversaries are honest-but-curious. This means that if a party A in the protocol is compromised by the adversary, then the adversary can only observe the information party A receives. The adversary cannot prevent the compromised party from following the protocol. This is also known as a passive adversary, and the goal of the adversary is to learn the private inputs of other parties.

3.2 Comparisons

The paper implements a comparison protocol that allows two parties to compare inputs. The protocol allows comparison between encrypted and unencrypted inputs, and is flexible in that either party can receive the final result of the comparison. The underlying mechanisms for comparison depends on the requirements of the system. For example, we use the protocol mentioned in [2], which uses garbled circuits, to compare unencrypted inputs , and we use the protocol mentioned in [3], which uses specialized homomorphic encryption, to compare encrypted inputs. From now on, when we mention a comparison protocol, it will refer to the one mentioned here. We treat this comparison protocol as a black box.

3.3 Dot Products

From now on, to remain consistent with the paper, $[\![m]\!]$ refers to encrypting the integer m using the Paillier encryption scheme. Paillier is an asymmetric-key encryption scheme that has several useful properties, one of which is additive homomorphism. Having additive homomorphism means that:

$$[\![m_1]\!] \cdot [\![m_2]\!] = [\![m_1 + m_2]\!]$$

We can use this property to perform scalar multiplication for a scalar c:

$$[\![c * m]\!] = [\![m]\!]^c$$

. Now, suppose party 1 has vector $x \in \mathbb{R}^d$ and party 2 has vector $y \in \mathbb{R}^d$ and they want to compute the dot product of x and y securely. Note that the dot product between x and y is defined as $\sum_{i=1}^{d} x_i y_i$. We want to compute the dot product without either party knowing the other party's vector. We assume Party 1 has the Secret Key SK and Party 2 has the public key PK of party 1. To perform this dot product over encrypted inputs, we can perform the following protocol:

1. Party 1 encrypts every component of their vector x and sends it over to party 2. More concretely, they send $c_i = [\![x_i]\!]$ for $i = 1, 2, ..., d$ to party 2.

2. Party 2 computes
$$c_i^{y_i} = ([\![x_i]\!])^{y_i}$$
which by the property of additive homomorphic encryption, is equal to $[\![(x_i \cdot y_i)]\!]$

3. Party 2 now computes and outputs
$$\prod_{i=1}^{d} ([\![x_i]\!])^{y_i} = \prod_{i=1}^{d} [\![(x_i \cdot y_i)]\!]$$

which is equal to $[\![(\sum_{i=1}^{d} x_i y_i)]\!]$, precisely the dot product of x and y. Party 2 outputs this value and the protocol is concluded.

From now on, whenever we mention taking a dot product securely, it will refer to the protocol mentioned here. We treat this protocol as a black box.

3.4 Arg Max

Suppose we have two parties, Party A and Party B. Party A has k values encrypted which Party B can decrypt with his secret key K and party B wants to know the index of the largest value in the sequence. The requirement is that party B only learns the index of the largest value and nothing else. We will walk through two attempts of solving this problem.

3.4.1 First attempt of argmax

First, to prevent party B from learning about the ordering of the k values, party A performs a random permutation π over the data. We denote the k values in the permuted ordering as $[\![a'_1]\!], ..., [\![a'_k]\!]$ Then, one step of the protocol is as follows:

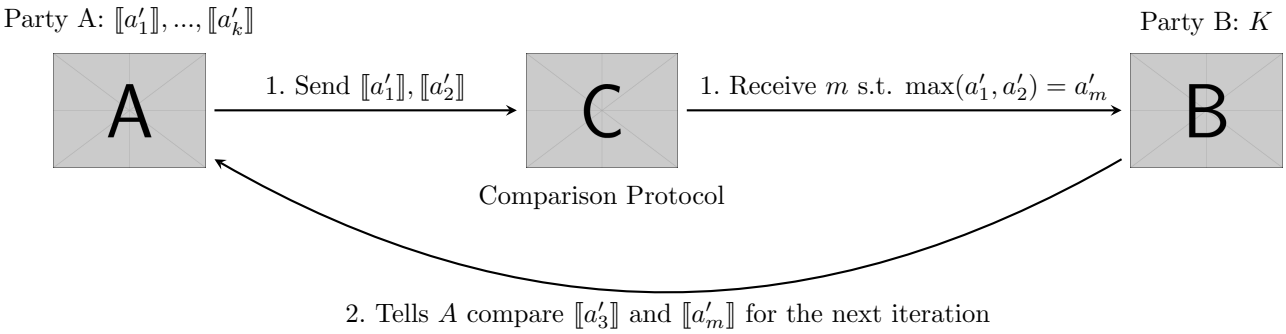

Figure 4: One iteration of the first attempt of argmax

More concretely, the first attempt of the argmax protocol is as follows:

1. A and B compare the first two values in the permuted sequence $[\![a'_1]\!]$ and $[\![a'_2]\!]$ using the comparison protocol described above. Party B learns the index m of the larger value.
2. Party B then tells A to compare $[\![a'_m]\!]$ and $[\![a'_3]\!]$ next.
3. Continuing this procedure by repeating steps 1 and 2 until the two parties loop through all the values, party B learns the index m^* (in the permuted order) of the largest value using only a linear number of scans through the sequence.
4. Once party B learns m^*, they can send m^* to party A.
5. Party A computes and outputs $\pi^{-1}(m^*)$, which is the index of the largest value in the original ordering.

However, in this protocol, Party A learns some partial information about the ordering of the values. This is because at each iteration, party B tells party A what values to compare next, thus allowing party A to know the relative ordering of the two values it initially compared with party B. Because party A has the permutation π, they can learn some partial information about the ordering of these values in the original

ordering. This violates the security guarantees as party A learns more than just the index of the largest value. One way to solve this is to compare every pair of inputs with B. This results in party A knowing nothing about the relative ordering of the elements since party B will request the same pair of values to compare next, regardless of the outcome of the comparison between the initial pair. However, this results in a quadratic number of comparisons, which is inefficient. To maintain the security guarantees and only use a linear number of orderings, we have to do something more clever.

3.4.2 Second attempt of argmax

The main issue we saw before was that Party A was able to obtain partial information about the ordering of the values. One step of the improved protocol is as follows:

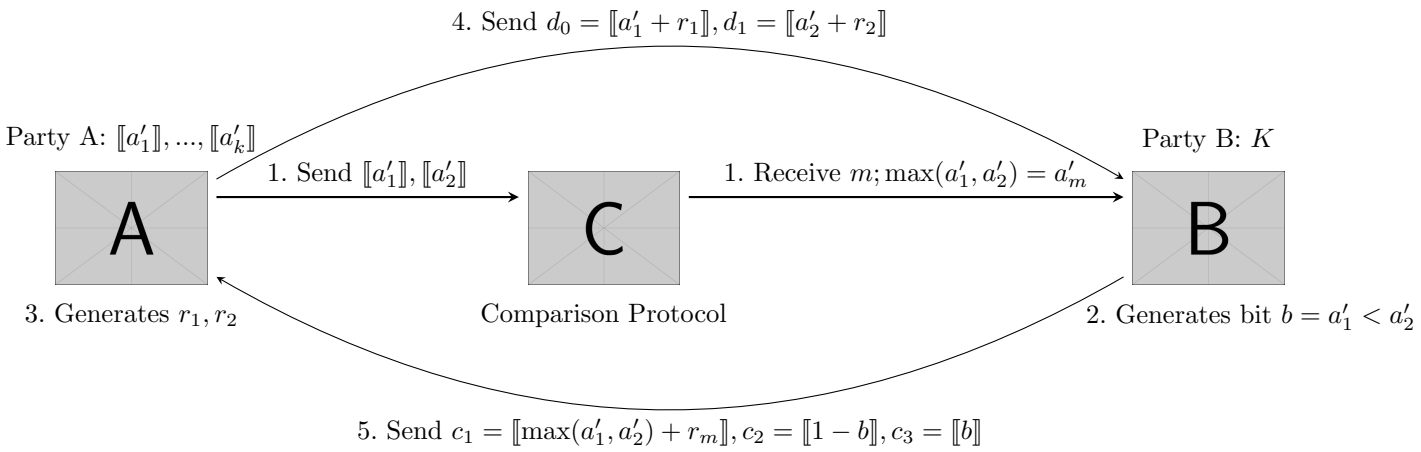

Figure 5: One iteration of the second attempt of argmax

To fix the issue of A learning the relative ordering the values, we proceed with the following protocol:

1. We have A and B run the comparison protocol on the first two values of the permuted ordering $[\![a'_1]\!]$ and $[\![a'_2]\!]$. Party B learns the index m that corresponds with the larger value $[\![a'_m]\!]$.
2. Party B generates a bit $b = a'_1 < a'_2$ that indicates which value is larger.
3. Party A generates 2 random numbers r_1 and r_2.
4. Party A then computes the following and sends it to party B:
   - $d_1 = [\![a'_1]\!] \cdot [\![r_1]\!] = [\![a'_1 + r_1]\!]$
   - $d_2 = [\![a'_2]\!] \cdot [\![r_2]\!] = [\![a'_2 + r_2]\!]$
5. Party B then computes the following and sends it to party A:
   - $c_1 = [\![\max(a'_1, a'_2) + r_m]\!]$ back to party A
   - $c_2 = [\![1 - b]\!]$
   - $c_3 = [\![b]\!]$
6. Once party A receives the 3 ciphertexts, they compute:
   $$f = c_1 \cdot c_2^{-r_1} \cdot c_3^{-r_2}$$
   where r_1 and r_2 are the random values party A previously generated.

7. f and $[\![a_3']\!]$ are the two new values party A and party B compare. The above is then repeated k times for the k values party A possesses.

8. Party B sends m^*, the index of the largest value in the permuted ordering, to party A.

9. Party A computes and outputs $\pi^{-1}(m^*)$

Let us first focus on this equation

$$c_1 \cdot c_2^{-r_1} \cdot c_3^{-r_2}$$

and see why this works. If $b = 0$, then

1. $\max(a_1', a_2') = a_1'$
2. $c_1 = [\![a_1' + r_1]\!]$
3. $c_2 = [\![1]\!]$
4. $c_3 = [\![0]\!]$

We can therefore see that

$$\begin{aligned} c_1 \cdot c_2^{r_1} \cdot c_3^{r_2} &= [\![a_1' + r_1]\!] \cdot ([\![1]\!])^{r_1} \cdot ([\![0]\!])^{r_2} \\ &= [\![a_1' + r_1]\!] \cdot [\![-r_1 * 1]\!] \cdot [\![-r_2 * 0]\!] \\ &= [\![a_1' + r_1 - r_1 - 0]\!] \\ &= [\![a_1']\!] \end{aligned} \tag{3}$$

If $b = 1$, similar computation can be done to show that party A receives $[\![a_2]\!]$. As we can see, at each iteration party A computes the encryption of the larger value and thus learns nothing about the ordering. This is because they do not know which value is encrypted inside the ciphertext f they compute. Thus, they do not learn any information of the relative ordering of the elements. Although party B can decrypt the ciphertexts d_0, d_1, they only learn $a_1' + r_1$ and $a_2' + r_2$. Because r_1 and r_2 are random values, party B learns nothing about the original values in the sequence. Therefore, the second attempt of the argmax protocol provides the security guarantee that party A and B learn nothing except for the index of the largest value, while still maintaining efficiency that is linear in the number of values. Whenever we refer to using an argmax protocol, it will refer to the one mentioned here. We treat this protocol as a black box.

3.5 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE for short) is a type of encryption scheme that is more powerful than additive homomorphic encryption as FHE allows us to compute arbitrary functions on ciphertexts. From now on, $[m]$ will be the notation for encrypting a message m using FHE. Given the ciphertext $[m]$, encryption key pk and function f, $\text{Eval}(pk, f, [m])$ outputs $[f(m)]$. This powerful functionality comes at the cost of performance as traditional FHE schemes are often quite slow. There are a few techniques that help optimize FHE mentioned in the paper as follows:

1. Using a leveled FHE scheme as described in [4]
2. Using FHE SIMD slots.
3. Using $\mathbb{F}_2$ as the plaintext space.

First, leveled FHE schemes attempt to limit the depth of the underlying function circuit. Leveled FHE schemes such as HElib mentioned in [4] support only a fixed multiplicative depth instead of traditional FHE schemes which support arbitrary depth circuits. Leveled FHE schemes have shown to be empirically much faster than traditional FHE schemes when the depth of the circuit is small. FHE SIMD (Single instruction

multiple data) Slots allow us to pack multiple messages into a single ciphertext to increase efficiency. We can therefore do the following with FHE SIMD Slots:

$$c = \text{Enc}(m_1, m_2, ..., m_n)$$

$$\text{Eval}(c, f, pk) \rightarrow \text{Enc}(f(m_1), f(m_2), ..., f(m_n))$$

Finally, using $\mathbb{F}_2$ as the plaintext space has been shown to offer speedups for traditional FHE schemes as well. This means that the values which are encrypted are bits. These techniques will be used later in order to increase the performance of privacy-preserving decision trees.

4 Privacy-Preserving Classification

We will now give a high-level overview for each privacy-preserving classification protocol.

4.1 Privacy-Preserving Hyperplane Decision

As mentioned previously, the protocol revolves around taking the argmax of several inner products. Therefore, to perform this protocol, we take in a model w from the server which consists of k vectors $\{w_i\}_{i=1}^k$, where each $w_i \in \mathbb{R}^d$. We also take in as input the desired vector of features $x \in \mathbb{R}^d$ from the client. The protocol for privacy-preserving hyperplane decision is as follows:

1. Run the secure dot product protocol between the server and client to compute the dot product of w_i and x for $i = 1, 2, ..., k$. The client receives $\{[\![v_i]\!]\}_{i=1}^k$ where $v_i = \langle w_i, x \rangle$
2. Run the secure argmax protocol in the client is party A with the k encrypted values $\{[\![v_i]\!]\}_{i=1}^k$ and the server is party B. From the secure argmax protocol, the client receives the index i of the largest value of the sequence.

4.2 Privacy-Preserving Naïve Bayes

As mentioned previously, the protocol revolves around taking the argmax of conditional probabilities. The Naïve Bayes classifier can be performed using the building blocks previously mentioned. The privacy-preserving Naïve Bayes protocol is as follows:

1. The server prepares the probabilities mentioned previously in the Naïve Bayes protocol. The first is

   $$P(i) = \log(P(C = c_i))i = 1, ..., k$$

   : the probabilities that each class appears. The second is

   $$T_{ij} = \{\{\{\log(P(X_j = v|C = c_i))\}_{v \in D_j}\}_{j=1}^d\}_{i=1}^k$$

   : the probabilities that a component of X takes on a certain value v in its domain given that X is a part of class c_i.
2. The server sends these probabilities to the client, encrypted.
3. For $i = 1, 2, ..., k$, the client computes

   $$[\![p_i]\!] = [\![P(i)]\!] \prod_{j=1}^{d} [\![T_{ij}]\!] = \log(P(C = c_i)) + \sum_{j=1}^{d} \log(P(X_j = v|C = c_i))$$

4. The client has values $[\![p_i]\!]$ for $i = 1, ..., k$ and then runs the secure argmax protocol with the server and obtains the index of the largest value.
5. The client outputs the index, indicating the classification class of the input vector.

4.3 Privacy-Preserving Decision Trees

The key to privacy-preserving decision trees is that we can represent a decision tree as a polynomial function P. For example, consider the following decision tree:

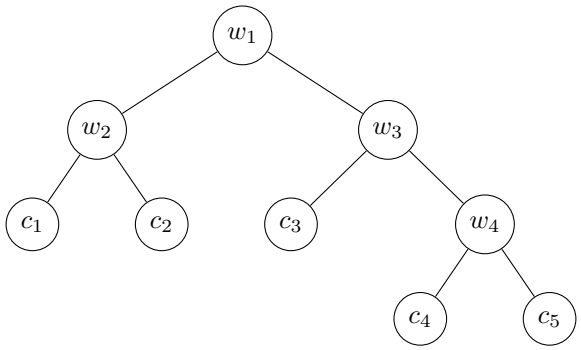

Figure 6: Decision tree example used in [1]

Here c_1 through c_5 are the classification classes. The w_i's are the threshold values we use in the decision tree. We define b_i be the comparison bit between the input and the threshold value w_i. For example, $b_1 = 1$ if $x_1 \leq w_1$ and 0 otherwise. The polynomial function P we can construct for the tree is as follows:

$$P(b_1, ..., b_4, c_1, ..., c_5) = b_1(b_3 \cdot (b_4 \cdot c_5 + (1 - b_4) \cdot c_4) + (1 - b_3) \cdot c_3) + (1 - b_1)(b_2 \cdot c_2 + (1 - b_2) \cdot c_1) \quad (4)$$

Therefore, given a sequence of comparison bits b_1, b_2, b_3, b_4 the polynomial function P outputs the corresponding classification class. These comparison bits can be obtained by running the secure comparison protocol mentioned above in section 3.2. Therefore, now we have the protocol for privacy-preserving decision trees. Here, the server has the decision tree and to remain consistent with the paper, the client has the input feature vector in $\mathbb{R}^n$.

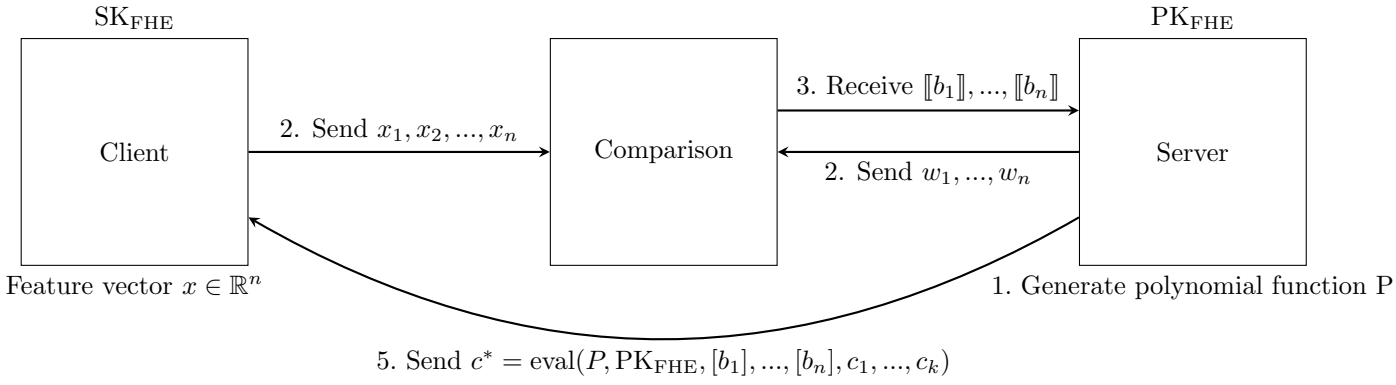

Figure 7: Secure Decision Tree Protocol

Note that step 4 is missing in the figure. It is not important for the high-level overview of the protocol, but we include it for completeness. We proceed with the privacy-preserving decision tree protocol as follows. In the start of the protocol, the server has the public key for FHE: PK_{FHE} and the client has the secret key for FHE: SK_{FHE}.

1. The server generates a polynomial function P such as the one shown above that encodes the decision tree.
2. The server and client run the comparison protocol, comparing each component of the client's input vector x_i with the corresponding threshold value w_i.
3. From the comparison protocol, the server obtains $[\![b_i]\!]$. for $i = 1, ..., n$.
4. The server then runs a protocol to change the encryption of these b_i's to ciphertexts encrypted under a FHE scheme.
5. The server applies the eval function on P, the bits b_1 to b_n, encrypted under a FHE scheme, and the classes c_1 to c_k. It sends back the output

$$c^* = \text{eval}(P, \text{PK}_{\text{FHE}}, [b_1], ..., [b_n], c_1, ..., c_k)$$

where c^* is the outcome class, to the client.

There are a few speedups discussed in section 3.5 that can make the traditionally slow FHE more efficient and practical in real-world settings.

1. We can use a leveled FHE scheme which is empirically much faster than traditional FHE for circuits with small depth. When evaluating the polynomial function P, we can reduce the depth of the circuit by performing multiplications in a tree-like manner rather than in a linear manner.
2. We can use $\mathbb{F}_2$ or mod 2 as the plaintext space. To do this, we have to represent each class as a sequence of bits. If there are k classes, then $l = \log_2 k$ bits are be needed to represent all the classes.

If we use $\mathbb{F}_2$ as the plaintext space, we must then evaluate the polynomial function P l times, once for every bit of the outcome class. More concretely, on the jth evaluation, we call:

$$\text{eval}(P, \text{PK}_{\text{FHE}}, [\![b_1]\!], ..., [\![b_n]\!], c_{1j}, c_{2j}, ..., c_{kj})$$

where c_{ij} is the notation we use to indicate jth bit of the ith classification class. To avoid performing l multiplications, we use FHE SIMD Slots as described above in section 3.5. To do this, for each class c_i, we create the ciphertext $[c_{i0}, ..., c_{il-1}]$ which consists of all the bits of c_i. For each bit b_i, we can create the FHE ciphertext $[b_i, ..., b_i]$, essentially repeating b_i l times in the ciphertext. Thus, we can take advantage of FHE SIMD Slots and run the eval function on P and all of the aforementioned ciphertexts. Assuming we have k classification classes, this is equivalent to computing:

$$[P(b_1, ..., b_n, c_{10}, ..., c_{k0}), P(b_1, ..., b_n, c_{11}, ..., c_{k1}), ..., P(b_1, ..., b_n, c_{1l-1}, ..., c_{kl-1})]$$

This is equal to $[c_{o0}, ..., c_{ol-1}]$ where c_o is the outcome class. The client can decrypt the ciphertext and reconstruct the bits to get the final value of the outcome class. Therefore, with these performance enhancements, we can perform the protocol for privacy-preserving decision trees efficiently.

References

[1] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[2] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 478–492, 2013.

[3] Thijs Veugen. Comparing encrypted data. Unpublished, 2011.

[4] Shai Halevi and Victor Shoup. Algorithms in HElib. In *CRYPTO (1)*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2014.