Patrick Schwarz, BSc

Privacy Preserving Machine Learning

An implementation using fully homomorphic encryption in HElib

Master's Thesis

to achieve the university degree of

Diplomingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl-Ing. Dr.techn. Christian Rechberger,
Dipl.-Ing. BSc Lukas Helminger,
Dipl.-Ing. BSc Roman Walch

Institute of Applied Information Processing and Communications

Graz, 2020

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Datum	Unterschrift

Abstract

Today Big Data and Machine Learning enable us to find new information within data collected over the years. Many try to collect as much private information of users as possible to make even more precise predictions. Correct anticipation of customer behaviour or any future event can be a great advantage. But even with a lot of data, a company might not have the computational power or knowledge to create an accurate model for predictions. In this case, a second party, like a cloud service, may provide the missing components, required for training such a model.
However, a cloud service customer might not want to share valuable information with the service provider to keep a potential advantage in the industry. In many cases, the General Data Protection Regulation (GDPR) even prohibits the propagation of personal data.
In this thesis, we developed a solution to this problem. We provide an implementation of a Random Forest classifier that uses fully homomorphic encryption (FHE) to train on encrypted data, protecting confidentiality. Therefore, the customer profits from the knowledge and performance of the cloud service without revealing any information.
Contrary to the state-of-the-art, the entire generation of models requires only one initial transmission of parameters and encrypted training data. Afterward, the whole process requires no participation of the customer at all, which also reduces the required infrastructure. In all schemes developed up until now, the customer must stay on-line during the whole procedure.
In summary, we may say that our scheme allows customers to train a model on a cloud service without the necessity of sharing the private information with anyone.

Kurzfassung

Heutzutage erlauben uns große Datenmengen und Machine Learning neue Informationen zu finden. Deshalb versuchen Firmen immer mehr private Informationen über User zu sammeln, um deren Voraussagen in die Zukunft zu verbessern. Die Möglichkeit zukünftige Events präzise vorauszusagen ist zweifelsohne ein gewaltiger Vorteil. Jedoch reicht eine Vielzahl an Daten oft nicht aus. Um neue Information extrahieren zu können werden Hintergrundwissen bezüglich Lernalgorithmen als auch Rechenleistung benötigt. In diesem Fall wird oft auf Cloud-Dienste zurückgegriffen, welche die benötigten Komponenten liefern.
Nichts desto trotz möchte ein Kunde des Dienstes verhindern, Daten mit diesem zu teilen, um sich einen bestehenden Vorteil am Markt zu erhalten. In vielen Fällen verbietet die Datenschutzgrundverordnung (DSGVO) [1] sogar die Weitergabe von benutzerbezogenen Daten. Mit dieser Arbeit wollen wir eine von uns entwickelte Lösung vorstellen. Wir präsentieren einen Random Forest Lernalgorithmus, der mittels Fully Homomorphic Encryption (FHE) ein Modell anhand verschlüsselter Daten erstellt. Die Verschlüsselung sorgt dafür, dass der Dienst keine relevanten Informationen erhält, solange er nicht den geheimen Schlüssel besitzt. Somit profitiert der Kunde von dem Service, ohne seine Daten offenzulegen.
Im Gegensatz zum neuesten Stand der Technik benötigt das Erzeugen eines Models nur eine einzige Interkation zwischen Kunde und Anbieter. Nach dem initialen Austausch der Daten benötigt es also keine weitere Kommunikation zwischen beiden Partnern. Dies führt zu einer Reduktion der benötigten Infrastruktur am Kunden. In allen zuvor entwickelten Verfahren war es bisher nicht möglich das Anlernen, ohne die Hilfe des Kunden durchzuführen.
Abschließend bleibt festzuhalten, dass unser Verfahren dem Kunden ermöglicht ein Modell mit dem Wissen und der Performance des Cloudservices zu erstellen, ohne private Informationen mit dem Anbieter zu teilen.

Contents

1 Introduction

With Machine Learning, we can discover hidden information within data collected over the years. The possibility to anticipate user behavior or to predict any future event turns out to be a great advantage.
To perform learning algorithms on immense amounts of data, we require a lot of computational power. Nowadays, many companies turn away from the idea to harvest a lot of processing capabilities within their organization. Cloud services enable us to request computational power on demand and just for the time needed. One big problem with this idea is the confidentiality of our data. Most of the data we produce will get collected by different entities, who try to keep it for themselves to exploit its value. In many cases, the General Data Protection Regulation (GDPR) [1] even prohibits the sharing of personal data with third parties.
While the infrastructure in companies is shrinking, their efforts in developing new and even more precise learning algorithms grows. Therefore, in this thesis we aim to protect not only the data but also the algorithms used to extract further information. Our training algorithm allows two parties to build a model on encrypted data without the necessity of leaking any knowledge or information to the other party.
Figure 1.1 describes a simple communication model between our two parties. The customer, owning the data, wants to train a model to predict future events. The other party, a cloud service, consists of a server cluster with high computational power. Furthermore, the cloud service knows how to create a model suited for the data provided by the client.

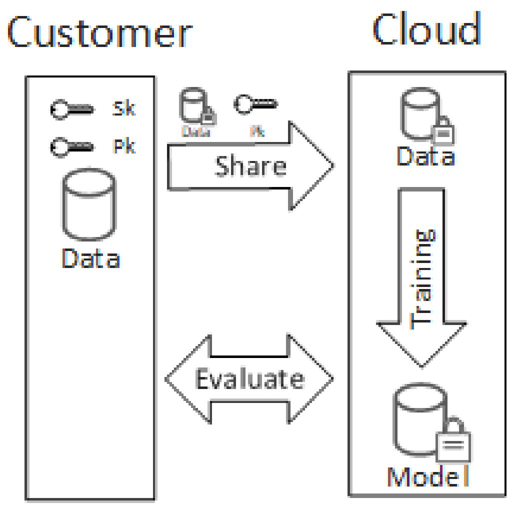

Figure 1.1: Communication model between customer and cloud.

The customer sends his encrypted training set to the cloud service. Since the cloud trains on encrypted data, the resulting model is encrypted as well. Now we can decide if we want the model placed at the cloud or sent back to the customer. If it stays at the cloud, the customer can evaluate new samples by encrypting and sending them to the cloud. Then the computational rich entity evaluates and sends the encrypted result back to the customer, who then decrypts it. In the other case, the customer receives the whole model, which he than decrypts for further use.
In both cases, the training data is kept confidential from the cloud. If the model remains in the hand of the cloud service, we can also ensure that the client will not learn anything about the training procedure used by the service.
In this thesis, we use fully homomorphic encryption (FHE) [2] to create a learning algorithm that trains on encrypted data. Our C++ implementation is based on a Random Forest Classifier [3] and uses the HElib [4] framework for FHE support. The scheme is highly parallelizable due to the multi-threading capabilities of HElib as well as some custom optimizations. A Random Forest consists of a variety of Decision Trees [5]. We train each

tree with a randomly sampled subset of the original data. Therefore, we decided to concentrate on the implementation of a Decision Tree training algorithm, which can then be extended to a Random Forest classifier.

1.0.1 Related Work

For the construction of a Decision Tree on encrypted data, not much prior work can be found, except for a work by Akavia et al. [6]. They present a procedure which uses the CKKS [7] fully homomorphic encryption scheme to homomorphically train a Decision Tree.
Their setting also consists of two parties, the customer owning the training data and high performant cloud service. While in our contribution, the communication ends after the initial transmission of parameters and training data, their procedure requires that the client stays on-line. In their scheme, the client needs to decrypt intermediate values, multiply them together, and return the encrypted results. For each node of the Decision Tree, server and client exchange and compute values, which leads to an intensive data transfer. However, the calculations in plain as well as the fresh encryption of intermediate results, lead to a significant performance speedup.

2 Fully Homomorphic Encryption

This chapter explains the principles of homomorphic encryption. Furthermore, we will explain the BGV [8] scheme which we use in the implementation of the learning algorithm in this thesis.
The idea of homomorphic encryption is to allow computations on encrypted data without knowing the private key. An encryption is homomorphic if we are able to compute $f(Enc(A), Enc(B)) = Enc(f(A, B))$ without using the private key. The function f can be composed of multiple operations, most notably, addition and multiplication. Most of the schemes support addition, multiplication, or both.
First of all, we want to explain the differences between partially, somewhat, and fully homomorphic encryption schemes.

2.1 Partially Homomorphic Encryption

Partially homomorphic encryptions allow us to perform only one homomorphic operation. For additive homomorphic encryptions it holds that $Enc(A) + Enc(B) = Enc(A + B)$, for multiplicative homomorphic encryptions $Enc(A) \times Enc(B) = Enc(A \times B)$.
As an example, we demonstrate an additive, and an multiplicative homomorphic encryption scheme, both based on ElGamal [9]. The encryption scheme consists of a cyclic group $\mathbb{G}$ of order q with generator g. The secret key of the scheme is x. The public key would be $(\mathbb{G}, q, g, h)$, where $h = g^x$, which is called shared secret. To perform an encryption of a message m we do

$$\begin{aligned} s &= h^r \\ c &= \{c_1, c_2\} = \{g^r, s \cdot m\}, \end{aligned}$$

where r is chosen uniformly at random from the set $\{1, ..., q-1\}$. To decrypt the ciphertext, we compute

$$c_1^{-x} \cdot c_2 = (g^r)^{-x} \cdot g^{xr} \cdot m = g^{-xr+xr} \cdot m = g^0 \cdot m = 1 \cdot m = m.$$

Now, consider two messages m_1, m_2 encrypted under the same secret key x. Under this encryption it is possible to multiply two cihpertext like this

$$\begin{aligned} E(m_1) \cdot E(m_2) &= (g^{r_1}, m_1 \cdot h^{r_1})(g^{r_2}, m_2 \cdot h^{r_2}) \\ &= (g^{r_1+r_2}, (m_1 \cdot m_2)h^{r_1+r_2}) \\ &= (g^{r_3}, (m_1 \cdot m_2)h^{r_3}) \\ &= E(m_1 \cdot m_2), \end{aligned}$$

where r_3 is the substitution of $r_1 + r_2$.
Now, in order to make an additive homomorphic encryption scheme, we change the encryption in the following way

$$\begin{aligned} s &= h^r \\ c &= \{c_1, c_2\} = \{g^r, g^m \cdot s\}. \end{aligned}$$

We now get

$$\begin{aligned} E(m_1) \cdot E(m_2) &= (g^{r_1}, g^{m_1} \cdot h^{r_1})(g^{r_2}, g^{m_2} \cdot h^{r_2}) \\ &= (g^{r_1+r_2}, g^{m_1+m_2}h^{r_1+r_2}) \\ &= (g^{r_3}, g^{m_1+m_2}h^{r_3}) \\ &= E(m_1 + m_2). \end{aligned}$$

2.2 Somewhat Homomorphic Encryption

To create an even more powerful homomorphism we talk about somewhat homomorphic encryption (SHE). This scheme supports both addition and multiplication, but the number of operations we can chain together is limited. In order to use a somewhat homomorphic encryption scheme we have to know the number of consecutive operations beforehand to choose a suitable parameter set.

2.3 Fully Homomorphic Encryption

An even more powerful HE variant is fully homomorphic encryption (FHE), presented first by Gentry in 2009 [2]. Fully homomorphic encryption schemes can perform addition and multiplication arbitrary times. Therefore we can construct a combination of operations of arbitrary length without knowing the depth of the so called circuit beforehand.
All of the different types have advantages and disadvantages. While the support of two operations within one scheme is rather complex, it opens up new paths. In the field $\mathbb{Z}_2$ an addition is equivalent to an XOR and a multiplication is realized by the AND operation. Thus, an FHE scheme supporting the field $\mathbb{Z}_2$ as plaintext space allows us to perform logical operations homomorphically. By combining these two operations with the rules provided by [10], we can generate a punch of different logic gates, as we describe in Table 2.1.

Gate	Example	Substitution
NOT	NOT A	A XOR 1
OR	A OR B	((A XOR 1) AND (B XOR 1)) XOR 1
NOR	A NOR B	(A XOR 1) AND (B XOR 1)
XNOR	A XNOR B	A XOR B XOR 1
NAND	A NAND B	(A AND B) XOR 1

Table 2.1: All logic gates from $\{XOR, AND\}$.

Especially NAND-Gates are of great importance, because one may construct any logic boolean-circuit (i.e. logical compositions of gates with one boolean output) with just this one gate. NAND-Gates are also functionally complete [11]. This means that any given program can be translated into a combination of those gates. In summary, it is possible to evaluate any program of any length on encrypted data in an FHE-scheme.

2.4 Notation Homomorphic Encryption

We write vectors in bold, e.g., $\boldsymbol{v} \in \mathbb{R}^n$. The notation v_i refers to the i-th element of $\boldsymbol{v}$. Sometimes we will abuse this notation by saying $v_{i,a} \in V$ to refer to the a-th element of the i-th row vector in the matrix V. Matrices will be written in capital letters.
Furthermore, we use the polynomial ring $R = \mathbb{Z}[x]/(x^d + 1)$ where d is a power of two and $R_q = R/qR$ reduces all coefficients of the polynomial mod q.
The scalar product of two vectors $\boldsymbol{a}, \boldsymbol{b}$ is written as $\langle \boldsymbol{a}, \boldsymbol{b} \rangle$.

2.5 Learning With Errors

Today's FHE schemes are based on the hardness assumption of the Learning with Errors problem, short LWE-problem. The LWE-hardness assumption is the claim that the LWE search problem is hard to solve. As stated in [12], the LWE search problem is to find an unknown $\boldsymbol{s} \in \mathbb{Z}_2^n$ given a list of arbitrary noisy equations like

$$\begin{aligned} \langle \boldsymbol{a_1}, \boldsymbol{s} \rangle + \epsilon_1 &= b_1 \in \mathbb{Z}_2 \\ \langle \boldsymbol{a_2}, \boldsymbol{s} \rangle + \epsilon_2 &= b_2 \in \mathbb{Z}_2 \\ &\vdots \end{aligned}$$

where $n \geq 1$, $\epsilon_i \geq 0$ and $\boldsymbol{a_i}$ is chosen uniformly at random from $\mathbb{Z}_2^n$ for all i. Each equation is correct independently with probability $1 - \epsilon_i$. Notice, that if $\epsilon_i = 0$ the solution to the problem can be found efficiently by Gaussian elimination. This requires $\mathcal{O}(n)$ equations to be a determined system and can be solved in polynomial time.
The solving of the problem becomes significantly harder if we add a small error ϵ of the Gaussian distribution $\mathcal{X} = D_{\mathbb{Z},\sigma}$ as described in [13] with σ denoting the standard deviation and $\mathbb{Z}$ result space. The best known algorithm to solve this problem was presented by Blum, Kalai, and Wasserman [14]. The algorithm requires $2^{\mathcal{O}(n/\log n)}$ equations and time.
A natural extension to the problem is the definition of higher moduli. By

choosing $q = q(n) \leq poly(n)$ as a prime the problem becomes finding $\boldsymbol{s} \in Z_q^n$ from a arbitrary list of equations

$$\begin{gathered} \langle \boldsymbol{a_1}, \boldsymbol{s} \rangle + \epsilon_1 = b_1 \in \mathbb{Z}_q \\ \langle \boldsymbol{a_2}, \boldsymbol{s} \rangle + \epsilon_2 = b_2 \in \mathbb{Z}_q \\ \vdots \end{gathered}$$

where the $\boldsymbol{a_i}$'s are chosen independently from $\mathbb{Z}_q^n$ and the errors ϵ_i are sampled from $\mathcal{X}$. In this scenario the algorithm from Blum et al. requires $2^{\mathcal{O}(n)}$ equations and time.
In Figure 2.1 we can see an example of the LWE search problem with sampled data. In this example we consider $q = 13$, $n = 4$ and $m = 7$. The matrix A contains all $\boldsymbol{a_i}$'s, vector $\boldsymbol{b}$ all b_i's and vector $\boldsymbol{\varepsilon}$ all ϵ_i's.

$$\underset{A \epsilon \mathbb{Z}_{13}^{7\times4}}{\begin{bmatrix} 4 & 1 & 11 & 10 \\ 5 & 5 & 9 & 5 \\ 3 & 9 & 0 & 10 \\ 1 & 3 & 3 & 2 \\ 12 & 7 & 3 & 4 \\ 6 & 5 & 11 & 4 \\ 3 & 3 & 5 & 0 \end{bmatrix}} \text{ x } \underset{\boldsymbol{s} \epsilon \mathbb{Z}_{13}^{4\times1}}{\begin{bmatrix} 6 \\ 9 \\ 11 \\ 11 \end{bmatrix}} + \underset{\boldsymbol{\varepsilon} \epsilon \mathbb{Z}_{13}^{7\times1}}{\begin{bmatrix} 0 \\ -1 \\ 1 \\ 1 \\ 1 \\ 0 \\ -1 \end{bmatrix}} = \underset{\boldsymbol{b} \epsilon \mathbb{Z}_{13}^{7\times1}}{\begin{bmatrix} 4 \\ 7 \\ 2 \\ 11 \\ 5 \\ 12 \\ 8 \end{bmatrix}}$$

Figure 2.1: LWE-problem example.

In addition, we want to demonstrate how we can build an encryption scheme based on the LWE hardness assumption, as described in [12]. First we choose $\boldsymbol{s} \in \mathbb{Z}_q^n$ uniformly at random as the private key. Afterwards, we sample m vectors $\boldsymbol{a_1}, ..., \boldsymbol{a_m} \in \mathbb{Z}_q^n$ and m errors $\epsilon_1, ..., \epsilon_m$ sampled from $\mathcal{X}$.

The public key consists of the matrix $A = \{\boldsymbol{a_i}, ..., \boldsymbol{a_m}\} \in \mathbb{Z}_q^{n \times m}$ and a vector $\boldsymbol{b} = \{b_1, ..., b_m\} \in \mathbb{Z}_q^m$ where $b_i = \langle \boldsymbol{a_i}, \boldsymbol{s} \rangle + e_i \in \mathbb{Z}_q$. To encrypt a one bit message $M \in \{0, 1\}$ we generate a random vector $\boldsymbol{r} \in \mathbb{Z}_2^m$ and calculate

$$\begin{aligned} \boldsymbol{u} &= \sum_{i=1}^{m} r_i \cdot \boldsymbol{a_i} \in \mathbb{Z}_q^n \\ v &= \frac{q}{2} \cdot M + \sum_{i=1}^{m} r_i \cdot b_i \in \mathbb{Z}_q, \end{aligned}$$

where $\{\boldsymbol{u}, v\}$ is our encrypted ciphertext. In order to decrypt our ciphertext, we calculate

$$c = v - \langle \boldsymbol{u}, \boldsymbol{s} \rangle \in \mathbb{Z}_q.$$

If the intermediate result c is greater $\frac{q}{2}$ then the message was 0, otherwise it was 1. The downside of this scheme is that we require a lot of equations to encrypt just one bit of information.
Now, that we have explained the basics of the hardness assumption we must refer to the disadvantage for schemes based on it. When we start to use the homomorphic properties of the scheme, by adding and multiplying ciphertexts, we will notice that the error ϵ in the ciphertexts grow with each operation. At some point the error will make a decryption of the ciphertext impossible. This is true for all types of LWE based encryption schemes.

2.5.1 Ring Learning With Errors

Next we will explain Ring-LWE [13], short RLWE, which extends LWE by sampling the variables of the ring R. Let $R = \mathbb{Z}[x]/f(x)$ be the ring of integer polynomials modulo $f(x) = x^n + 1$, with n as a power of 2 to make it irreducible over the rationals. An irreducible polynomial is a polynomial that cannot be factored into the product of two sub-polynomials. Let q be a prime modulus fulfilling $q = 1 \mod 2n$. Then $R_q = R/qR = \mathbb{Z}_q[x]/f(x)$

describes the the ring of integer polynomials modulo $f(x)$ with coefficients in $\mathbb{Z}_q$. This ensures, that each coefficient cannot be greater than q and the highest degree of the polynomials would be $n-1$. The RLWE search problem is to find the coefficients of the ring polynomial $s \in R_q$ given the scalar product

$$s \cdot a + \epsilon = b \in R_q$$

where the a is chosen independently from R_q. Each coefficient is inflicted with an error ϵ from the Gaussian distribution $\mathcal{X} = D_{\mathbb{Z}^n,\sigma}$ as described in [13], where σ denotes the standard deviation and $\mathbb{Z}^n$ describes the result space.
Figure 2.2 shows an example of the RLWE search problem with sampled data. In this example the ring is defined as $R_{13} = \mathbb{Z}_{13}[x] / \langle x^4 + 1 \rangle$.

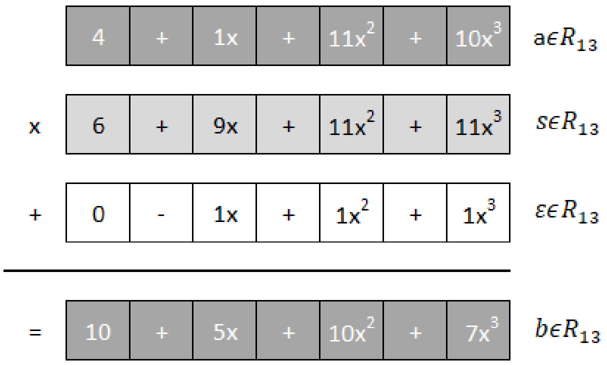

Figure 2.2: RLWE-problem example.

Cryptographic schemes based on the RLWE perform much better than the ones based on LWE. In [12] they state, that the reason for this performance increase is the smaller key size. The size of public and private keys under RLWE is roughly the square root compared to LWE with the same security level. Furthermore, the difference in size influences the performance of the

schemes drastically.
Finally, we want to give an example of an RLWE-Scheme as described in [15] and [16]. Lets assume we have Ring $R_q = \mathbb{Z}_q[x]/\Phi_m(x)$ and elements sampled from $\mathcal{X}$ are polynomials with small coefficients because $\sigma << q$. The positive parameter $m \in \mathbb{Z}$ defines $\Phi_m(x)$, where $\Phi_m(x)$ is the m-th polynomial with degree $n = \phi(m)$ with *phi* describing the Euler's totient function. First, we sample the secret key s uniformly from the distribution $\mathcal{X}$ denoted by $s \leftarrow \mathcal{X}$. Next we sample the error $e \leftarrow \mathcal{X}$, the polynomial $a \in R_q$ and set the polynomial b to $b = -(a \cdot s + e \cdot p) \in R_q$, where p defines our plaintext space R_p. Then we can use (a, b) as the public key. In order to encrypt a message $m \in R_p$, where $q \gg p$, we do

$$\begin{aligned} &u, f, g \leftarrow \mathcal{X}, \quad \text{i.e.} \quad u, f, g \in R_q \\ &ct = (c_0, c_1) = (b \cdot u + g \cdot p + m, a \cdot u + f \cdot p) \in R_q^2. \end{aligned}$$

A freshly encrypted ciphertext consists of two polynomials c_0 , c_1. However, the size of the ciphertext may increase due to multiplication, which we describe later in this section. We decrypt the ciphertext $ct = (c_0, ..., c_k)$ of variable length k as

$$m = \left[[\langle c, s' \rangle]_q \right]_p \in R_p$$

where the secret key vector s' is constructed as $s' = (1, s, s^2, ..., s^k)$.
Next we talk about addition. Lets consider the two ciphertexts $ct = (c_0, c_1, ..., c_\delta)$ and $ct' = (c'_0, c'_1, ..., c'_\lambda)$. If $\delta \neq \lambda$ then the shorter cihpertext is padded with zeroes. Addition can be performed elementwise:

$$ct_{add} = (c_0 + c'_0, c_1 + c'_1, ..., c_{max(\delta,\lambda)} + c'_{max(\delta,\lambda)}) \in R_q^{max(\delta,\lambda)}.$$

For multiplication it is a little bit more complicated. At first we consider the simple example of two freshly encrypted ciphertexts $ct = (c_0, c_1)$ and $ct' = (c'_0, c'_1)$ for which multiplication is executed as

$$ct_{mul} = (c_0 \cdot c'_0, c_0 \cdot c'_1 + c_1 \cdot c'_0, c_1 \cdot c'_1) \in R_q^3.$$

To multiply arbitrary sized ciphertexts $ct = (c_0, c_1, ..., c_\delta)$ and $ct' = (c'_0, c'_1, ..., c'_\lambda)$ we do

$$\left(\sum_{i=0}^{\delta} c_i \cdot v^i \right) \cdot \left(\sum_{i=0}^{\lambda} c'_i \cdot v^i \right) \equiv \sum_{i=0}^{\delta+\lambda} \hat{c}_i \cdot v^i.$$

where v is treated as an unknown variable. Note that we do not pad the ciphertexts with zeros here. The output ciphertext is $ct_{mlt} = (\hat{c}_0, ..., \hat{c}_{\delta+\lambda})$. If both ciphertexts have the same size we can reduce it to the tensored product lenght $\binom{n+1}{2}$.
There are several ways to control the size of the ciphertext which we explain later in Subsection 2.7.1.

2.5.2 General Learning With Errors

Since the LWE and RLWE are syntactically the same, there is a more general approach to describe both hardness assumptions. In the general-LWE (GLWE), as stated in [8], $f(x)$ is chosen to be a polynomial for some m coprime with q. For security parameter λ, let $n = n(\lambda)$ be an integer dimension, let $f(x) = x^d + 1$ where $d = d(\lambda)$ is a power of 2, let $q = q(\lambda) \geq 2$ be a prime integer, let $R = \mathbb{Z}[x]/(f(x))$ and $R_q = R/qR$, and let $\mathcal{X} = \mathcal{X}(\lambda)$ be a distribution sampling polynomials with small coefficients limited by λ.
The GLWE search problem is to find the coefficients of the ring polynomials $s \in R_q$ given the scalar products

$$\begin{aligned} s \cdot a_1 + \epsilon_1 = b_1 &\in R_q \\ &\vdots \\ s \cdot a_n + \epsilon_n = b_n &\in R_q \end{aligned}$$

where the a_i's are chosen independently from R_q and the ϵ_i's are sampled from $\mathcal{X}$.
The definition makes it possible to describe both cases in which R describes either $\mathbb{Z}$ or a polynomial ring and different vector dimensions over those rings. Instantiating GLWE with $d = 1$ is equal to LWE while GLWE with $n = 1$ is equal to RLWE. In the first case, the polynomial would consist of only one coefficient, while the number of equations n must be $n > 1$. In the second case, we define a polynomial with a higher degree and define only one equation $n = 1$.

2.6 Bootstrapping

In 2009 Gentry succeeded in constructing the first fully homomorphic encryption scheme [2] by introducing his novel bootstrapping method, also called recryption in the literature.
In GLWE based homomorphic encryption schemes, the error introduced during encryption grows with each operation. At some point, the error becomes too large, and the decryption of the ciphertext fails. With bootstrapping it is possible to reset the noise to its freshly encrypted state as often as needed.
In order to bootstrap a ciphertext encrypted under public key pk_1, we encrypt it a second time under pk_2. Next, we use a homomorphic circuit that uses the secret key sk_1 encrypted under pk_2. The circuit decrypts the first encryption under pk_1, which results in a ciphertext encrypted under pk_2. For better understanding, the formula below describes the basic steps:

$$\begin{aligned} c_{pk_1} &= E_{pk_1}(m) \\ c_{pk_1,pk_2} &= E_{pk_2}(c_{pk_1}) \\ s_{pk_2} &= E_{pk_2}(sk_1) \\ c_{pk_2} &= \textsc{Evaluate}_{pk_2}(D_{pk_2}, c_{pk_1,pk_2}, s_{pk_2}), \end{aligned}$$

where D_{pk_2} is the decryption circuit under pk_2. Therefore D_{pk_2} requires s_{pk_2} which holds parts of the secret key sk_1 encrypted under pk_2. After bootstrapping, the error of the original encryption vanishes completely, and the final noise is composed of a fresh encryption under pk_2 and the noise added by evaluation of the decryption circuit of pk_1.
If we replace pk_2 with pk_1 in all formulas above, we will use the same key for bootstrapping and encryption. In this scenario, we encrypt the secret key with its public key. Schemes doing so require circular security, which is not proven secure until today. However, many schemes, like the one used in HElib [4] still assume circular security to avoid dealing with multiple public/secret key pairs.

2.7 BGV

The BGV [8] scheme is one of the successors of the first FHE scheme from Gentry in 2009 [2]. This fully homomorphic scheme bases its security on the GLWE hardness assumption, which we describe in section 2.5. Brakerski, Gentry, and Vaikuntanathan published the scheme in 2011. To manage the noise more efficiently, they invented a new feature called modulus switching, where they switch the ciphertext to a smaller modulus. The reduction of the modulus also reduces the relative noise in the ciphertext, even without evaluating the costly bootstrapping procedure.
Next, we want to explain the basic components of the scheme. All components used for the scheme belong to a ring $R_q = \mathbb{Z}_q/f(x)$, which we explain in section 2.5. In Figure 2.3, one can see a basic encryption scheme using the GLWE hardness assumption, which is essentially the original procedure from Gentry 2009 and serves as an example for the differences to the BGV-encryption.

- E.Setup($1^\lambda, 1^\mu, b$): Use the bit $b \in \{0,1\}$ to determine whether we are setting parameters for a LWEbased scheme (where $d = 1$) or a RLWE-based scheme (where $n = 1$). Choose a μ-bit modulus q and choose the other parameters ($d = d(\lambda, \mu, b)$, $n = n(\lambda, \mu, b)$, $N = \lceil (2n+1) \log q \rceil$, $\mathcal{X} = \mathcal{X}(\lambda, \mu, b)$) appropriately to ensure that the scheme is based on a GLWE instance that achieves 2^λ security against known attacks. Let $R = \mathbb{Z}[x]/(x^d + 1)$ and let $params = (q, d, n, N, \mathcal{X})$.
- E.SecretKeyGen($params$): Draw $\boldsymbol{s}' \leftarrow \mathcal{X}^n$. Set $sk = \boldsymbol{s} \leftarrow (1, s'[1], ..., s'[n]) \in R_q^{n+1}$.
- E.PublicKeyGen($params, sk$): Takes as input a secret key $sk = \boldsymbol{s} = (1, \boldsymbol{s}')$ with $s[0] = 1$ and $\boldsymbol{s}' \in R_q^n$ and the $params$. Generate matrix $A' \leftarrow R_q^{N \times n}$ uniformly and a vector $\boldsymbol{e} \leftarrow \mathcal{X}^N$ and set $\boldsymbol{b} \leftarrow A's' + 2e$. Set A to be the $(n+1)$-column matrix consisting of $\boldsymbol{b}$ followed by the n columns of -A'. (Observe: $A \cdot \boldsymbol{s} = 2\boldsymbol{e}$.) Set the public key $pk = A$.
- E.Enc($params, pk, m$): To encrypt a message $m \in R_2$, set $\boldsymbol{m} \leftarrow (m, 0, ..., 0) \in R_q^{n+1}$, sample $r \leftarrow R_2^N$ and output the ciphertext

$\boldsymbol{c} \leftarrow \boldsymbol{m} + A^T\boldsymbol{r} \in R_q^{n+1}$.

- E.Dec($params, sk, \boldsymbol{c}$): Output $m \leftarrow [[\langle \boldsymbol{c}, \boldsymbol{s} \rangle]_q]_2$.

Figure 2.3: Basic GLWE-Based Encryption Scheme [8].

Lets consider the LWE hardness assumption for simplification. Assume the ring $\mathbb{Z}_q$, the ring over the integers modulo an odd integer q and a ciphertext $c \in R^n$, which encrypts the message m under the secret key s, then

$$m = [[\langle c, s \rangle]_q]_2 \in R_2,$$

holds. The expression $[\cdot]_q$ denotes the modular reduction in the interval $(-q/2, q/2)$. As long as the error/noise e, calculated as

$$e = [\langle c, s \rangle]_q = \langle c, s \rangle - kq \in R_q,$$

does not exceed the bound $|e| < q/2$, we are able to decrypt the message. The error grows with each operation. For two ciphertexts with noises at most B, addition results into a magnitude of the noise at most $2B$. Multiplication increases the magnitude of the noise at most B^2.
It is obvious to see that if we continue to multiply our ciphertext we will end up growing exponential.
Now suppose that our modulus q is such that $q \approx x^L$. By multiplying two ciphertexts, with a noise magnitude of x, the error increases to x^2. If we continue multiplication the noise reaches the ceiling after only $\log L$ multiplications.

2.7.1 Key Switching

In [17] Brakerski and Vaikuntanathan introduced procedures for relinearization (i.e. dimension reduction) which allow us to reduce the ciphertext growth after multiplication.
After the multiplication of two freshly encrypted ciphertexts $c_1, c_2 \in R_q^2$ which encrypt $m_1, m_2 \in R_p$ under key $s_1 = (1, s) \in R_q^2$ we end up with a new ciphertext $c_3 = c_1 \otimes c_2 \in R_q^3$ which encrypts the product $m_1 \cdot m_2$ under $s_1' = s_1 \otimes s_1 \in R_q^3$. With each multiplication the ciphertext would grow

exponentially. Therefore we have to reduce the size with the key switching method. As the name implies key switching transforms a ciphertext $c_3 \in R_q^3$ into $c' \in R_q^2$ under $s_2 \in R_q^2$ while maintaining the following equality:

$$\langle c_3, s_1' \rangle \equiv \langle c', s_2 \rangle \mod q$$

Before we start we will explain some subroutines that are used within the key switching technique as described in [17]:

- $BitDecomp(x \in R_q^n, q)$ decomposes x into its bit representation. Namely, write $x = \sum_{j=0}^{\lfloor \log q \rfloor} 2^j \cdot u_j$, where all of the vectors u_j are in R_2^n, and output $\{u_0, u_1, ..., u_{\lfloor \log q \rfloor}\} \in R_2^{n \cdot \lceil \log q \rceil}$.
- $Powersof2(x \in R_q^n, q)$ outputs the vector $\{x, 2 \cdot x, ..., 2^{\lfloor \log q \rfloor} \cdot x\} \in R_q^{n \cdot \lceil \log q \rceil}$.

It is important to notice that for vectors c, s of equal length we have $\langle BitDecomp(c, q), Powersof2(s, q) \rangle \equiv \langle c, s \rangle \mod q$.
Key switching consists of two procedures. In the first step we generate a key switching key which represents the encryption of parts of s_1 under s_2 as described in [17] but with our exemplary sizes:

$SwitchKeyGen((s_1 \otimes s_1) \in R_q^3, s_2 \in R_q^2)$:

1. Run $A \leftarrow E.PublicKeyGen(s_2, N)$ for $N = 3 \cdot \lceil \log q \rceil$.
2. Set $B \leftarrow A + Powersof2(s_1)$ (Add $Powersof2(s_1) \in R_q^N$ to A's first column.) Output $\tau_{s_1 \to s_2} = B \in R_q^{N \times 2}$.

$SwitchKey(\tau_{s_1 \to s_2}, c_1)$: Output $c_2 = BitDecomp(c_1)^T \cdot B \in R_q^2$

It would be possible to do a key switch without the subroutines $BitDecomp$ and $Powersof2$. But the error in the ciphertext would increase rapidly. The following formula shows how the error behaves during key switching:

$$\langle c_2, s_2 \rangle = 2 \langle BitDecomp(c_1), e_2 \rangle + \langle c_1, s_1 \rangle \mod q.$$

Because the dot product $\langle BitDecomp(c_1), e_2 \rangle$ is very small the error increases by a small additive factor.

2.7.2 Modulo Switching

Modulus switching helps to reduce the noise magnitude in ciphertexts. It allows us to keep the noise level constant by decreasing the size of the modulus q. However, making q smaller reduces the homomorphic capacity (i.e., multiplicative depth) of the scheme.
Lets consider an odd number p and a ciphertext c modulus another odd number q. After a modulus switch, the new ciphertext c' modulus p can be decrypted to the same message, i.e., the following equation holds:

$$[\langle c, s\rangle]_q \equiv [\langle c', s\rangle]_p \mod 2.$$

More importantly we can say if $q > p > 1$ than the magnitude of the error decreases

$$[\langle c', s\rangle]_p \approx (p/q) \cdot [\langle c, s\rangle]_q.$$

Lets assume $q \approx p \cdot x$ and the magnitude of the error of the ciphertext c under modulo q is x^2 than the magnitude of the error in ciphertext c' under modulo p would be x.
To achieve multiple reductions of the modulus, we take a ladder of decreasing moduli $q_L > ... > q_1 > q_0$ such that $q_i \approx q_{i+1}/x | \forall i \in \{0, 1, ..., L\}$. After each multiplication, we switch the modulus and accomplish the ability to multiply ciphertexts up to a depth of L. After reaching the last modulo, it is obvious that we cannot switch moduli any more. However, we can use bootstrapping to reset the ciphertext's noise level again.
We illustrate the full BGV scheme in Figure 2.4, including the refreshing of the ciphertext using the modulo switch operation.

- FHE.Setup($1^\lambda, 1^L, b$): Takes as input the security parameter λ, a number of levels L, and a bit b. Use the bit $b \in 0, 1$ to determine whether we are setting parameters for a LWE-based scheme (where $d = 1$) or a RLWE-based scheme (where $n = 1$). Let $\mu = \mu(\lambda, L, b) = \theta(\log \lambda + \log L)$ be a parameter that we will specify in detail later. For $j = L$ (input level of circuit) to 0 (output level), run $params_j \leftarrow E.Setup(1^\lambda, 1^{(j+1)\cdot\mu}, b)$ to obtain a ladder of decreasing moduli from $q_L((L+1)\cdot\mu$ bits) down to $q'(\mu$ bits). For $j = L - 1$ to 0, replace the value of $d_j \in params_j$ with $d = d_L$ and the distribution $\mathcal{X}_j$ with

$\mathcal{X} = \mathcal{X}_L$. (That is, the ring dimension and noise distribution do not depend on the circuit level, but the vector dimension n_j still might.)

- FHE.KeyGen($\{params_j\}$): For $j = L$ down to 0, do the following:
  1. Run $s_j \leftarrow E.SecretKeyGen(params_j)$ and $A_j \leftarrow E.PublicKeyGen(params_j, s_j)$.
  2. Set $s'_j \leftarrow s_j \oplus s_j \in R_{q_j}^{\left\lceil \frac{nj+1}{2} \right\rceil}$. That is, s'_j is a tensoring of s_j with itself whose coefficients are each the product of two coefficients of s_j in R_{q_j}.
  3. Set $s''_j \leftarrow BitDecomp(s'_j, q_j)$.
  4. Run $\mathcal{T}_{s''_{j+1} \Rightarrow s_j} \leftarrow SwitchKeyGen(s''_j, s_{j-1})$. (Omit this step when $j = L$.)

  The secret key sk consists of the s_j's and the public key pk consists of the Aj's and $\mathcal{T}s''_{j+1} \Rightarrow s_j$'s.
- FHE.Enc($params, pk, m$): Take a message in R_2. Run E.Enc(A_L, m).
- FHE.Dec($params, sk, c$): Suppose the ciphertext c is under key s_j . Run E.Dec(s_j, c). (The ciphertext could be augmented with an index indicating which level it belongs to.)
- FHE.Add(pk, c_1, c_2): Takes two ciphertexts encrypted under the same s_j . (If they are not initially, use FHE.Refresh (below) to make it so.) Set $c_3 \leftarrow c_1 + c_2 \mod q_j$. Interpret c_3 as a ciphertext under s'_j (s'_j's coefficients include all of s'_j's since $s'_j = s_j \oplus s_j$ and s_j 's first coefficient is 1) and output: $c4 \leftarrow$ FHE.Refresh($c_3, \mathcal{T}s''_j \Rightarrow s_{j-1}, q_j, q_{j-1}$)
- FHE.Mult(pk, c_1, c_2): Takes two ciphertexts encrypted under the same s_j . If they are not initially, use FHE.Refresh (below) to make it so.) First, multiply: the new ciphertext, under the secret key $s'_j = s_j \oplus s_j$, is the coefficient vector c_3 of the linear equation $L^{long}_{c_1,c_2}(x \oplus x)$. Then, output: $c4 \leftarrow$ FHE.Refresh($c_3, \mathcal{T}s''_j \Rightarrow s_{j-1}, q_j, q_{j-1}$)
- FHE.Refresh($c, \mathcal{T}s''_j \Rightarrow s_{j-1}, q_j, q_{j-1}$): Takes a ciphertext encrypted under s'_j , the auxiliary information $\mathcal{T}s''_j \Rightarrow s_{j-1}$ to facilitate key switching, and the current and next moduli q_j and q_{j-1}. Do the following:

1. Expand: Set $c_1 \leftarrow$ Powersof2(c, q_j). (Observe: $\left\langle c_1, s_j'' \right\rangle \equiv \left\langle c, s_j' \right\rangle$ mod q_j by Lemma 2.)
2. Switch Moduli: Set $c_2 \leftarrow$Scale$(c_1, q_j, q_{j-1}, 2)$, a ciphertext under the key s_j'' for modulus q_{j-1}.
3. Switch Keys: Output $c_3 \leftarrow$ SwitchKey$(\mathcal{T}_{s_j'' \Rightarrow s_{j-1}}, c_2, q_{j-1})$, a ciphertext under the key s_{j-1} for modulus q_{j-1}.

Figure 2.4: BGV FHE Scheme without Bootstrapping [8].

2.7.3 Batching

In this section, we want to talk about batching, which is another improvement in performance for many FHE schemes, including BGV. We cannot profit from the full potential of the BGV-scheme by encrypting only a single integer per ciphertext.
The concept of batching is based on the Chinese Remainder Theorem, and applicable in the RLWE version of the BGV with the ring $R = \mathbb{Z}/\Phi_m(x)$. The Chinese Remainder Theorem makes it possible to encode multiple elements of $\mathbb{Z}_p$ within one polynomial R_p. Performing an operation in R_p also performs the operation on all encoded integers in Z_p without influencing each other.
In BGV, the number of plaintexts p_{slots} depends on the plaintext modulus p and the polynomial $\Phi_m(x)$ like

$$p_{slots} = \frac{\phi(m)}{ord_m(p)}$$

where $\phi(\cdot)$ describes the Euler totient function and $ord_m(p)$ is the multiplicative order of p modulo m. The variable $m \in \mathbb{Z}^+$ defines $\Phi_m(x)$, where $\Phi_m(x)$ is the m-th polynomial which has degree $n = \phi(m)$ and is monic and irreducible.
In the literature, batching is often compared to SIMD instructions of modern CPUs, which allow executing code in parallel on multiple data points. Using batching allows us to drastically speed up the evaluation of parallelizable circuits.

2.8 HElib

In this chapter, we will discuss the different computational aspects of the HElib [4] library, an open-source implementation of FHE schemes. The primary developers Shai Halevi and Victor Shoup released it first in 2013 right after Craig Gentry, the person responsible for the breakthrough in FHE in 2009, switched to IBM.
The library implements two of the most promising FHE schemes, namely the Brakerski-Gentry-Vaikuntanathan (BGV) [8] and he Approximate Number scheme of Cheon-Kim-Kim-Song (CKKS) [7]. HElib is written in C++ and uses the NTL library to support big integer operations and multi-threading. It implements all described optimizations, including modulus switching, relienarization (key-switching), bootstrapping and ciphertext packing (batching).

2.8.1 Operation Costs

Besides the described operations, HElib supports the addition and multiplication with plaintext constants as well. These constant operations are usually cheaper then ciphertext-ciphertext operations and have less impact on the noise level.
Furthermore, several operations modifying the packing of the ciphertexts are supported, like slot shifting and slot rotation. These operations allow us to operate on the original plaintexts encoded in one ciphertext.

Operation	Time	Noise	Comments
Addition	cheap	cheap	entry-wise addition of vectors
Constant-add	cheap	cheap	entry-wise addition of a constant vector
Multiplication	expensive	expensive	entry-wise multiplication of vectors
Constant multiply	cheap	moderate	entry-wise multiplication by constant vector
Rotation	expensive	cheap	cyclic rotation of vector by any amount

Table 2.2: Homomorphic operations and their costs [18].

2.8.2 Binary Operation Costs

In our implementation, we use the binary plaintext space $\mathbb{Z}_2$, which means our encrypted data consists of bits only. To support operations like addition, multiplication, or division on multi-bit input, we have to implement a binary-circuit.
HElib [4] already implements addition, multiplication, and comparison of multi-bit ciphertexts. We want to mention that these implementations focus on reducing the depth of the circuit and are, therefore, not optimized for runtime.
In Table 2.3 we list the cost of different binary circuits in HElib.

Operation	Lvl	Complexity	Comments
AddTwoNumbers	$\lceil log_2(t+2) \rceil$	$\mathcal{O}(t log_2(t))$	Addition of two t-bit values
AddManyNumbers	$\lceil \log_{3/2}(n) \rceil +$ $\lceil \log_2(t + \log_{3/2}(n) + 2) \rceil$		Addition of n t-bit values
MultTwoNumbers	$1 + \lceil \log_{3/2}(t) \rceil +$ $\lceil \log_2(t + \log_{3/2}(t) + 2) \rceil$		Multiply two t-bit values
CompareTwoNumbers	$\lceil log_2(t+2) \rceil$	$\mathcal{O}(t)$	Comparison of two t-bit values

Table 2.3: Homomorphic operations and their costs [19].

3 Decision Tree

In this chapter, we want to explain how a Decision Tree training algorithm works and how we transformed it into a boolean circuit, which we can evaluate through FHE on encrypted training data. Different problems arise, translating the learning algorithm into a homomorphic process. We will discuss each of these problems and their solutions.
A Decision Tree is a flowchart-like structure in the form of a tree. Each node on the path illustrates a decision, which leads to a leaf node or terminal node representing a class-label.
Tree-based learning is known to be one of the most straightforward learning procedures, even though it handles a large variety of learning problems. The only limitation to its applicability is the necessity of labeled training data. Since we use supervised learning, the algorithm needs training data with class-labels assigned to it to produce a model. In addition to linear problems, they also allow us to solve non-linear problems as well.
In Figure 3.1, we see how a Decision Tree is constructed from the example dataset in Table 3.1 taken from [20]. The first step in training a Decision Tree is to split the current set of data by a splitting criterion. The criterion yields a decision for our path through the tree representing the best attribute for splitting. Then we separate the data into all the unique values of this attribute. The attribute serves as our decisions, which splits our dataset into smaller and smaller subtrees.
If a split or subset contains only one class-label, it is called a pure set and does not need any further splitting. Therefore, these nodes turn into leaf nodes representing the remaining class-label. Repeating the process leads to a tree, which consists of decisions and class-labels.
To evaluate a new data sample, we simply follow each decision until we reach a leaf node that holds the classification result.

Day	Outlook	Humidity	Wind	Play
1	sunny	high	weak	no
2	sunny	high	strong	no
3	overcast	high	weak	yes
4	rainy	high	weak	yes
5	rainy	normal	weak	yes
6	rainy	normal	strong	no
7	overcast	normal	strong	yes
8	sunny	high	weak	no
9	sunny	normal	weak	yes
10	rainy	normal	weak	yes
11	sunny	normal	strong	yes
12	overcast	high	strong	yes
13	overcast	normal	weak	yes
14	rainy	high	strong	no

Table 3.1: Decision Tree example dataset [20].

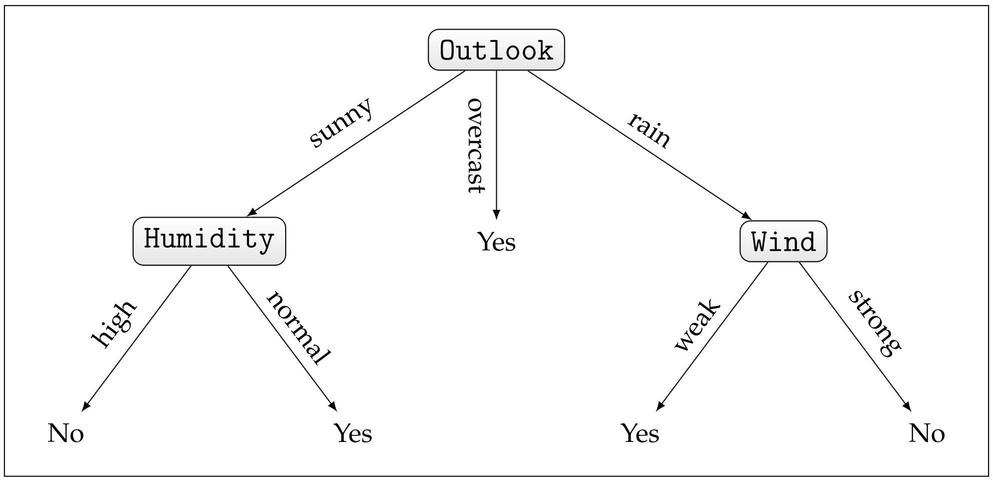

Figure 3.1: Decision Tree example.

3.1 Notation

Table 3.2 illustrates the used notation, all variables and their dimensions.

Symbol	Description
$\boldsymbol{y} \in \mathbb{Z}^n$	Vector of target values
$y \in \mathbb{Z}$	Target value
$A \in \mathbb{Z}^{n \times m}$	Matrix of attributes/features
$\mathbf{a} \in \mathbb{Z}^n$	Vector of attributes/features
$L \in \mathbb{Z}^j$	Set of labels
$l \in \mathbb{Z}$	Label
$D \in \mathbb{Z}^{n \times (m+1)}$	Matrix of attributes/features and target values
$\mathbf{d} \in \mathbb{Z}^{(m+1)}$	Vector of attributes/features and target value

Table 3.2: Notation Decision Tree.

Each row of the training data consists of a label $y \in \mathbb{Z}$ and a feature vector $\mathbf{d} \in \mathbb{Z}^m$ with m denoting the number of features. Consider the vector $\mathbf{y} \in \mathbb{Z}^n$ representing the target values and matrix $A \in \mathbb{Z}^{n \times m}$ containing all features of our training data. Combining both into one matrix $D \in \mathbb{Z}^{n \times (m+1)}$ with elements $\mathbf{d} \in D$ where $d_0 = y$ and $\{d_1, ..., d_m\} = \mathbf{a}$ holds.
In the upcoming sections we will use the indicator function which is defined as:

$$\mathbb{1}_A(z) := \begin{cases} 1 & \text{if } z \in A, \\ 0 & \text{if } z \notin A. \end{cases}.$$

The next definition selects all elements $\mathbf{d} \in D$ which fulfil the condition $d_x \odot v$, with $\odot \in \{=, <, \leq, >, \geq\}$, the element's index $x \in \{1, ..., |d|\}$, and v as the comparative value:

$$\sigma_{x,\odot,v}(D) = \{\mathbf{d} \in D | d_x \odot v\}$$

3.2 ID3, C4.5, Cart

There are many different implementations and definitions on how to train a Decision Tree. This section will explain the three most popular procedures. First, we will talk about the "Iterative Dichotomiser 3" (ID3) training algorithm [5], which is one of the simplest Decision Tree algorithms. There are multiple implementations of ID3, depending on the definition of the learning problem. It uses the Information Gain, which we describe in Equation 3.1, as its splitting criterion. The criterion is an impurity based measure, which tries to reduce the entropy of the dataset by splitting it with the attribute decreasing the entropy the most. In [21] the formula for the Information Gain is:

$$\begin{aligned} InformationGain(x, D) = Entropy(D) - \\ \frac{1}{|D|} \sum_{v \in unique(x,D)} |\sigma_{x,=,v}(D)| \cdot Entropy(\sigma_{x,=,v}(D)), \end{aligned} \tag{3.1}$$

where $unique(x, D)$ are the unique feature values d_x of the elements $d \in D$ on index $x \in \{1, ..., m\}$. The Entropy is defined in [21] as

$$Entropy(D) = \sum_{l \in L} \frac{|\sigma_{0,=,v}(D)|}{|D|} \cdot \log_2 \frac{\sigma_{0,=,v}(D)}{|D|} \tag{3.2}$$

where $d_{i,0}$ is the target values of row i.
The data is split by the attribute's distinct values, each representing a new subset. In [21], the algorithm has the following stopping conditions:

- All instances in the training set belong to a single target value of $\mathbf{y}$.
- The maximum tree depth has been reached.
- The number of cases in the terminal node is less than the minimum number of cases for parent nodes.
- While splitting a node, the number of cases in one or more child nodes would be less than the minimum number of cases for child nodes.
- The best splitting criterion is smaller than a certain threshold.

The algorithm itself has some drawbacks, but it stands out for its simplicity. In [21], they describe a list of drawbacks:

- ID3 does not guarantee an optimal solution, and it can get stuck in local optimums because it uses a greedy strategy. During a search, we can use backtracking to avoid local optimum.
- ID3 can overfit to the training data. Smaller Decision Trees should be preferred over larger ones to avoid overfitting. This algorithm usually produces small trees, but it does not always produce the smallest tree.

- ID3 is designed for nominal attributes. Therefore, continuous data can be used only after converting them to nominal bins.

The C4.5 [22] is an evolution of the ID3 algorithm. The algorithm uses the gain ratio, described in [22], as its splitting criterion:

$$GainRatio(x, D) = \frac{InformationGain(x, D)}{Entropy(D)} \tag{3.3}$$

C4.5 stops splitting when the number of instances to split is below a certain threshold. The improvements of the new algorithms, as stated in [21], are:

- C4.5 uses a pruning procedure that removes branches that do not contribute to the accuracy and replace them with leaf nodes.
- C4.5 allows attribute values to be missing (marked as ?).
- C4.5 handles continuous attributes by splitting the attribute's value range into two subsets (binary split). Specifically, it searches for the best threshold that maximizes the gain ratio criterion. All values above the threshold constitute the first subset, and all other values constitute the second subset.

Another algorithm we want to introduce is called Cart, which stands for "Classification And Regression Tree", which is implemented in the well-known Decision Tree classifier in the scikit-learn [23] library for python. The Cart algorithm utilizes the Gini Index, which will be explained later in subsection 3.3.2, as its splitting criteria. Like in C4.5, the tree consists of binary splits. Therefore, it can handle numerical and continuous data because the split depends on a specific value of an attribute and not on the attribute itself. The algorithm can be used for classification and regression as well.
The implementation in this thesis favors the Cart algorithm because of the simple to implement splitting criterion. Since the Cart algorithm does not guarantee an optimal solution, the homomorphic implementation does not guarantee one as well. The stopping criteria for Cart and the ID3 algorithm are the same. To support continuous data, we multiply the values with the precision needed for the current data set. After we scaled the values appropriately, we convert them to integer values. To make use of nominal values, we apply a special encoding called One-Hot Encoding (Section 3.4).

```
TreeGrowing($D, SplitCriterion, StoppingCriterion$)
Where:
    $D$ - Features + Target Values
    $SplitCriterion$ — the method for evaluating a certain split
    $StoppingCriterion$ — the criteria to stop the growing process
Create a new tree $T$ with a single root node.
IF $StoppingCriterion(D)$ THEN
    Mark $T$ as a leaf with the most
    common value $d_0$ of the elements $d\ inD$ as a label.
ELSE
    $\forall x \in \{1, ..., m\}$ find $x$ that obtains the best $SplitCriterion(x, D)$.
    Label $T$ with $x$
    FOR each distinct outcome $v = d_x$ of all elements $d \in D$:
        Set $Subtree_i = TreeGrowing((d \in D | d_x = v))$.
        Connect the root node of $T$ to $Subtree_i$ with
            an edge that is labeled as $v$
    END FOR
END IF
RETURN $TreePruning(T, D)$

TreePruning ($T, D$)
Where:
    $T$ - The tree to be pruned
    $D$ - Features + Target Values
DO
    Select a node $t \in T$ such that pruning it
    maximally improves some evaluation criteria
    IF $t \neq \emptyset$ THEN $T = pruned(T, t)$
UNTIL $t = \emptyset$
RETURN $T$
```

Figure 3.2: Top-down algorithmic framework for Decision Tree induction [21].

Figure 3.2 explains the steps of a general Decision Tree learning algorithm. After we finish learning the Decision Tree, we start pruning each node. Pruning removes branches of the tree that provide little estimation power by

checking each edge in the finished tree. We estimated that the recommended pruning procedures for the Cart algorithm would make a large impact on the performance of the homomorphic version. However, to evaluate the nodes, we would have to decrypt them by including the party with the secret key into the procedure. Otherwise, we must pursue all possible outcomes, which makes pruning obsolete. Since this was not the goal of our approach, we decided to leave the implementation of pruning procedures for future work.
There are two different stopping criteria used in our solution. The first bounds the tree to a certain depth to keep the tree small and to prevent overfitting. By reducing the depth of the tree, we can stop before all the binary splits are just representations of the training data itself.
The second stopping condition considers the size of the node to split. If a node contains fewer training examples than a given threshold, we stop splitting. Splitting a node with only a view training examples will hardly represent the whole training set. Moreover, a certain amount of training examples must conclude the resulting class in a leaf node otherwise the classification would also lead to overfitting.
Now we want to show a more precise description of the Cart procedure since the first algorithm shows a more general approach and omits the binary-splitting. The algorithm in 3.3 does not represent all definitions of the Cart algorithm, but it resembles our approach.

```
TreeGrowing($D, Depth, Min$)
Where:
    $D$ - Features + Target values
    $Depth$ - The remaining depth to calculate
    $Min$ - The minimal number of rows to split
Create a new tree $T$ with a single root node.
IF $Depth = 0$ OR $Min >= |D|$ THEN
    Mark $T$ as a leaf with the most
    common value $d_0$ of the elements $d\ inD$ as a label.
ELSE
    find $v$ and $x$ for $\forall i \in \{0, n-1\}, \forall x \in \{1, m\}$ with best
        $Gini((d \in D | d_x > v), (d \in D | d_x \leq v))$.
    Label $T$ with $v$ and $x$
    Set $T.left = TreeGrowing((d \in D|d_x > v), Depth - 1, Min)$.
    Set $T.right = TreeGrowing((d \in D|d_x \leq v), Depth - 1, Min)$.
END IF
RETURN $T$
```

Figure 3.3: Top-down algorithmic framework for Cart algorithm [21].

We can see how the algorithm splits the data into two subsets with the best value out of all attributes of all data rows in the set. We stop if the depth reaches zero or the subset of the current node becomes smaller than a certain threshold.
At last, we have to talk about the differences of our algorithm to the implementation of scikit-learn [24]. Scikit-learn uses no maximal depth as default, and they do include a threshold for the impurity reduction, which represents an additional stopping criterion. Additionally, scikit-learn allows most of the inputs to be weighted, but in the default configuration all weights are equal and do not influence the outcome.

3.3 Splitting Criteria

In this section, we explain how we chose the splitting criterion for our learning algorithm. There is a large variety of criteria concerning the classification and regression using Decision Trees. Most splitting functions are univariate. Therefore, they need only one attribute for splitting. The function provides us with a criterion to find the best attribute for splitting. In terms of homomorphic encryption, it is essential to find a rather simple function that can be calculated by addition and multiplication.
For our successor, we chose the so-called Gini Impurity or Gini Index, which is an impurity-based criterion. Impurity-based criteria state the reduction in the impurity of the training set D according to a value $v = d_{i,x} \in D$.

3.3.1 Division

One of the most expensive operations to create a Decision Tree based on binary circuits is division. For a n-bit value the division circuit stated in [25] has a multiplicative depth of n^2. For comparison adding two n-bit numbers costs approximately $\log_2 n$ multiplications, see Table 2.3, which shows us the tremendous costs of division. Another problem of division concerns the fact that we cannot use continuous values in the BGV-scheme. This limitation leads to a loss of precision after each division.
Most of the splitting criteria used for training Decision Trees require divisions. Nevertheless, we can overcome this problem by multiplying the training data before we encrypt. Another solution is the modification of our algorithm to avoid division.
In Section 3.3.2, we will explain how we can rewrite the selected splitting criteria to eliminate divisions.

3.3.2 Gini Index

This section explains the general formula of the Gini Index. Afterward, we will show how to change it to comply with our needs and why the original version did not.
The definition for the Gini Index is

$$Gini(D) = 1 - \sum_{l \in L} \left(\frac{\left| \{d_0 \in D \mid d_0 = l\} \right|}{|S|} \right)^2 \tag{3.4}$$

with L denoting the set of labels.
The Gini Index describes the probability of classifying a randomly picked data point incorrectly. Therefore, the best Gini Index would be 0. However, this is only possible if all the resulting splits are pure sets. As a reminder, a pure set describes a subset which consists exclusively of one target label. Figure 3.4 illustrates an example of a perfect split in which each of the two subsets is pure and has a Gini Index of 0.

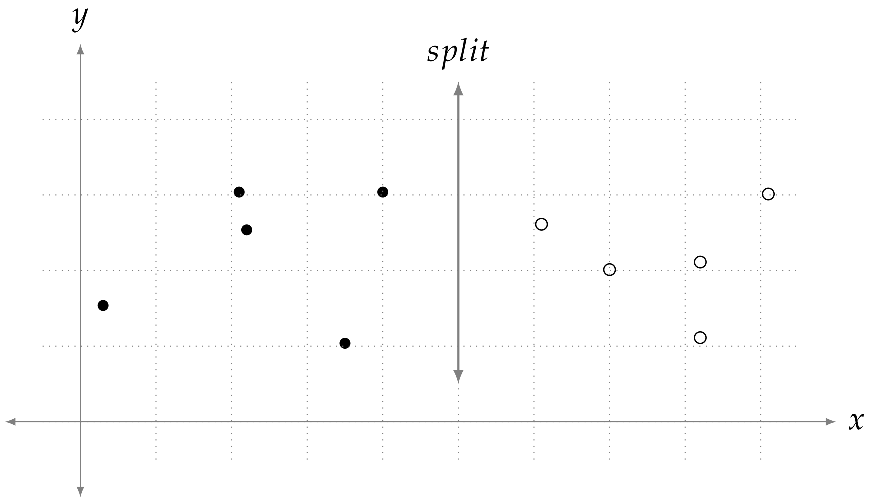

Figure 3.4: Perfect split example.

Formulated into an optimization problem, this means that we search for the attribute that splits our dataset so that the Gini Index becomes minimal. Normally the data would be split into all different values of each attribute. This solution causes problems if the domain of the target attribute is relatively wide. Therefore, we use a different approach. We do not only search for the best attribute to split but also for the best value $v = s_{i,a}$, where $i \in \{0, ..., N\}$ describes the row index and $a \in A$ the attribute index. The value splits the dataset into two different subsets

$$\begin{aligned} S_{left}(\theta) &= (s_{i,a} \in S \mid s_{i,a} > v) \\ S_{right}(\theta) &= (s_{i,a} \in S \mid s_{i,a} \leq v), \end{aligned} \tag{3.5}$$

where $i \in \{0, ..., N\}$ describes the row index and $\theta = \{a, v\}$ is the attribute index and the value to split. The Gini Index calculation for a binary split is weighted by the total number of instances in the parent node. The Gini Score for a chosen split point in a binary classification problem is therefore calculated as

$$Gini_{score}(S, \theta) = Gini(S_{left}(\theta)) \frac{|S_{left}(\theta)|}{|S|} + Gini(S_{right}(\theta)) \frac{|S_{right}(\theta)|}{|S|}. \tag{3.6}$$

Both results are weighted with the number of elements in each split. The minimization problem resulting from this is given by

$$\min_{\theta} \quad Gini_{score}(S,\theta) \tag{3.7}$$

Now, in order to make the formula usable for our homomorphic circuit, we ensure that we do not use any computations that are expensive in the sense of circuit depth. Therefore, we must eliminate divisions.
The following transformation of the formula shows the solution to this problem. We start by inserting the formula for the $Gini_{score}$ into the optimization problem

$$\min_{\theta} \quad Gini(S_{left}(\theta))\frac{|S_{left}(\theta)|}{|S|} + Gini(S_{right}(\theta))\frac{|S_{right}(\theta)|}{|S|}.$$

By inserting the formula for the Gini Index we get

$$\begin{aligned}\min_{\theta} \quad & 1 - \sum_{l \in L}\left(\frac{|\{s_{i,0} \in S_{left}(\theta) \mid s_{i,0} = l\}|}{|S_{left}(\theta)|}\right)^2 \frac{|S_{left}(\theta)|}{|S|} \\ + \quad & 1 - \sum_{l \in L}\left(\frac{|\{s_{i,0} \in S_{right}(\theta) \mid s_{i,0} = l\}|}{|S_{right}(\theta)|}\right)^2 \frac{|S_{right}(\theta)|}{|S|}.\end{aligned}$$

Next we get rid of the constant values and the variable S, which are equal for all splits. We subtract 2 and multiply by $|S|$

$$\begin{aligned}\min_{\theta} \quad & -\sum_{l \in L}\left(\frac{|\{s_{i,0} \in S_{left}(\theta) \mid s_{i,0} = l\}|}{|S_{left}(\theta)|}\right)^2 |S_{left}(\theta)| \\ & -\sum_{l \in L}\left(\frac{|\{s_{i,0} \in S_{right}(\theta) \mid s_{i,0} = l\}|}{|S_{right}(\theta)|}\right)^2 |S_{right}(\theta)|\end{aligned}$$

Now we want to rearrange the formula so it contains only one common divisor

$$\begin{aligned}
\max_{\theta} \quad & \frac{\sum_{l \in L}(\left|\{s_{i,0} \in S_{left}(\theta) \mid s_{i,0} = l\}\right|)^2}{\left|S_{left}(\theta)\right|} \\
+ \quad & \frac{\sum_{l \in L}(\left|\{s_{i,0} \in S_{right}(\theta) \mid s_{i,0} = l\}\right|)^2}{\left|S_{right}(\theta)\right|} \\
= \max_{\theta} \quad & \frac{\sum_{l \in L}(\left|\{s_{i,0} \in S_{left}(\theta) \mid s_{i,0} = l\}\right|)^2 \left|S_{left}(\theta)\right|}{\left|S_{left}(\theta)\right| \left|S_{right}(\theta)\right|} \\
+ \quad & \frac{\sum_{l \in L}(\left|\{s_{i,0} \in S_{right}(\theta) \mid s_{i,0} = l\}\right|)^2 \left|S_{right}(\theta)\right|}{\left|S_{left}(\theta)\right| \left|S_{right}(\theta)\right|}
\end{aligned}$$

Last but not least, we rewrite the Gini Score so we end up with two terms, $Gini_{value}$ for Gini Value and $Gini_{coeff}$ for Gini Coefficient, which divide each other and can be compared to different splits of the current subset S.
By substituting the terms above and below the division we get

$$\max_{\theta} \quad \frac{Gini_{value}(S,\theta)}{Gini_{coeff}(S,\theta)}. \tag{3.8}$$

In order to perform comparisons without any division we rewrite the comparison of two different splits. We consider the two different values $\theta_1, \theta_2 \in S$ which each respectively create a split. The modifications to compare two possible splits with each other are:

$$\begin{aligned}
& Gini_{score}(S,\theta_1) > Gini_{score}(S,\theta_2) \\
\Leftrightarrow \quad & \frac{Gini_{value}(S,\theta_1)}{Gini_{coeff}(S,\theta_1)} > \frac{Gini_{value}(S,\theta_2)}{Gini_{coeff}(S,\theta_2)} \\
\Leftrightarrow \quad & Gini_{value}(S,\theta_1) \cdot Gini_{coeff}(S,\theta_2) > Gini_{value}(S,\theta_2) \cdot Gini_{coeff}(S,\theta_1)
\end{aligned} \tag{3.9}$$

By using Equation 3.9 to determine the highest $Gini_{score}$, we can prevent division throughout the whole process. There are a few more aspects in the sense of optimizing the algorithm, which we discuss later in Chapter 4 of this thesis.

3.4 One-Hot Encoding

There are two different types of data used in statistics. First of all, numerical data, for example, the length of a car or a person's height. Secondly, categorical data which represents characteristics like gender or color.

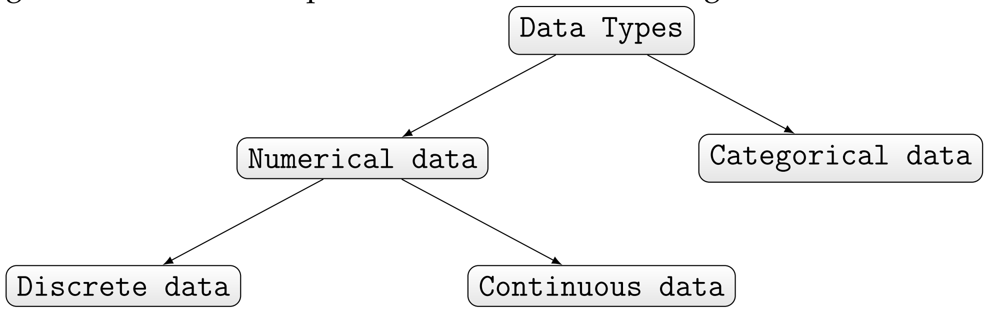

Figure 3.5: Statistical data types.

Discrete and continuous data are subgroups of numerical data, as shown by Figure 3.5. The set of integers describes discrete values, while continuous data consists of real values.
Since the BGV scheme supports only the set of integers, we introduce some methods to extend the range of data types.
To handle continuous data, we must figure out the required precision for the training data in question. The precision may vary between different datasets, and it requires testing for each model. Before converting all values to integers, we enlarge them to meet the tolerances.
Next, we show how we handle categorical inputs. Normally, one would split categorical information into all possible outcomes, but our Decision Tree algorithm uses binary-splitting, which supports only two possible outcomes. For this purpose, we use so-called indicator variables or dummy variables [26]. In the scikit-learn environment, they use the term One-Hot Encoding [27] for the procedure generating those variables.
The encoding takes all categorical values of an attribute and transforms them into attributes, represented as yes-no questions.
In Figure 3.6, we encode the attribute "Outlook" with the One-Hot Encoding.

Original			Encoded		
Day	Outlook	Day	isOvercast	isRainy	isSunny
1	sunny	1	No	No	Yes
2	sunny	2	No	No	Yes
3	overcast	3	Yes	No	No
4	rainy	4	No	Yes	No
5	rainy	5	No	Yes	No
6	rainy	6	No	Yes	No
7	overcast	7	Yes	No	No
8	sunny	8	No	No	Yes
9	sunny	9	No	No	Yes
10	rainy	10	No	Yes	No
11	sunny	11	No	No	Yes
12	overcast	12	Yes	No	No
13	overcast	13	Yes	No	No
14	rainy	14	No	Yes	No

Figure 3.6: One-Hot Encoding example [20].

In summary, we can say that our implementation supports all the different data types used in learning problems. However, we must keep in mind that for continuous data, we may lose accuracy if we do not choose precise bounds.

3.5 Random Forest

In this section, we explain how to use the Decision Tree training algorithm to create a Random Forest classifier[28]. As the name might suggest, Random Forests consist of a collection of Decision Trees. The algorithm enables us to solve classification and regression problems. Every tree represents a classification/regression result, which contributes in the form of a majority vote, to the overall prediction.

The basic principle behind a Random Forest is the wisdom of crowds. The trees protect each other from their errors as long as they are independent of each other. Therefore, the correlation has to be low between each model to produce an ensemble prediction that outperforms any of the individual

ones.
The question is, how does a random forest ensure that the models diversify each other? One method to prevent correlation is bagging, also called bootstrapping, not to confuse with bootstrapping of the BGV-scheme. To remind us again, Decision Trees are susceptible to changes in the training data. Small changes in the composition of the data will change the model drastically. Random Forests use this property to create a variety of completely different Decision Trees by simply batching random chunks of the training data together. This procedure is known as bagging.
It is important to notice that with bagging, we are not creating random subsets of smaller sizes, but we sample the set with replacements. Consider the example set $(1, 2, 3, 4, 5) \in \mathbb{Z}^5$ which may create two random samples $(1, 2, 2, 3, 4)$ and $(1, 2, 3, 3, 5)$. Both samples have the same length but completely different elements.
Another important aspect is the random selection of features. By excluding random features, we create a greater variety of trees to predict new information. As stated in [3], in the case of classification we require $\sqrt{m}$ randomly selected features of the total amount of features m. Figure 3.7 visualizes all steps of the procedure.

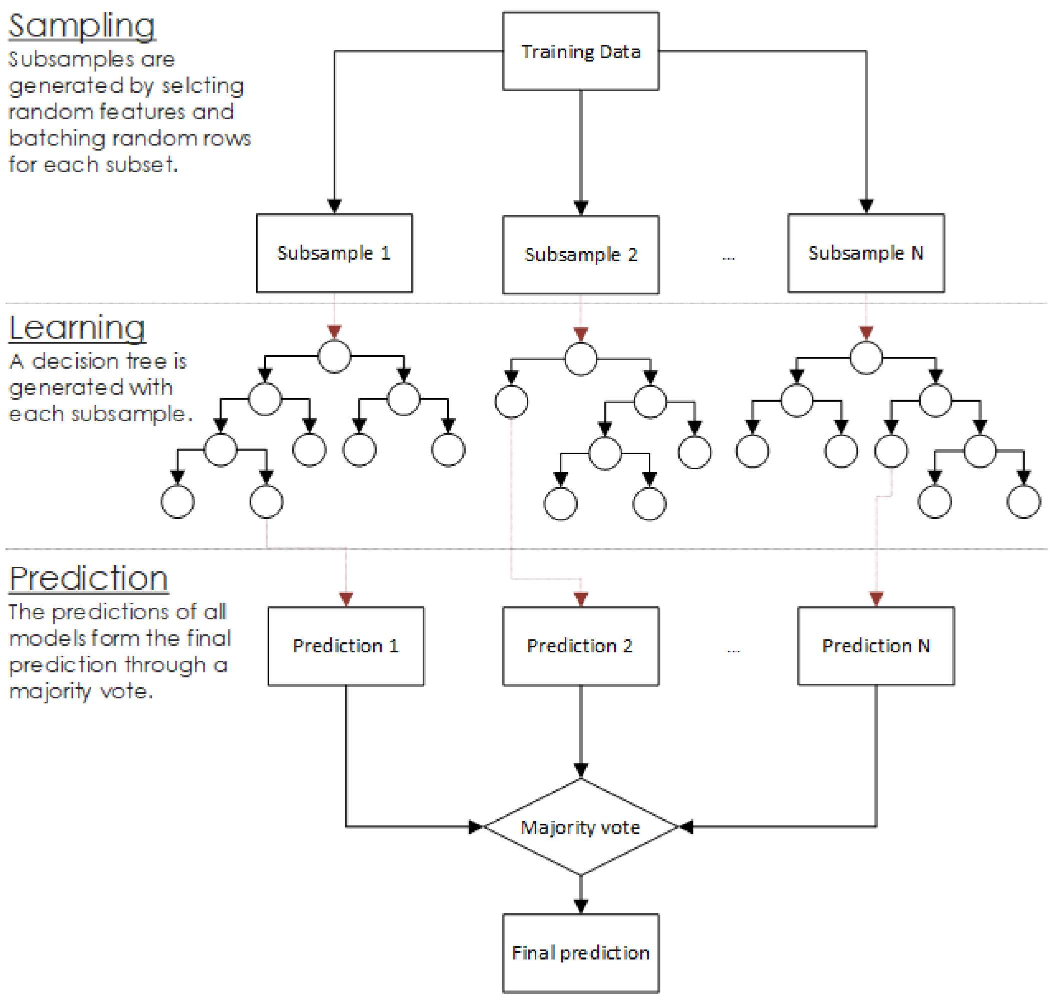
Sampling
Subsamples are generated by selcting random features and batching random rows for each subset.
Training Data
Subsample 1
Subsample 2
...
Subsample N
Learning
A decision tree is generated with each subsample.
Prediction
The predictions of all models form the final prediction through a majority vote.
Prediction 1
Prediction 2
...
Prediction N
Majority vote
Final prediction

Figure 3.7: Random Forest general overview.

Most learning algorithms require to tune the parameters to fit the current training data. In our implementation, only the party owning the data can evaluate the model. Therefore we have to make sure no tuning is required. The Random Forest learning algorithm overcomes this issue by selecting random samples and features, which makes tuning obsolete.

4 Implementation

In this chapter, we want to talk about the transformation of the Random Forest learning procedure into a homomorphic boolean circuit. We implemented the algorithm in both C++ using the HElib, and Python using the NumPy [29] and scikit-learn [23] libraries. To gain even more insight, we will compare certain parts of the code in both implementations.
Lets remember the real-world scenario for our solution from section 1.1. We have two parties, the customer and the cloud. The customer wants to create a model on which he can predict new data in the future. The cloud provides a service which constructs a model without seeing the data in plain at any point.
In the first phase, represented by figure 4.1, we show the communication between the customer and the cloud during the learning phase.

Training

customer		**Cloud**
$params \leftarrow FHE.Setup(\lambda, L)$		
$pk, sk \leftarrow FHE.KeyGen(params)$		
	$\xrightarrow{\text{pk}}$	
$A_{packed} \leftarrow packing(A_{plain})$		
$Y_{encoded} \leftarrow encode(\mathbf{y}_{plain})$		
$Y_{packed} \leftarrow packing(Y_{encoded})$		
$A_{pk} \leftarrow FHE.Enc(params, pk, A_{packed})$		
$Y_{pk} \leftarrow FHE.Enc(params, pk, Y_{packed})$		
	$\xrightarrow{A_{pk}, Y_{pk}}$	
		$cModel \leftarrow train(A_{pk}, Y_{pk}, pk)$

Figure 4.1: Training communication between customer and cloud.

The client starts by generating the parameters for encryption. Afterward, we encrypt the features A and target values $\mathbf{y}$ separately.
In the second phase, illustrated by figure 4.2, the customer wants to make a prediction on new data. The customer encrypts and sends the data into the cloud which then responds with a prediction. At last the customer decrypts the result.

Evaluating

customer		**Cloud**
$A_{packed} \leftarrow packing(\mathbf{a}_{plain})$		
$A_{pk} \leftarrow FHE.Enc(params, pk, A_{packed})$		
	$\xrightarrow{A_{pk}}$	
		$pred_{pk} \leftarrow predict(A_{pk}, Model_{pk}, pk)$
	$\xleftarrow{pred_{pk}}$	
$pred \leftarrow FHE.Dec(params, sk, pred_{pk})$		

Figure 4.2: Prediction communication between customer and cloud.

In the prediction phase, we could do the packing as well, but for simplicity, we constructed the model as well as the prediction for one sample at a time.

4.1 Preprocessing

For testing we use three different datasets, which we selected to represent different situations and scenarios. We chose different sizes of data to show the performance of the scheme. Since the HElib supports integers only, we must convert all non-integer features of the training data before training.

Name	Samples	Features	Classes	Strings	Continuous	Categorical
Tennis [20]	14	4	2	x		x
Iris [30]	150	4	3	x	x	
Balance [31]	625	4	3	x		

Table 4.1: Data Types included except integers in the different dataset.

Table 4.1 illustrates the required preprocessing steps for each of the datasets. For string-based features, we list all possible values and associate each distinct value with an integer.
Next, we encode categorical-based features with the One-Hot Encoding, as described in subsection 3.4. As a reminder, the One-Hot Encoding translates categorical data into numerical data. The downside of this method is that we end up with more features than we started with.
To handle continuous data, we multiply all values by the precision needed and cut the remaining decimals. In our case, we only scale the iris dataset. The data will be encrypted bitwise, which means that a n-bit number consists of n ciphertexts. To keep the explanations and dimensions of all variables more straightforward, we write $\mathbb{Z}$ instead of $\mathbb{Z}_2^n$ even though the second one would be more accurate.
In Table 4.2 we can see how the data changes with preprocessing. By encoding the categorical data in the tennis dataset the amount of features increases while the bits per feature (bpf) decrease. In the Iris dataset ,we multiply the whole set with 10, because all values have just one decimal place. This increases the bits needed to represent one feature of one row by 3 bits.

Name	Indexing	Upscaling	Encoding	Before		After	
				Features	Bits/Feature	Features	Bits/Feature
Tennis [20]	x		x	4	2	8	1
Iris [30]	x	x		4	3	4	7
Balance [31]	x			4	3	4	3

Table 4.2: Dimensions and preprocessing steps for each dataset.

Our tests have shown that preprocessing does not influence the performance of the scheme. Only the overall amount of samples changes how the learning algorithm performs. The increase of bpf from upscaling as well as the increase of features from the One-Hot-Encoding have a negligible impact

on the runtime.
In summary, we may say that our scheme supports every kind of data, without influencing the performance, as long as we preprocess it before encryption.

4.2 Model

This section describes the structure or the final model. The model represents a forest containing multiple Decision Trees. Those trees harbor multiple levels of decisions that classify new data. The two parameters:

- tree depth - the level at which we stop splitting
- tree count - the amount of Decision Trees

decide the size of the model from the start. Normally multiple stopping criteria influence the construction of each tree. Since we have no way of validating the criteria on encrypted data, we omit most of them. Therefore each Decision Tree will be a complete binary tree limited only by the tree depth n, resulting in $2n - 1$ nodes. In Figure 4.3, we can see the general data structure with all elements and attributes.

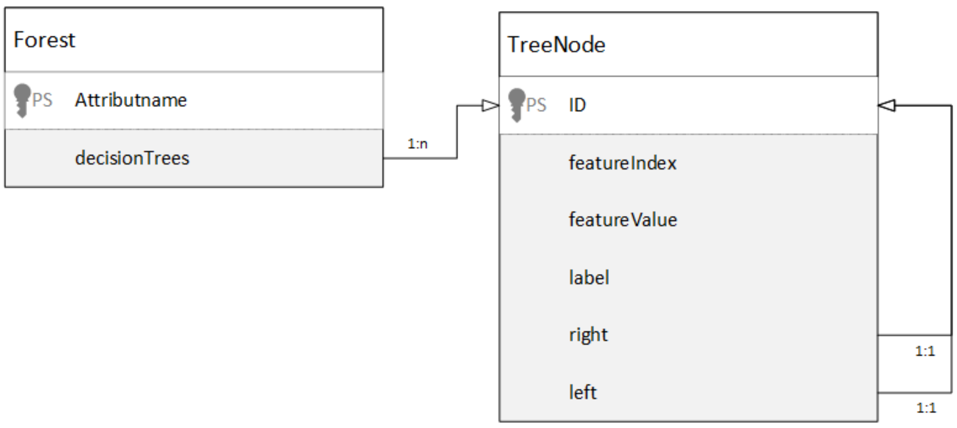

Figure 4.3: Model data structure.

4.3 Measuring

This section will explain how we measure the performance of the scheme and all its variations. Our scheme produces a forest of Decision Trees. Each of these trees has multiple levels of binary splits. To get a good comparison between long and short tree depths, we decided to take the timing of the generation of an average split $split_{mean}$. Therefore, we calculate the average time of all splits of all trees within one forest.

4.4 Packing

As a reminder, batching allows us to pack multiple plaintext values in so-called slots. In this section, we explain the different packing approaches as well as some encoding needed for computation later on. The number of slots n_{slots} one ciphertext holds depends on the parameters used to initialize the BGV scheme as we explained in Section 2.7.3.
Our training data consists of the feature matrix $A \in \mathbb{Z}^{n \times m}$ and a target value vector $\boldsymbol{y} \in \mathbb{Z}^n$, where n denotes the number of samples and m the number of features. Features and target values will be packed and encrypted separately.

4.4.1 General Consideration

The easiest idea to pack our data would be sample wise. As long as $m \leq n_{slots}$ we can pack all features of a sample into one ciphertext. It is important to notice that our construction does not work if $m > n_{slots}$. The next step, as we implemented it in this thesis, packs multiple samples into one ciphertext. The maximum samples per ciphertext spc_{max} is given by

$$spc_{max} = \lfloor n_{slots}/m \rfloor ,$$

which reduces the number of ciphertexts, produced by the feature matrix, to $\lceil \frac{n}{spc_{max}} \rceil$. If $(n_{slots} \mod m) \neq 0$ or $(n \mod spc_{max}) \neq 0$ then the remaining slots are filled with zeros.
Before we start packing the target values $\boldsymbol{y}$, we must encode them. This intermediate step is necessary to simplify the operations between target

values and all other variables in the later procedure.
The vector $\boldsymbol{y}'_l = (y'_{l,1}, ..., y'_{l,n}) \in \mathbb{Z}_2^n$ is computed as follows

$$y'_{l,i} = \mathbb{1}_{\{l\}}(y_i) \in \mathbb{Z}_2,$$

for all $i \in \{0, ..., n-1\}$. This holds for all labels $l \in L$.
Next we generate a matrix $Y_l \in \mathbb{Z}_2^{n\times m}$ for each label l like

$$Y_l = [\boldsymbol{y}'_l, ..., \boldsymbol{y}'_l] \in \mathbb{Z}_2^{n\times m}.$$

Now the feature matrix A and all Y_l matrices have the same dimensions. This makes it easier after encryption to perform operations between those matrices.
To give an example we consider the matrix $A \in \mathbb{Z}^{4\times 3}$, the vector $\boldsymbol{y} \in \mathbb{Z}^4$ and $s = 14$. The values of A and $\boldsymbol{y}$ are given below:

$\boldsymbol{a}_0$	$\boldsymbol{a}_1$	$\boldsymbol{a}_2$	$\boldsymbol{y}$
1	2	3	0
4	5	6	1
7	8	9	0
10	11	12	2
13	14	15	2

.

Now we pack all the features into two ciphertext vectors c_0, c_1, which is the maximum samples per ciphertext $spc_{max} = \lfloor n_{slots}/m \rfloor = \lfloor 14/5 \rfloor = 2$, and pad the remaining slots with zeros like

Slot	0	1	2	3	4	5	6	7	8	9	10	11	12	13
c_0	1	2	3	4	5	6	7	8	9	10	11	12	0	0
c_1	13	14	15	0	0	0	0	0	0	0	0	0	0	0

.

For the target values we do the same for each label $l \in L$. The resulting vectors look like

Slot		0	1	2	3	4	5	6	7	8	9	10	11	12	13
$l = 0$	c_0	1	1	1	0	0	0	1	1	1	0	0	0	0	0
	c_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$l = 1$	c_0	0	0	0	1	1	1	0	0	0	0	0	0	0	0
	c_1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$l = 2$	c_0	0	0	0	0	0	0	0	0	0	1	1	1	0	0
	c_1	1	1	1	0	0	0	0	0	0	0	0	0	0	0

,

while the last two slots are padded again.

4.4.2 Further Optimization

Packing as many samples as possible into one ciphertext is not always beneficial. Therefore we created different approaches by adjusting the samples per ciphertext spc to compare the performance of varying packing schemes. To introduce a more general definition for packing, we consider the dataset A and the resulting packed dataset $A' \in \mathbb{Z}^{\frac{n}{spc} \times n_{slots}}$, with its elements $\mathbf{a}'_i \in A'$ constructed like

$$\mathbf{a}'_i = (a_{i \cdot spc}, a_{i \cdot spc+1}, ..., a_{i \cdot spc+spc}) \in \mathbb{Z}^{n_{slots}}.$$

with $i \in \{0, ..., \lceil \frac{n}{spc} \rceil\}$. If $(n \mod spc) \neq 0$ as well if $(n_{slots} \mod m) \neq 0$ the remaining slots are filled with zeros. Using this general definition we want to introduce 4 different ways of packing by changing the packing rate spc. The first approach serves as a reference and encrypts every sample on its own. This results into $spc = 1$. This approach should be the worst-case scenario because we can barely profit from the parallelism of the homomorphic SIMD operations.

In the second approach, we pack the same amount of values into one row as we pack rows. While we create overhead by shifting the samples within the ciphertext in the learning process, we also increase the amount of parallelism. We calculate the packing rate spc for this scenario like

$$spc = \left\lfloor \sqrt[2]{\frac{n}{m}} \right\rfloor .$$

The third method packs the training data A in a way that the length of the packed dataset C roughly equals spc. In this scenario, we pack even more ciphertexts into one row. While the overhead from packing grows, the overall amount of ciphertexts decreases. For this scenario the spc equals

$$spc = \lfloor \sqrt[2]{n} \rfloor .$$

The final approach serves as a comparative value as well and packs as many values as possible into one ciphertext $spc = spc_{max}$. Even if the packing rate spc outgrows its maximum, we limit it with spc_{max}.

4.5 Split Values

In this section, we want to discuss an important optimization in the learning algorithm. We remember that in the Decision Tree learning procedure, we must find the best value to split the dataset. To achieve that, we try every value of any feature for all rows in our dataset. For example, a dataset with 625 samples and 4 features requires 2500 runs of the Gini Index to find the best value/feature combination to split. In our case, the problem reduces to 625 runs because we can calculate all 4 features in parallel using SIMD.
Now we consider two possible optimizations. The *bpf-approach* reduces the number of runs to the minimum number of bits per feature bpf needed to encrypt the dataset. Since we encrypt the data bitwise, we can perform this optimization even on encrypted data without any further effort. Each possible value in the set $\{0, \ldots, 2^{bpf} - 1\}$ has to be tested, which poses a significant improvement for specific datasets. For example, the highest value used in the balance dataset is 5. Therefore we need 3 bpf to encrypt the dataset. Consequently, we must test 2^3 values to find the best split.
The *distinct-values-approach* requires preprocessing before encryption. We generate a list of distinct values for each feature in the dataset. We notice an enormous improvement in attributes with categorical information. For example, we reduce the number of runs for the Balance dataset from 625 to 5. In the case, where the amount of possible split values outgrows the number of samples, the optimization becomes useless.
To utilize the compressed state of the data, we must pack the split values as well. If the length of split values varies between each feature, we must fill the missing values with zeros. Each of these vectors represents a column in the split value matrix $S \in \mathbb{Z}^{n' \times m}$ where n' represents the number of split values per feature. If we pack multiple samples per ciphertext with

$spc > 1$ we must duplicate each row vector $\boldsymbol{s} \in S$, which results into $S' = [\boldsymbol{s}, ..., \boldsymbol{s}] \in \mathbb{Z}^{n' \times (m \cdot spc)}$. In this case, we compare and calculate not only one but all features simultaneously.
Table 4.3 below shows an example of this procedure, with the feature matrix $A \in \mathbb{Z}^{5 \times 3}$ and a packing rate $spc = 2$.

A			Distinct			Packed $spc = 2$					
a_1	a_2	a_3	a_1	a_2	a_3	a_1	a_2	a_3	a_1	a_2	a_3
1	0	2	0	0	2	0	0	2	0	0	2
0	1	2	1	1		1	1	0	1	1	0
0	2	2		2		0	2	0	0	2	0
0	3	1		3		0	3	0	0	3	0
1	3	2									

Table 4.3: Encoding of split values.

Both approaches have their advantages and disadvantages, which is why we investigated both.

4.6 Training

Next, we want to talk about the training of the model. In chapter 3.5, we explained how the Random Forest classifier uses multiple Decision Trees to predict upon new data and combines the result via a majority vote. Since the implementation of the majority vote is straight forward, we want to focus on the construction of a Decision Tree.
For the HElib, it is vital to know the maximum amount of consecutive multiplications to estimate the size of the parameters. Therefore, each subsection analyses their required multiplicative depth, and in the end, we will merge all results to give an overall estimate for the whole procedure.
In our implementation, a Decision Tree will always be a binary tree because we cannot evaluate most of the stopping criteria as long as the data is encrypted. We limit the growth of the tree with the maximal depth criteria. If we reach a certain depth, the algorithm stops. The minimum size criterium contains the algorithm if a node has fewer elements than the threshold. In

this case, the node inherits the classification result of the parent. These two criteria reduce the complexity of the resulting trees which generalize the data well.
Our tree starts with a node called root, which splits up into a left and right node. For every node, we have to find the best feature and the best feature value to split the samples into sub-nodes. Finding the parameters includes the three following steps:

- Generate splits
- Calculate the Gini Indices
- Compare Gini Indices

To generate all possible splits, we use one of the two approaches from Section 4.5. For each split vector $\boldsymbol{s} \in \mathbb{Z}_{bpf}^{m \cdot spc}$, with m representing the number of features, spc the packing rate, and bpf the bits per feature, we separate the data rows into two subsets. Then we calculate the Gini Index and compare the results. We repeat this procedure for each subgroup created until we reach the maximum depth of the tree.
Consider the packed feature matrix $A \in \mathbb{Z}_{bpf}^{\lceil \frac{n}{spc} \rceil \times (m \cdot spc)}$ with n denoting the number of samples and the packed target value matrices $Y_i \in \mathbb{Z}_2^{\lceil \frac{n}{spc} \rceil \times (m \cdot spc)}$ with i denoting the label index for each label $L \in \mathbb{Z}^j$. The code below illustrates the general training procedure for splitting one node in python:

```
if depth > max_depth:
 return to_terminal(group,L)
splits = []
for s in S:
 split = copy.deepcopy(node)
 split['value'] = s[0:m]
 split['left'],split['right'] = split_by_value(node, D, s)
 split = gini_index(split, Y, L, min_size, m, spc)
  splits.append(split)

node = maxRow(splits)
node = maxFeature(node,featureCount)
```

Figure 4.4: Splitting one node in python.

Table 4.4 illustrates the input and output node generated by the code above.

Input	Output
Node $group \in \mathbb{Z}_2^{\lceil \frac{n}{spc} \rceil \times (m \cdot spc)}$ $label \in \mathbb{Z}^m$	Node $group \in \mathbb{Z}_2^{\lceil \frac{n}{spc} \rceil \times (m \cdot spc)}$ $label \in \mathbb{Z}^m$ $gini\_value \in \mathbb{Z}^m$ $gini\_coeff \in \mathbb{Z}^m$ $value \in \mathbb{Z}^m$ $index \in \mathbb{Z}_2^m$ $left:$ $group \in \mathbb{Z}_2^{\lceil \frac{n}{spc} \rceil \times (m \cdot spc)}$ $label \in \mathbb{Z}^m$ $right:$ $group \in \mathbb{Z}_2^{\lceil \frac{n}{spc} \rceil \times (m \cdot spc)}$ $label \in \mathbb{Z}^m$

Table 4.4: Input and output of splitting one node.

4.6.1 Generate Splits

We want to show, in more detail, how packing, explained in 4.4, improves the performance of the overall scheme. We start with the first part, which splits our current samples into two subsets $left, right \in \mathbb{Z}_2^{\lceil \frac{n}{spc} \rceil \times (m \cdot spc)}$ using the split vector s. Those subsets, as well as its parent, represent the boolean information if a feature value belongs to the set. The code below illustrates the procedure with our packed variables:

```
def split_by_value(node, A, s,spc):
    left = numpy.copy(node['group'])
    right = numpy.copy(node['group'])
    for row in range(len(A)):
        ni = A[row] < s
        left[row] &= ni
        right[row] &= ~ni
    return {'group':left},{'group':right}
```

Figure 4.5: Splitting a set into two subsets with a comparative value.

With packing we can reduce the original runtime of $O(m \cdot n \cdot bpf)$ to $O(\frac{n}{spc} \cdot bpf)$, while the multiplicative depth remains unchanged as illustrated in Table 4.5.

Name	Multiplicative depth
LVL$(split_by_value())$	$\lceil \log_2(bpf+2) \rceil + 1$

Table 4.5: Multiplicative depth of Gini Index.

4.6.2 Calculate Gini Index

Now we want to talk about the calculation of the Gini Index. Most of the encoding steps we have explained before come in handy right now. We split the calculation of the Gini Index into smaller parts. Figure 4.6 shows the whole calculation of the Gini Index.

```python
for g in ['left', 'right']:
 g_Y_l = []
 for l in L:
  g_Y = node[g]['group'] & Y[l]
  g_Y = numpy.sum(g_Y,axis=0)
  g_Y = numpy.sum(g_Y.reshape(spc, m),axis=0)
  g_Y_l.append(g_Y)

 g_count.append(numpy.sum(g_Y_l,axis=0))
 g_count[-1] = numpy.maximum(g_count[g],1)

 node[g]['label'] = numpy.argmax(g_Y_l,axis=0)
 mu = min_size > g_count[-1]
 node[g]['label'] = mux(node['label'],node[g]['label'],mu)

 g_Y_l_2 = numpy.power(g_Y_l, 2)
 g_Y_l_sum.append(numpy.sum(g_Y_l_2, axis=0))

 gini_parts = []
 gini_parts.append(g_Y_l_sum[0] * g_count[1])
 gini_parts.append(g_Y_l_sum[1] * g_count[0])
 node['gini_value'] = numpy.sum(gini_parts ,axis=0)
 node['gini_coeff'] = g_count[0]*g_count[1]
```

Figure 4.6: Calculating the Gini Index in python.

In Line 10 we make sure that the number of elements does not include zero which would lead to errors.
Lines 12 to 14 evaluate the minimum size criteria by using the intermediate results. Since we cannot evaluate the stopping criteria in plain, we use a multiplexer (Mux) to overcome this issue. If the dataset is big enough, the multiplexer selects the maximum of our label sums, while in the second case we select the result passed down from the parent. If one dataset is too small all of its children are as well. In this case the label propagates into each child until the maximum depth is reached.
Finally, we get the $Gini_{value}$ and $Gini_{coeff}$ which will be used for comparison with the other splits.
To make the analysis of the multiplicative depth easier we must first talk about the amount of bits needed to represent each intermediate result. With each bit the multiplicative depth increases accordingly, therefore Table 4.6 displays the maximum value of each of the intermediate results.

variable	maximum value
g_Y_l	n
g_count	n
$g\_Y\_l\_2$	n^2
$g\_Y\_l\_sum$	n^2
$gini\_parts$	$\max\{n^2, \lfloor \frac{2n}{3} \rceil^2 \cdot \lfloor \frac{n}{3} \rceil\}$

Table 4.6: Maximum value for each intermediate result.

Next define Bits(x), which returns the bits required to represent each of those values:

$$\textsc{Bits}(x) = \lfloor \log_2(x) \rfloor + 1$$

Table 2.3 shows the multiplicative depth for each of our four binary operations addTwoNumbers (atn), addManyNumbers (amn), multTwoNumbers (mtn) and compareTwoNumbers (ctn). The multiplicative depth of each of those operations are:

$$\begin{aligned}
\textsc{atn}(t) =& O(\lceil \log_2(t+2) \rceil) \\
\textsc{amn}(n', t) =& O(\lceil \log_{3/2}(n') \rceil + \textsc{mtn}(t + \log_{3/2}(n'))) \\
\textsc{mtn}(t) =& O(1 + \textsc{amn}(t, t)) \\
\textsc{ctn}(t) =& O(\lceil \log_2(t+2) \rceil)
\end{aligned}$$

with t denoting the number of bits and n' the amount of values.
Next we analyse each part of the Gini Index calculation. Table 4.7 represents the multiplicative depth of each step. We split the multiplicative depth into two columns. While the first column represents the multiplicative depth inherited by intermediate results, the second one shows the multiplicative depth required by the calculation itself.

Name	Multiplicative depth	
	Intermediate	New
$\textsc{lvl}(g\_Y\_l)$	$\textsc{lvl}(split\_by\_value())+$	$4 + \textsc{amn}(\frac{n}{spc \cdot 15}, 4) + \textsc{amn}(spc, \textsc{Bits}(\frac{n}{spc}))$
$\textsc{lvl}(g\_count)$	$\textsc{lvl}(g\_Y\_l)+$	$\textsc{amn}(\lvert Y \rvert, \textsc{Bits}(n)) + \textsc{ctn}(\textsc{Bits}(n)) + 1$
$\textsc{lvl}(g\_Y\_l\_2)$	$\textsc{lvl}(g\_Y\_l)+$	$\textsc{mtn}(\textsc{Bits}(n))$
$\textsc{lvl}(g\_Y\_l\_sum)$	$\textsc{lvl}(g\_Y\_l\_2)+$	$\textsc{amn}(\lvert Y \rvert, \textsc{Bits}(n^2))$
$\textsc{lvl}(gini\_parts)$	$\textsc{lvl}(g\_Y\_l\_sum)+$	$\textsc{mtn}(\textsc{Bits}(n^2))$
$\textsc{lvl}(node['gini\_value'])$	$\textsc{lvl}(gini\_parts)+$	$\textsc{mtn}(\textsc{Bits}(\max\{n^2, \lfloor \frac{2n}{3} \rceil^2 \cdot \lfloor \frac{n}{3} \rceil\}))$
$\textsc{lvl}(node['gini\_coeff'])$	$\textsc{lvl}(g\_count)+$	$\textsc{mtn}(\textsc{Bits}(n))$
$\textsc{lvl}(gini\_index())$		$\textsc{lvl}(node['gini\_value'])$

Table 4.7: Multiplicative depth of Gini Index calculation.

4.6.3 Compare Gini Indices

First, we have created all possible splits for the current node. Then we have calculated the Gini Indices for all of those splits, and now we want to compare all of them to find the best separation possible.
We represent each split as a node, containing the two groups of the split, $Gini_{value} \in \mathbb{Z}$ and $Gini_{coeff} \in \mathbb{Z}$. $Gini_{value}$ and $Gini_{coeff}$ hold m results, one for each feature, because we packed and calculated all features simultaneously. To compare two splits A, B we do

$$\max(A, B) = \textsc{mux}(A, B, A.Gini_{value} \cdot B.Gini_{coeff} > B.Gini_{value} \cdot A.Gini_{value})$$

, where MUX is a multiplexer selecting A if the comparison is true and B otherwise. If we compare all splits with each other we receive a node containing the maxima for all features. To reduce the multiplicative depth from $|s|$ to $\log_2(|s|)$ we use a hierarchical approach for comparing. Figure 4.9 shows the comparison of all splits with each other.

```
def maxRow(nodes,spc):
 q = Queue()
 [q.put(node) for node in nodes]
 while q.qsize() > 1:
        a,b = q.get(),q.get()
        giniIndexA = a['gini_value'] * b['gini_coeff']
        giniIndexB = b['gini_value'] * a['gini_coeff']
        mu = giniIndexA  > giniIndexB
        q.put( muxNode(a,b,mu,spc))
 return q.get()
```

Figure 4.7: Comparing the splits and selecting the maxima.

To analyse the multiplicative depth, we start by finding the maximum value for each of the different variables, listed in 4.8.

variable	max value
$node['gini\_value']$	$\max\{n^2, \lceil \frac{n}{2} \rceil^2 \cdot \lfloor \frac{n}{2} \rfloor\}$
$node['gini\_coeff']$	$\lceil \frac{n}{2} \rceil \cdot \lfloor \frac{n}{2} \rfloor$
$giniIndex$	$\max\{n^2 \cdot \lceil \frac{n}{2} \rceil \cdot \lfloor \frac{n}{2} \rfloor, \lceil \frac{n}{2} \rceil^3 \cdot \lfloor \frac{n}{2} \rfloor^2\}$

Table 4.8: Maximum values of the Gini Index calculation.

Now, that we have our maxima, we calculate the multiplicative depth required for the comparison of all splits:

$$\lceil \log_2(n) \rceil \cdot (\text{MTN}(\text{Bits}(node['gini\_value'])) + \text{CTN}(\text{Bits}(giniIndex))).$$

Afterward, we compare the maxima of each feature with each other. In the code below we show how we compare the features in one split and select the maximum:

```
node['index'] = numpy.identity(m)
for i in range(ceil(log2(m))):
 valueA = node['gini_value']
 valueB = shift(node['gini_value'],-pow(2,i))

 coeffA = node['gini_coeff']
 coeffB = shift(node['gini_coeff'],-pow(2,i))

 indexA = node['index']
 indexB = shift(node['index'],-pow(2,i))

 giniIndexA = valueA * coeffB
 giniIndexB = valueB * coeffA

 mu = giniIndexA > giniIndexB

 node['gini_value'] = mux(valueA,valueB,mu)
 node['gini_coeff'] = mux(coeffA,coeffB,mu)
 node['index'] = mux(indexA,indexB,mu)

node['index'] &= selectMask(0,m)
for i in range(len(node['index'])):
 node['index'][i] = shift(node['index'][i],i)
node['index'] = numpy.sum(node['index'],axis=1)
```

Figure 4.8: Comparing the features in one split and selecting the maximum.

To find the best split amongst our features, we shift and mask our ciphertexts. We give an example on how we process $Gini_{value} \in \mathbb{Z}$ and $Gini_{coeff} \in \mathbb{Z}$ to compare features a_0 and a_1 as well as a_2 and a_3:

Operation	Variable	a_0	a_1	a_2	a_3
Original	$Gini_{value}$	12	22	17	5
	$Gini_{coeff}$	4	7	4	5
Masked	$Gini_{value}$	12	0	17	0
	$Gini_{coeff}$	4	0	4	0
Shifted	$Gini_{value}$	22	17	5	0
	$Gini_{coeff}$	7	4	5	0
Shifted and Masked	$Gini_{value}$	22	0	5	0
	$Gini_{coeff}$	7	0	5	0

.

Now, we compare the highlighted rows as we did before with the first as node A and the second as node B:

Operation	0	1	2	3
$A.Gini_{value} \cdot B.Gini_{coeff} > B.Gini_{value} \cdot A.Gini_{value}$	0	0	1	0

.

Next we take the result of the comparison and use the multiplexer to select either node A or node B resulting into

Operation	Variable	0	1	2	3
$\max(A, B)$	$Gini_{value}$	22	0	17	0
	$Gini_{coeff}$	7	0	4	0

.

Afterward, we repeat the comparison $\lceil log_2(m) \rceil$ times, until only one feature is left.
Up until now, we found the highest Gini Index of all feature, but we lost the information, which feature index it was. To find out the index of the feature we have to remember which feature is bigger than the other. First we generate an identity matrix which represents each feature index. To give an example we consider the values from before were we have $m = 4$ features f to create the indentity matrix I_A and its shifted version I_B:

I_A

f	0	1	2	3
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1

I_B

f	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0

Next, we perform $\text{MUX}(I_A, I_B, A.Gini_{value} \cdot B.Gini_{coeff} > B.Gini_{value} \cdot A.Gini_{value})$ resulting into:

f	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	0	0	1	0
3	0	0	0	0

, which shows us that feature a_1 is bigger than a_0 as well as a_2 is bigger than a_3. After we repeat the comparison $\lceil log_2(m) \rceil$ times, we end up with a single one in the most left column:

f	0	1	2	3
0	0	0	0	0
1	1	0	0	0
2	0	0	0	0
3	0	0	0	0

, indicating that feature 1 is the one with the highest Gini Index. To convert it back to a single ciphertext we shift each row by f:

f	0	1	2	3
0	0	0	0	0
1	0	1	0	0
2	0	0	0	0
3	0	0	0	0

.

Then we perform an XOR-Operation over all of the rows, to get the feature index ciphertext $(0,1,0,0)$. Since we do know all the maximum values of all included variables, we give the estimate for the multiplicative depth right away:

$$\lceil log_2(m) \rceil \cdot (1 + \text{MTN}(\text{Bits}(node['gini\_value'])) + \text{CTN}(\text{Bits}(giniIndex)))$$

Now, our split contains two subsets $left, right \in \mathbb{Z}_2^{\lceil \frac{n}{spc} \rceil \times (m \cdot spc)}$, which both hold m different possible splits. The biggest feature index allows us to select the split belonging to the index.
First, we perform an AND-operation between the biggest feature index and each row of both subsets. Afterward, we copy this result to all other feature indices. The code below illustrates the procedure in python:

```
for g in ['left','right']:
 node[g]['group'] &=  numpy.concatenate([node['index']]*spc)
 for i in range(ceil(log2(m))):
  groupA = node[g]['group']
  groupB = shift(node[g]['group'],-pow(2,i))
  groupA &= shiftMask(i,m,spc)
  groupB &= shiftMask(i,m,spc)

  node[g]['group'] = groupA ^ groupB

  groupA = node[g]['group']
  groupB = shift(node[g]['group'],pow(2,i))

  node[g]['group'] = groupA ^ groupB
```

Figure 4.9: Propagating the biggest feature index onto the split.

This step takes $\lceil \log_2 m \rceil \cdot 2 + 1$ multiplications. Table 4.9 illustrates the sum of all consecutive multiplications.

Name	Multiplicative depth
$\textsc{lvl}(compare())$	$\lceil \log_2 n \rceil \cdot (\textsc{mtn}(\textsc{Bits}(node['gini\_value'])) + \textsc{ctn}(\textsc{Bits}(giniIndex))) +$ $\lceil \log_2 m \rceil \cdot (\textsc{mtn}(\textsc{Bits}(node['gini\_value'])) + \textsc{ctn}(\textsc{Bits}(giniIndex)) + 1) +$ $\lceil \log_2 m \rceil \cdot 2 + 1$

Table 4.9: Multiplicative depth of Gini Index calculation.

4.7 Bootstrapping

In the BGV scheme bootstrapping allows us to refresh the multiplicative depth of a ciphertext. Therefore, we bootstrap our data occasionally to reduce the overall amount of consecutive multiplications needed. The reduction of multiplications requires a smaller parameter set which increases the overall performance. If we bootstrap too often the performance gain is lost, due to the intensive cost of the operation itself.
First we want to give an example, where bootstrapping decreases the amount of multiplications drastically. The function $c = \max\{a, b\}$, with $a, b, c \in R_q^n$ and n denoting the number of bits, will be used in our scheme quite often and has the multiplicative depth of $\textsc{ctn}(\textsc{Bits}(n)) + 1$.

```
Ctxt mu(pubKey), ni(pubKey);
compareTwoNumbers(mu, ni, CtPtrs_VecCt(a), CtPtrs_VecCt(b));
pubKey.thinReCrypt(mu);

HE::NMUX(c, CtPtrs_VecCt(a), CtPtrs_VecCt(b), mu);
```

While the function itself needs a multiplicative depth of $\textsc{ctn}(\textsc{Bits}(n))$, the resulting ciphertext c looses only one. It is worth mentioning that this approach only works if we assume circular security. Otherwise, the bootstrapped ciphertext $mu \in R_q$ and a, b would be encrypted under different keys. The amount of bits n does not influence the cost of bootstrapping, because we bootstrap mu which consists of only one bit.
Before we proceed to split the children of the current node, we want to bootstrap them. By doing so, we reduce the overall multiplicative depth required to the amount that one split needs. To accomplish this goal, we look at the input and output of one split, illustrated in Table 4.4. Here we can see that we have to re-crypt $node['group']$ and $node['lable']$ for each of the two sub-nodes $left, right$.
At last, we bootstrap some intermediate results of all functions to align the number of multiplications needed for all of them. In Table 4.10, we can see the result of this optimization.

Name	LVL needed
$\textsc{lvl}(split\_by\_value())$	$\lceil \log_2(bpf + 2) \rceil$
$\textsc{lvl}(gini\_index()())$	$\textsc{mtn}(\textsc{Bits}(\max\{n^2, \lfloor \frac{2n}{3} \rceil^2 \cdot \lfloor \frac{n}{3} \rceil\}))$
$\textsc{lvl}(compare())$	$\textsc{mtn}(\textsc{Bits}(\max\{n^2, \lceil \frac{n}{2} \rceil^2 \cdot \lfloor \frac{n}{2} \rfloor\}))$

Table 4.10: Reduction of the required multiplicative depth through bootstrapping.

Therefore we need at most a multiplicative depth of $\textsc{lvl}(compare())$ to perform a whole split. Since we bootstrap after each split as well, we can calculate the required multiplications before training. It is vital to know the multiplicative depth beforehand because it defines the size of the parameter set.

4.8 Benchmarks

In this section we give benchmarks to all of our proposed improvements. All of this calculations are performed on a server with 2 Xeon E5-2699v4 processors with 22 times 2.2 GHz cores each. We used 22 of these cores with 200 GB DDR4 for each run.

We listed the parameter sets used for the different dataset and security levels in Table 4.11.

Security	Dataset	m	nBits
20	Tennis	21845	527
	Iris	18631	578
	Balance	18631	578
80	Tennis	28679	527
	Iris	27311	578
	Balance	27311	578

Table 4.11: Parameters for HElib to perform training in 20 and 80-bit security for all datasets.

The integer m defines the ring R_m with the irreducible polynomial $\Phi_m(x)$ and the *nbits* variable describes our multiplicative depth in form of bits.

4.8.1 Impact of Bits per Feature

Here we look at the tests where we increased the bpf. We made the statement that there is no performance loss from preprocessing. We exchange some values within all three datasets to increase the bpf to 16 and 32 bits.

Bits	Tennis	Iris	Balance
Original	1.71	16.72	12.81
16	1.71	19.12	13.06
32	1.72	20.21	13.73

Table 4.12: Timings for different bits per feature with 20-bit security in hours.

Table 4.12 shows that even though we increase the amount of bits per feature drastically the time for a split does not. Therefore, we determined that the bpf do not play a big role in the performance.

4.8.2 Impact of Packing Approaches

Now, we show how the different packing scenarios impact the mean time to find the best split for one node in a Decision Tree.

Security	Packing Type	Tennis			Iris			Balance		
		$\|C\|$	spc	$split_{mean}$ [h]	$\|C\|$	spc	$split_{mean}$ [h]	$\|C\|$	spc	$split_{mean}$ [h]
20	$spc = 1$	14	1	1.75	150	1	13.06	625	1	26.60
	$spc = \lfloor \sqrt[2]{\frac{n}{m}} \rfloor$	14	1	1.74	25	6	8.83	53	12	8.67
	$spc = \lfloor \sqrt[2]{n} \rfloor$	5	3	1.52	13	12	8.64	25	25	8.07
	$spc = spc_{max}$	1	14	1.41	1	150	10.18	3	256	10.77
80	$spc = 1$	14	1	2.36	150	1	20.95	625	1	42.55
	$spc = \lfloor \sqrt[2]{\frac{n}{m}} \rfloor$	14	1	2.46	25	6	15.20	53	12	13.41
	$spc = \lfloor \sqrt[2]{n} \rfloor$	5	3	2.06	13	12	15.39	25	25	12.31
	$spc = spc_{max}$	1	14	1.93	1	150	21.47	3	300	15.45

Table 4.13: Timings for packing approaches in hours with $|C|$ determining the number of ciphertexts.

In the Table 4.13 we can see that packing gives us indeed a boost in performance even though the density shouldn't be maxed out. The highlighted rows indicate the best result with $spc = \lfloor \sqrt[2]{n} \rfloor$. In the tests we used the *distinct-values-approach*.

4.8.3 Impact of Split Values Approaches

Next we look at the impact of the the two optimization methods.

Security	Type	Tennis		Iris		Balance	
		$\|S\|$	$split_{mean}$ [h]	$\|S\|$	$split_{mean}$ [h]	$\|S\|$	$split_{mean}$ [h]
20	*bpf-approach*	2	1.53	128	19.77	8	9.49
	distinct-values-approach	2	1.52	43	8.64	5	8.07
80	*bpf-approach*	2	2.07	128	39.75	8	13.62
	distinct-values-approach	2	2.06	43	15.39	5	12.31

Table 4.14: Timings for optimization methods in hours with $|S|$ denoting the amount of split values.

In Table 4.14 we can see how the different approaches perform. The tests used the packing scenario in which $spc = \lfloor \sqrt[2]{n} \rfloor$ holds. As long as the dataset does not include many different values, the *distinct-values-approach*

will increase the performance drastically. Even though the *bpf-approach* does not increase the performance as well, it can be applied even without preprocessing, which reduces the amount of work on the client side.
In summary we can say that the *bpf-approach* is more general and can be applied to any encrypted data while the *distinct-values-approach* performs better in some scenarios even though we have to preprocess our data.

4.8.4 Multi-threading

First, we want to look at the multi-threading support of HElib. The library uses the multi-threading features implemented in NTL [32] to parallelize the code. The binary operations, provided by HElib, make use of multi-threading by calculating multiple bits in parallel. In Table 4.15, we tested with different amounts of bits and threads. Next we show the gain in speed with multi threading provided by the HElib out of the box.

Bits	Threads	20				80			
		atn	mtn	ctn	amn	atn	mtn	ctn	amn
4	1	1.6	6.4	1.5	3.3	2.1	8.4	1.9	4.5
	2	1.3	4.4	0.8	2.5	1.8	6.0	1.0	3.4
	4	1.1	3.0	0.5	2.2	1.6	4.1	0.8	3.0
	8	0.8	1.8	0.5	1.7	1.1	2.4	0.7	2.1
8	1	5.1	28.6	3.0	9.4	7.4	40.4	4.3	12.8
	2	4.6	22.4	1.8	7.4	6.6	32.1	2.6	9.7
	4	3.4	14.3	1.3	5.5	4.8	20.4	1.9	7.6
	8	2.9	9.4	1.2	3.6	4.3	13.5	1.7	4.8
	16	1.8	6.1	1.2	2.5	2.5	9.0	1.7	3.4
	32	1.7	5.5	1.2	2.5	2.5	8.3	1.8	3.5
16	1	21.5	138.7	7.8	28.2	29.5	195.7	10.5	38.8
	2	18.5	104.1	4.7	22.1	29.6	172.4	7.4	36.2
	4	12.8	66.1	3.3	19.6	16.9	91.4	4.4	28.1
	8	8.1	39.1	2.7	12.7	10.6	54.6	3.7	17.9
	16	6.2	25.5	2.5	8.4	8.0	37.6	3.5	11.8
	32	4.0	16.5	2.5	5.7	5.2	23.1	3.6	8.6
	64	3.9	15.7	2.5	5.8	5.1	21.7	3.7	7.9

Table 4.15: Testing HElibs multi-threading capabilities with increasing threads and bits per feature.

Table 4.15 shows the four binary operations addTwoNumbers(atn), multTwoNumbers(mtn), addManyNumbers(amn) and compareTwoNumbers(ctn). We tested 4, 8 and 16-bit values and increased the number of threads until there was no further improvement.
The results show that more bits per value indicate more performance gain through multi-threading. Unfortunately, we had to resign on that approach because the HElib kept failing with a segmentation fault [33]. We isolated the Helib error from our implementation and found that the failure occurs within the operation AddManyNumbers. Also, the rate of errors increases with the number of threads used.
From this point on, we present our multi-threading approach. The Random Forest learning algorithm generates a large variety of Decision Trees by sampling random sub-samples, as shown in Figure 4.10. Each Decision Tree consists of independent parameters and datasets which allow us to run

each construction simultaneously.

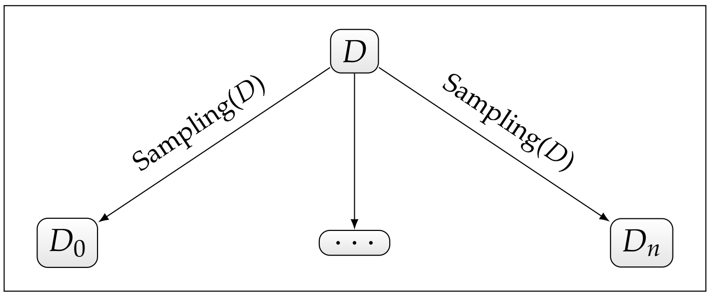

Figure 4.10: Opportunity for multi-threading while sampling new datasets for the Decision Trees in a Random Forest.

Table 4.10 illustrates how we split our dataset into multiple datasets, which can be used to train multiple Decision Trees in parallel.
Another opportunity for multi-threading lies within training of a Decision Trees. For each split we calculate multiple Gini Indices. Since those calculations are independent of each other, we can execute them simultaneously. We generate one Gini Index for each split value s_i contained within the current dataset D_i as shown in Figure 4.11.

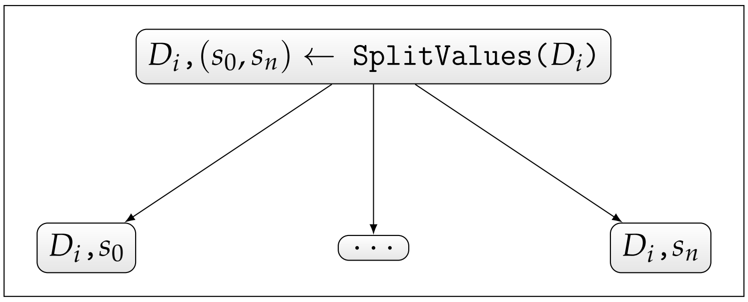

Figure 4.11: Opportunity for multi-threading while generating possible splits for a node.

After finishing two Gini Indices, we can compare them within another thread. We compare all results hirarchically until we end up with the biggest Gini Index.

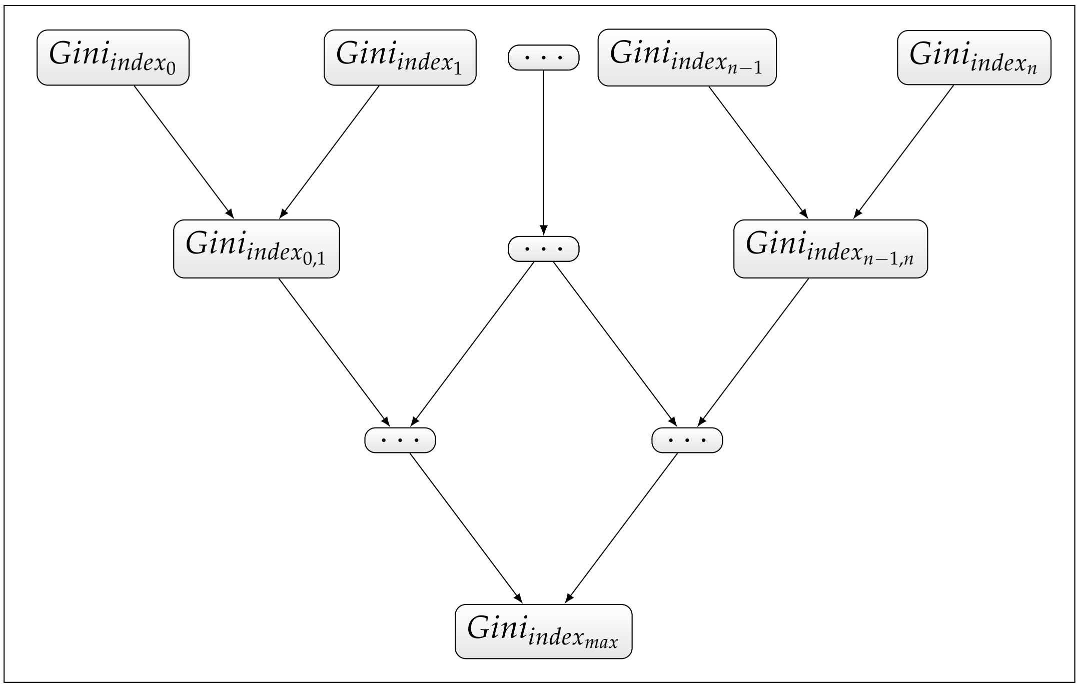

Figure 4.12: Multi threading in hierarchical comparing.

Figure 4.12 shows that in the first level of comparisons we can use $\frac{n}{2}$ threads, while the last level only allows one.
We also tried to combine the multi threading capabilities of HElib and our approach. Even though the process keeps failing a lot we where able to gather benchmarks which show the combined performance. This shows the potential of the scheme if the library is fixed. In our tests we could only test up to 4 threads because the errors increase with the number of threads used. Also 80 bit security produced too much errors, therefore we tested this approach with 20 bit security only.

Security	Threads	Tennis	Iris	Balance
20	1	1.52	8.64	8.07
	2	1.14	7.75	6.30
	4	0.97	6.80	4.86
80	1	2.06	15.39	12.31
	2	1.55	13.80	9.60
	4	1.32	12.11	7.40

Table 4.16: Mean time of a split with increasing number of threads in hours.

All in all, 4.16 shows that fixing the error within the library would make our scheme much more performant. Still, we can say that our approach is highly parallelizable.

4.8.5 Timings for Random Forest

Now, we estimate how long it would take us to train a whole Random Forest. In the scikit-learn Random Forest Classifier they use 100 trees as default for their estimator. If we assume 80-bit security and take the balance dataset, with 625 rows and 4 features, the $split_{mean}$ takes us around 12 hours. Therefore, a Decision Tree of maximal depth 4 would take us 3.5 days. For a Random Forest with 100 trees it would take us 350 days.
If we assumed the HElib to be fixed we can reduce to 8 hours for a mean split, 2.3 days for a Decision Tree and 234 days for the whole Random Forest.

4.9 Correctness

We decided to check the accuracy of our learning algorithm with an equivalent implementation in Python on unencrypted data, because the learning on encrypted data takes far too long. First, we checked if both implementations, the encrypted and the plain, produced the same output for all datasets with a maximum depth up to 3. Normally we could generate the same result every time, but with multi-threading we can not predict the outcome since the order of elements may change. Therefore, we created a python implementation that produces all optimal Decision Trees. Afterward

we checked if the model from the encrypted version is equal to one of the optimal solutions. Our tests indicate that both algorithms produce the same output, since we tested all datasets with many different parameters.
Next we compared the results of our reference [34] and the Scikit-learn [24] implementation to our implementation in python. We performed a cross-validation with randomly generated training and test sets and repeated each constellation of parameters multiple times to form a mean value.

Depth	Dataset	Minimum Sample Size								
		2			3			4		
		[34]	ours	[24]	[34]	ours	[24]	[34]	ours	[24]
2	balance	57.5	57.5	57.5	63.5	62.6	62.6	76.9	76.3	76.3
	iris	91.0	91.7	91.7	91.7	95.3	95.3	91.8	95.8	93.9
	tennis	63.3	63.3	63.3	63.3	73.3	69.0	63.3	73.3	68.3
3	balance	57.5	57.5	57.5	63.5	62.6	62.6	76.9	76.3	76.3
	iris	91.0	91.7	91.7	91.7	95.3	95.3	91.1	96.6	95.5
	tennis	63.3	63.3	63.3	63.3	63.3	61.3	63.3	63.3	62.6
4	balance	57.5	57.5	57.5	63.5	62.6	62.6	76.9	76.3	76.3
	iris	91.0	91.7	91.7	91.7	95.3	95.3	91.1	96.6	95.2
	tennis	50.0	56.6	56.6	50.0	56.6	56.6	50.0	56.6	56.6

Table 4.17: Evaluation scores in %.

The results in 4.17 show that all three implementations perform the same which indicates the correctness of our scheme.

5 Conclusion

Today, enormous amounts of data and Machine Learning allow us to predict future events even more accurately. To extract this information one needs the knowledge as well as the computational power to process all that data. Our scheme allows two parties to train a model privately. The customer, owning the training data, acquires a model while keeping its data confidential. The other party possesses the knowledge to create a model, suited for the customer's data, and owns the infrastructure required for the computation.

For this purpose, we implemented a Random Forest classifier, which uses a variety of Decision Trees to predict events through a majority vote. Our C++ algorithm constructs Decision Trees on encrypted data, which keeps the customer's data confidential. The prediction power of our scheme is equal to the newest implementation of scikit-learn.

Even though the fully homomorphic scheme BGV, supported by the HElib, allows only integers, we found a way to support a variety of data types through preprocessing.

The results show that the average time to generate one node of a Decision Tree on a server cluster, for a dataset of 625 samples and 4 features, takes around 12 hours. We estimate that a Random Forest with 100 trees needs about 350 days to construct.

In future work, we would like to improve the distribution on multiple servers in a cluster. The idea is to train one Decision Tree on separate servers, which would improve the overall amount of time drastically.

Another thought concerns the encoding of the data, where we split features with a larger value space into smaller ones to reduce the number of bits needed to encrypt the message.

Also, multikey support in a Fully Homomorphic Encryption scheme, like in [35], could allow us to learn a model in a multi-party computation fashion, where multiple parties contribute data while keeping it secret from the others.

All in all, the computational power required to train our private classifier in a respectable amount of time is relatively high. Still, future work might improve the performance of encrypted learning algorithms to the point where it becomes feasible for usage in practice.

Bibliography

[1] EU. (2016). Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation) (text with eea relevance), [Online]. Available: `https://eur-lex.europa.eu/eli/reg/2016/679/oj` (visited on 09/03/2020) (cit. on pp. iv, 1).

[2] C. Gentry, "A fully homomorphic encryption scheme", `crypto.stanford.edu/craig`, PhD thesis, Stanford University, 2009 (cit. on pp. 2, 6, 13, 14).

[3] L. Breiman, "Random forests", *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001 (cit. on pp. 2, 36).

[4] S. Halevi and V. Shoup, *An implementation of homomorphic encryption*, `https://github.com/homenc/HElib`, Accessed Dezember 2019 (cit. on pp. 2, 13, 20, 21).

[5] J. R. Quinlan, "Induction of decision trees", *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986 (cit. on pp. 2, 24).

[6] A. Akavia, M. Leibovich, Y. S. Resheff, R. Ron, M. Shahar, and M. Vald, "Privacy-preserving decision tree training and prediction against malicious server", *IACR Cryptology ePrint Archive*, vol. 2019, p. 1282, 2019 (cit. on p. 3).

[7] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, "Homomorphic encryption for arithmetic of approximate numbers", in *ASIACRYPT (1)*, ser. Lecture Notes in Computer Science, vol. 10624, Springer, 2017, pp. 409–437 (cit. on pp. 3, 20).

[8] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping", in *ITCS*, ACM, 2012, pp. 309–325 (cit. on pp. 4, 12, 14, 15, 19, 20).

[9] T. E. Gamal, “A public key cryptosystem and a signature scheme based on discrete logarithms”, in *CRYPTO*, ser. Lecture Notes in Computer Science, vol. 196, Springer, 1984, pp. 10–18 (cit. on p. 4).

[10] D. E. Lancaster, *TTL Cookbook*. USA: Sams, 1974, ISBN: 0672210355 (cit. on p. 6).

[11] G. J. Massey, “Concerning an alleged sheffer function”, *Notre Dame Journal of Formal Logic*, vol. 16, no. 4, pp. 549–550, 1975 (cit. on p. 6).

[12] O. Regev, “On lattices, learning with errors, random linear codes, and cryptography”, in *STOC*, ACM, 2005, pp. 84–93 (cit. on pp. 7, 8, 10).

[13] V. Lyubashevsky, C. Peikert, and O. Regev, “On ideal lattices and learning with errors over rings”, in *EUROCRYPT*, ser. Lecture Notes in Computer Science, vol. 6110, Springer, 2010, pp. 1–23 (cit. on pp. 7, 9, 10).

[14] A. Blum, A. Kalai, and H. Wasserman, “Noise-tolerant learning, the parity problem, and the statistical query model”, in *STOC*, ACM, 2000, pp. 435–440 (cit. on p. 7).

[15] M. Naehrig, K. E. Lauter, and V. Vaikuntanathan, “Can homomorphic encryption be practical?”, in *CCSW*, ACM, 2011, pp. 113–124 (cit. on p. 11).

[16] Z. Brakerski and V. Vaikuntanathan, “Fully homomorphic encryption from ring-lwe and security for key dependent messages”, in *CRYPTO*, ser. Lecture Notes in Computer Science, vol. 6841, Springer, 2011, pp. 505–524 (cit. on p. 11).

[17] Z. Brakerski and V. Vaikuntanathan, “Efficient fully homomorphic encryption from (standard) LWE”, in *FOCS*, IEEE Computer Society, 2011, pp. 97–106 (cit. on pp. 15, 16).

[18] S. Halevi and V. Shoup, “Algorithms in helib”, in *CRYPTO (1)*, ser. Lecture Notes in Computer Science, vol. 8616, Springer, 2014, pp. 554–571 (cit. on p. 20).

[19] J. L. H. Crawford, C. Gentry, S. Halevi, D. Platt, and V. Shoup, “Doing real work with FHE: the case of logistic regression”, in *WAHC@CCS*, ACM, 2018, pp. 1–12 (cit. on p. 21).

[20] R. Townsend. (2020). Tennis dataset, [Online]. Available: `https://github.com/sjwhitworth/golearn/blob/master/examples/datasets/tennis.csv` (visited on 01/15/2020) (cit. on pp. 22, 23, 35, 40).

[21] L. Rokach and O. Maimon, *Data Mining with Decision Trees - Theory and Applications. 2nd Edition*, ser. Series in Machine Perception and Artificial Intelligence. WorldScientific, 2014, vol. 81 (cit. on pp. 25–27, 29).

[22] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993 (cit. on p. 26).

[23] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: Experiences from the scikit-learn project", *CoRR*, vol. abs/1309.0238, 2013 (cit. on pp. 26, 38).

[24] scikit-learn 0.22.1 documentation. (2020). Sklearn.tree.decisiontreeclassifier, [Online]. Available: `https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html` (visited on 01/20/2020) (cit. on pp. 29, 65).

[25] Y. Chen and G. Gong, "Integer arithmetic over ciphertext and homomorphic data aggregation", in *CNS*, IEEE, 2015, pp. 628–632 (cit. on p. 30).

[26] M. Douglas C., P. Elizabeth A., and V. G. Geoffrey, "Introduction to linear regression analysis", *Biometrics*, vol. 69, no. 4, pp. 1087–1087, 2013 (cit. on p. 34).

[27] scikit-learn 0.22.1 documentation. (2020). Sklearn.preprocessing.onehotencoder, [Online]. Available: `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html` (visited on 02/10/2020) (cit. on p. 34).

[28] A. Cutler, D. Cutler, and J. Stevens, "Random forests", in. Jan. 2011, vol. 45, pp. 157–176. DOI: `10.1007/978-1-4419-9326-7_5` (cit. on p. 35).

[29] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation", *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 22–30, 2011 (cit. on p. 38).

[30] R. Fisher. (1988). Iris dataset, [Online]. Available: `http://archive.ics.uci.edu/ml/datasets/Iris` (visited on 03/15/2020) (cit. on p. 40).

[31] R. S. Siegler. (1976). Balance scale dataset, [Online]. Available: `https://archive.ics.uci.edu/ml/datasets/Balance+Scale` (visited on 03/15/2020) (cit. on p. 40).

[32] V. Shoup. (2020). Ntl: A library for doing number theory, [Online]. Available: `https://www.shoup.net/ntl/` (visited on 09/02/2020) (cit. on p. 60).

[33] P. Schwarz. (2020). Binaryarithmetic: Segmentation fault, [Online]. Available: `https://github.com/homenc/HElib/issues/354` (visited on 05/19/2020) (cit. on p. 61).

[34] J. Brownlee. (2020). How to implement the decision tree algorithm from scratch in python, [Online]. Available: `https://machinelearningmastery.com/implement-decision-tree-algorithm-scratch-python/` (visited on 02/05/2020) (cit. on p. 65).

[35] A. López-Alt, E. Tromer, and V. Vaikuntanathan, "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption", in *STOC*, ACM, 2012, pp. 1219–1234 (cit. on p. 66).