ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA MẠNG MÁY TÍNH VÀ TRUYỀN THÔNG

LƯU GIA HUY

NGUYỄN VĂN KHANG KIM

BÁO CÁO ĐỒ ÁN MÔN HỌC

KẾT HỢP MÃ HÓA ĐỒNG CẤU VÀ MÁY HỌC TRONG PHÂN TÍCH KHẢ NĂNG MẮC BỆNH TIM

Combining fully homomorphic encryption and machine learning for cardiovascular disease prediction

TP. HỒ CHÍ MINH, 2023

ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

KHOA MẠNG MÁY TÍNH VÀ TRUYỀN THÔNG

LƯU GIA HUY – 21520916

NGUYỄN VĂN KHANG KIM – 21520314

BÁO CÁO ĐỒ ÁN MÔN HỌC

KẾT HỢP MÃ HÓA ĐỒNG CẤU VÀ MÁY HỌC TRONG PHÂN TÍCH KHẢ NĂNG MẮC BỆNH TIM

Combining fully homomorphic encryption and machine learning for cardiovascular disease prediction

GIẢNG VIÊN HƯỚNG DẪN

TS.NGUYỄN NGỌC TỰ

TP. HỒ CHÍ MINH, 2023

Mục lục

ANALYZING SCENARIO, ISSUES

1.1. Scenario

1.1.1. Reason for choosing this topic

Currently, heart disease is a significant and prevalent health issue worldwide. According to statistics from the World Health Organization (WHO), cardiovascular diseases are the leading cause of global mortality. Early detection and analysis of the risk of heart disease can play a crucial role in prevention, diagnosis, and treatment.

Combining homomorphic encryption and machine learning is a promising approach to address the issue of analyzing the risk of heart disease. Here are some reasons for choosing this topic:

- Privacy preservation: Homomorphic encryption allows for data analysis without revealing personal information of patients. This ensures privacy and compliance with healthcare data security regulations.
- Enhanced analytical capabilities: The combination of homomorphic encryption and machine learning harnesses the computational power of machine learning to effectively analyze heart disease data. Applying complex machine learning algorithms and models on homomorphically encrypted data yields accurate and reliable analyses.
- Potential applications in public health: Analyzing the risk of heart disease can provide crucial information on risk factors and disease prediction. The findings from this research can support public health policy decisions and preventive measures for heart disease.
- Scalability and applicability: Combining homomorphic encryption and machine learning is a highly promising research direction. Homomorphic encryption technology is continually evolving and can be widely applicable in the healthcare field and data analysis.

1.1.2. **Benefit**:

- Enhanced security: Homomorphic encryption helps protect sensitive patient data during the analysis process. Instead of sharing raw data, only the analysis results are shared, reducing the risk of personal information leakage.
- Utilization of data from multiple sources: Combining homomorphic encryption and machine learning allows for the use of data from various sources without revealing personal information. This opens up opportunities for research and analysis on large and diverse datasets, improving the accuracy and reliability of results.
- Ensuring consistency and reliability: Using homomorphic encryption and machine learning ensures consistency throughout the analysis process. By applying the same encryption method and machine learning models to different datasets, reliable and comparable analysis results can be achieved.
- Contributing to personalized medicine development: Analyzing the risk of heart disease based on the combination of homomorphic encryption and machine learning can contribute to the development of personalized medicine. The analysis results can help provide customized treatment recommendations and individual health monitoring, benefiting each patient.

In summary, the combination of homomorphic encryption and machine learning in analyzing the risk of heart disease brings numerous benefits, from protecting privacy and enhancing security to utilizing multi-source data, ensuring consistency and reliability, and contributing to the development of personalized medicine.

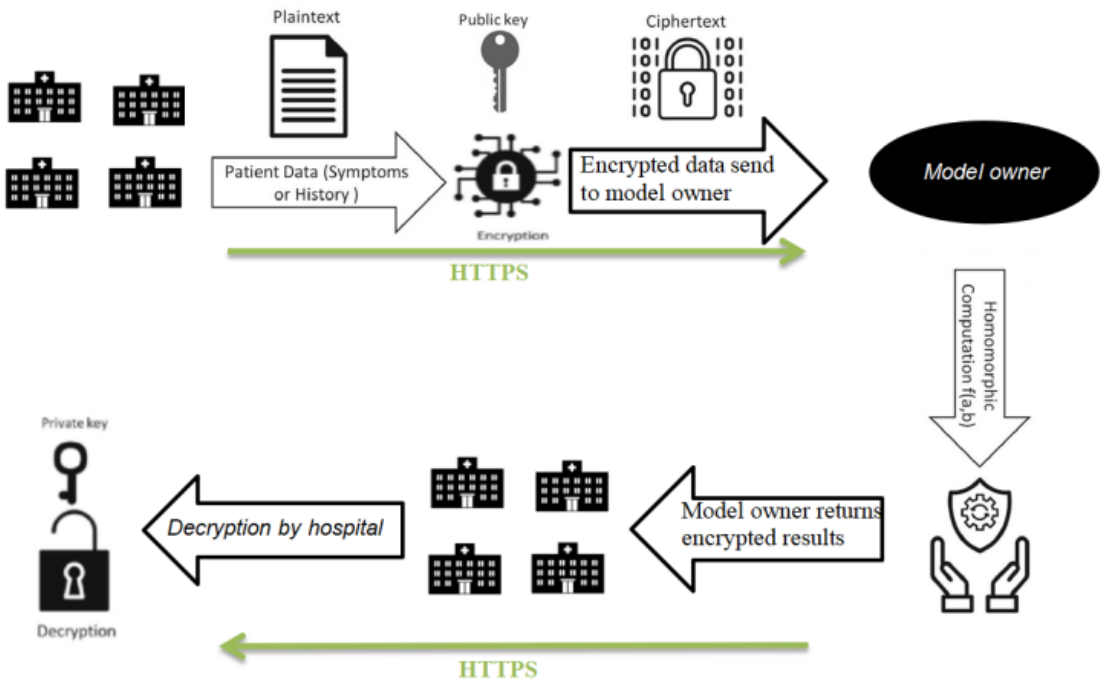
1.1.3. **The application scenario**

In this scenario, hospitals are seeking solutions for early detection and proactive analysis and treatment of heart disease in patients. To accomplish this, they need to utilize existing data, including cases of both diagnosed and non-diagnosed individuals that have been examined and stored in the hospital's database. Computing and making predictions about the likelihood of having the disease based

on this data will assist hospitals in applying the most timely and effective treatment methods for patients.

However, hospitals possess only a localized volume of medical data, primarily at the local or within the same country or region. Furthermore, hospitals may not be able to meet the requirements for fast and efficient computation with the massive amount of data they possess. This is where a third party with comprehensive computational and technical resources needs to be involved, along with a more objective dataset that is not limited to a small geographical area.

1.2. System architecture



1.3. Relevant parties

- Hospitals: providing their data to the third party.
- Third party: possessing computational expertise and excellent resources to return predictions for heart disease.

1.4. Assets to be protected

- Patient data: including indicators, symptoms, conditions, etc.
- Return results: Results calculated and returned by the third party.

1.5. Security risks

- During transmission, data can be stolen by attackers.
- The third party may be untrustworthy and could misuse patient data for unauthorized purposes.

Required security features to be built

- Secure data transmission: Implementing secure protocols, such as SSL/TLS or HTTPS, to encrypt data during transmission, protecting it from unauthorized access, interception, and ensuring the confidentiality and integrity of the data. Note that the implementation details will not be covered in this report.
- Homomorphic encryption: Utilizing homomorphic encryption techniques that allow performing computations on encrypted data without the need for decryption, ensuring privacy while still enabling computational analysis on the encrypted data.

2. SOLUTION DESIGN

2.1. Solution analysis

- Data security during transmission: (Not implemented in this project)
  - Use secure communication protocols such as HTTPS to ensure security during the data transmission between relevant parties.

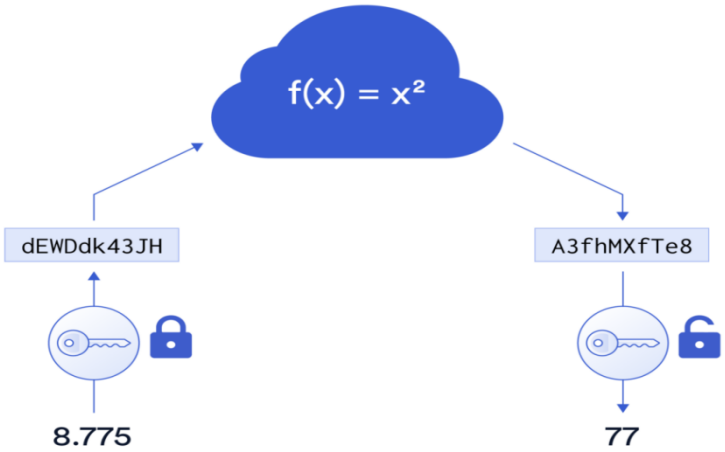
- Use strong encryption technologies like SSL/TLS to protect data from being stolen or tampered with during the transmission.
-

- Data encryption with support for computation on encrypted data:

- Use Fully Homomorphic Encryption (FHE) to encrypt the data. Encrypt the data using the public key and send it to a third party for computation. Upon receiving the result, decrypt it using the private key.

  In this case, I will proceed with scenarios involving encryption using two schemes of Fully Homomorphic Encryption (FHE): CKKS and BGV. I will also use a machine learning model such as Random Forest. Finally, I will provide an analysis and performance comparison.

2.2. Fully Homomorphic Encryption



Fully Homomorphic Encryption (FHE) is a concept in the field of encryption that allows performing computations on encrypted data without the need to decrypt the data. This means that operations can be performed on encrypted data, resulting in encrypted data corresponding to the outcome. The final result can be decrypted afterward.

FHE enables secure computation on sensitive data while maintaining privacy. It provides the ability to perform operations on encrypted data, preserving the confidentiality of the data throughout the computation process. This is particularly useful in scenarios where data privacy is paramount, such as outsourcing computation to untrusted third parties or performing analysis on sensitive data without exposing the raw information.

EX: $f(r) = 2r$

$$f(x+y) = f(x) + f(y)$$

$$f(-2+5) = f(-2) + f(5)$$

$$f(3) = f(-2) + f(5)$$

$$6 = -4 + 10$$

$$6 = 6$$

FHE allows the same operations to be performed on both encrypted and plaintext data. This means that the result of a computation on encrypted data corresponds to the result of the corresponding operation on the plaintext data.

FHE has numerous important applications, particularly in protecting privacy in computations on sensitive data. Instead of needing to decrypt the data to perform computations, FHE allows parties to perform computations on encrypted data without knowing specific information about the data. This ensures that the data remains undisclosed during the computation process.

However, FHE presents technical challenges, including slow computational speed and high complexity of operations on encrypted data. As a result, FHE is an ongoing area of research, with continuous efforts to improve performance and real-world applicability.

2.2.1. BGV scheme

2.2.1.1. Plaintext and Ciphertext Spaces

Plaintext: $P= R_t = Z_t[x]/(x^n+1)$

Cipher: $C = R_q * R_q$; $R_q = Z_q[x]/(x^n+1)$

For efficiency purposes, n is usually set as a power of 2 integer. q is usually much greater than t, hence, the cardinality of C is much larger than that of P, which also means a plaintext message M in P can be mapped to multiple valid ciphertexts in C.

2.2.1.2. Parameters

R_2 : mostly used in the encryption keys, is a uniform random distribution used to sample polynomials with integer coefficients in {-1,0,1}.

X : is the error distribution defined as a discrete Gaussian distribution

R_q : is a uniform random distribution over R_q for sampling polynomials in {0,1,...,q-1}

2.2.1.3. Plaintext Encoding and Decoding

The integer encoding scheme, works as follow:

1. Represent m in binary representation:

$m = a_{n-1}.....a_2a_1a_0$

2. Compose $M = a_{n-1}x + ...a_2x^2 + a_1x + a_0$

Sincde n is too large in practice, unused bits are set to 0.

Decode: Decoding for the interger encoding scheme works by evaluating the plaintext polynomial at k, computing M(k)

2.2.1.4. **Key Generation**

The secret key SK is a random ternary polynomial that is generated from R_2. The public key PK is a pair of polynomial (PK_1,PK_2) calculated as follow:

$$PK_1 = [-1(a \bullet SK + t \bullet e)]_q$$

$$PK_2 = a$$

Where a is a random polynomial in R_q, and e is a random error polynomial sampled form X. The notation $[\bullet]_q$ means that polynomial arithmetic should be done modulo q.

2.2.1.5. **Encryption and Decryption**

The encryption algorithm takes as input a plaintext message M in P and the public key PK and outputs ciphertext C = (C_1, C_2) in C encrypting the input message as a result. Encryption proceeds as follows: we generate three small random polynomials u form R_2 and e_1, e_2 from X and compute:

$$C_1 = [PK_1 \bullet u + t \bullet e_1 + M\ ]_q$$

$$C_2 = [PK_2 \bullet u + t \bullet e_2\ ]_q$$

Decryption proceeds as follows:

$$M = [[\ C_1 + C_2 \bullet SK]_q]_t$$

2.2.2. **CKKS scheme**

2.2.2.1. **Plaintext and Ciphertext Spaces**

In CKKS, the plaintext and ciphertext spaces are almost the same. They include elements of the polynomial ring $R_q = Z_q[x]/f(x)$, where q is an integer called the coefficient modulus and f(x) is a polynomial known as the polynomial modulus. Elements of R_q are polynomials with integer coefficients bounded by q. Their degrees are also bounded by the degree of f(x). The most common choice of

in the literature is $f(x) = x^n + 1$ with n (known as the ring dimension) being a power of 2 number. The difference between CKKS plaintext and ciphertext instances is the number of ring elements they contain. An instance of CKKS plaintext includes one ring element (polynomial) whereas an instance of CKKS ciphertext includes at least 2 ring elements.

2.2.2.2. **Plaintext Encoding and Decoding**

Another main contribution of the CKKS work is anew method that maps a vector of complex numbers into a single plaintext object and vice versa. Encoding works as follows, given an $\frac{n}{2}$ vector of complex numbers $z \in C^{n/2}$, return a single plaintext element $a \in R$. Decoding does and return a vector of complex numbers. The map Δ is the complex canonical embedding which is a variant of the Fourier transform.

$$\mathsf{ENCODE}(z, \Delta) = \lfloor \Delta \cdot \pi^{-1}(z) \rceil$$
$$\mathsf{DECODE}(a, \Delta) = \pi(\frac{1}{\Delta} \cdot a)$$

2.2.2.3. **Parameters**

R_2 : mostly used in the encryption keys, is a uniform random distribution used to sample polynomials with integer coefficients in {-1,0,1}.

X : is the error distribution defined as a discrete Gaussian distribution.

R_q : is a uniform random distribution over R_q

2.2.2.4. **Key Generation**

We sample the secret key SK an element form R_2, a polynomial of degree n with coefficients in {-1,0,1}. The public key PK is a pair of polynomials (PK_1 , PK_2) calculated as follows:

$$PK_1 = [-a \bullet SK + e]_q$$

$$PK_2 = a \xleftarrow{u} R_q$$

a is a random polynomial sampled uniformly from R_q, and e is a random error polynomial sampled from R_2. Recall that the notation $[\bullet]_q$ implies that polynomial arithmetic should be done modulo q. Note that as is in R_q, polynomial arithmetic should also be performed modulo the ring polynomial modulus $(x^n + 1)$.

2.2.2.5. Encryption and Decryption

To encrypt a plaintext message M in R(which is an encoding of the input payload vector). We generate 3 small random polynomials u formm R_2 and e_1 and e_2 from X and return the cipher $C = (C_1, C_2)$ in R^2_q as follows:

$$C_1 = [PK_1 \bullet u + e_1 + M]_q$$

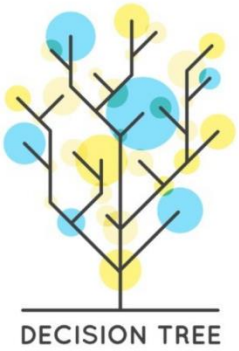
$$C_2 = [PK_2 \bullet u + e_2]_q$$

Decrypt:

$$M = [C_1 + C_2 \bullet SK]_q$$

2.3. Machine Learning Model

2.3.1. Decision tree



2.3.1.1. Definition

- A decision tree is a machine learning model that interprets and represents decisions or predictions using a tree-like structure. It is a supervised learning model widely used in classification and prediction tasks.
- A decision tree is built based on easy-to-understand and applicable decision rules. It consists of nodes in the tree representing decisions or conditions, and branches in the tree representing outcomes or actions. Each node in the tree represents an attribute, and each branch stemming from a node represents a value of that attribute.
- The process of constructing a decision tree starts from the root node and continues downward following the rules and conditions until reaching the leaf nodes, where the final decisions are made. These rules and conditions are learned from training data, where the data samples have been labeled or have known outcomes in advance.

2.3.1.2. Advantages

- The Decision Tree model is simple, intuitive, and easy to understand after a brief explanation.
- Some Decision Tree algorithms can handle missing or erroneous data without the need for methods like "imputing missing values" or data elimination. Additionally, Decision Trees are less affected by outliers.
- This is a nonparametric method, meaning it does not rely on initial assumptions about distributional laws as in statistics. As a result, the analysis results are always objective.
- It provides highly accurate prediction results, is easy to implement, and requires fast training. There is no need to transform variables.

2.3.1.3. Disadvantages

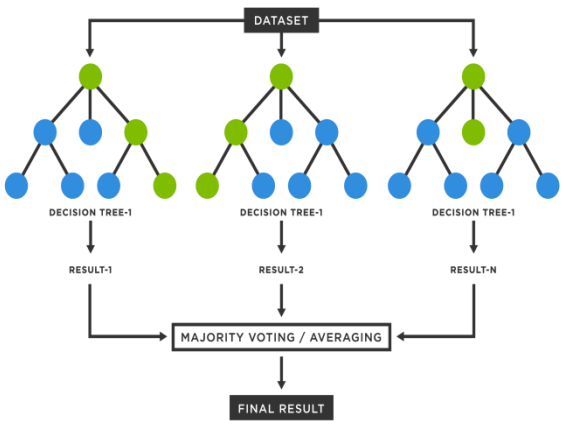
- Decision Trees work effectively on simple datasets with few interrelated variables. When applied to complex datasets with many variables and different attributes, it can

lead to overfitting, where the model becomes too specific to the training data and fails to accurately classify new data.

- Minor changes in the dataset can significantly impact the structure of the model.
- This model has a potential bias if the dataset is imbalanced, only considering typical values, and the risk of underfitting.
- Decision Trees require the training and test datasets to be perfectly prepared, with high-quality and balanced data according to the classes or groups in the target variable.

2.3.2. Random Forest

2.3.2.1. Definition



Random Forest is a machine learning method that combines multiple decision trees to create a more powerful predictive model. Random Forest is used for classification and prediction tasks.

The main idea of Random Forest is to build a collection of independent decision trees and combine the results from all the decision trees to make the final prediction. The process of building a Random Forest involves the following steps:

- Determining the number of decision trees: Decide the number of decision trees that the Random Forest will use. This is a parameter that needs to be specified in advance.
- Building independent decision trees: For each decision tree in the Random Forest, a random subset of the training data is sampled using bootstrap sampling. Then, the decision tree is constructed by dividing the subset of data into groups based on attributes and values.
- Combining results from decision trees: The results from each decision tree are combined to generate the final prediction. For classification problems, a voting method is used, where the prediction is determined by counting the votes for each class. For prediction tasks, the result can be the average or variance of the predictions from the decision trees.

2.3.2.2. Advantages

- Flexibility: Random Forest has the ability to handle complex data and both discrete and continuous attributes.
- Handling missing data: Random Forest can effectively handle missing information in the data.
- Overfitting control: By combining decision trees, Random Forest reduces the risk of overfitting and provides high accuracy on new data.

2.3.2.3. Disadvantages

- Lack of interpretability: As the number of decision trees increases, Random Forest can become difficult to interpret when there are too many trees in the ensemble. Therefore, interpreting and explaining the decisions of the model can become challenging.
- Computational cost: Random Forest requires building and training multiple independent decision trees, so it can be computationally expensive compared to a single decision tree.

3. DEPLOYMENT

3.1. Deployment scenario analysis

With the dataset obtained from Kaggle, we will proceed to convert it to integers using the BGV scheme while keeping the real numbers intact using the CKKS scheme.

Then, the client (hospital) will encrypt the data with the corresponding scheme, label the original data, and send it to the server (model owner).

On the server side, the labeled data will be used for training and testing. Then, the labels will be referenced to the corresponding encrypted values using the public key to perform computations.

The encrypted computation results will be sent back to the client, who will use the private key to decrypt them.

Here, I will proceed with two deployment scenarios:

- Random Forest with the CKKS scheme
- Random Forest with the BGV scheme

3.2. Resource usage

3.2.1. Machine Specifications

Utilize a virtual machine running Ubuntu 22.04 with the following specifications:

- Memory: 4GB
- Processors: 8
- Hard Disk (SCSI): 50GB

3.2.2. Data usage

Dataset: https://www.kaggle.com/datasets/sid321axn/heart-statlog-cleveland-hungary-final

The dataset consists of 1190 records of patients from US, UK, Switzerland and Hungary. It has 11 features and 1 target variable.This dataset consists of 11 features and a target variable. It has 6 nominal variables and 5 numeric variables. The detailed description of all the features are as follows:

1. **Age**: Patients Age in years (Numeric)

2. **Sex**: Gender of patient (Male - 1, Female - 0) (Nominal)

3. **Chest Pain Type**: Type of chest pain experienced by patient categorized into 1 typical, 2 typical angina, 3 non- anginal pain, 4 asymptomatic (Nominal)

4. **resting bp s**: Level of blood pressure at resting mode in mm/HG (Numerical)

5. **cholestrol**: Serum cholestrol in mg/dl (Numeric)

6. **fasting blood sugar**: Blood sugar levels on fasting > 120 mg/dl represents as 1 in case of true and 0 as false (Nominal)

7. **resting ecg**: Result of electrocardiogram while at rest are represented in 3 distinct values 0 : Normal 1: Abnormality in ST-T wave 2: Left ventricular hypertrophy (Nominal)

8. **max heart rate**: Maximum heart rate achieved (Numeric)

9. **exercise angina**: Angina induced by exercise 0 depicting NO 1 depicting Yes (Nominal)

10. **oldpeak**: Exercise induced ST-depression in comparison with the state of rest (Numeric)

11. **ST slope**: ST segment measured in terms of slope during peak exercise 0: Normal 1: Upsloping 2: Flat 3: Downsloping (Nominal)

12. **target**: It is the target variable which we have to predict 1 means patient is suffering from heart risk and 0 means patient is normal. (*Target variable*)

3.3. Data proccessing steps

We will perform two main steps in this stage: data labeling and data encryption.

Data labeling: We will assign labels to the data points in the original dataset. This process may involve determining classes/groups and labeling the data points accordingly. For example, assigning label 0 for Class A, label 1 for Class B, and so on.

Data encryption: After labeling, we will proceed to encrypt the original data values. Depending on the chosen scheme (CKKS or BGV), we will apply the corresponding encryption methods to protect the data. Encryption may involve applying specific encryption operations and algorithms to transform the data values into an encrypted form that can only be decrypted by the owner's corresponding private key.

Age	Sex	...	Label
60	1	...	1
...	...	...	...
75	0	...	0

A	B	.....	Z
enc(60)	enc(1)	.....	enc(1)
...	...	...	...
enc(75)	enc(0)	...	enc(0)

A	B	...	Z
A60	B1	...	Z1
...	...	...	...
A75	B0	...	Z0

3.4. Random forest with CKKS scheme

3.4.1. Initializing Parameters

```
uint32_t multDepth = 8;
uint32_t scaleModSize = 50;
uint32_t batchSize = 1;

CCParams<CryptoContextCKKSRNS> parameters;
CryptoContext<DCRTPoly> cc;
KeyPair<DCRTPoly> keys;
```

```
auto start = std::chrono::high_resolution_clock::now();
parameters.SetMultiplicativeDepth(multDepth);
parameters.SetScalingModSize(scaleModSize);
parameters.SetBatchSize(batchSize);
cc = GenCryptoContext(parameters);

cc->Enable(PKE);
cc->Enable(KEYSWITCH);
cc->Enable(LEVELEDSHE);
std::cout << "CKKS scheme is using ring dimension " << cc->GetRingDimension() << std::endl << std::endl;

keys = cc->KeyGen();
std::cout << "Public key: " << keys.publicKey << std::endl;
std::cout << "Secret key: " << keys.secretKey << std::endl;
```

We are considering setting up the CKKS (Cryptographically Secure Keyed Sum) cryptographic environment. The first step in this process is to establish the CryptoContext. We need to determine the main parameters.

- **multDepth**

  One of these parameters is the multiplicative depth. Multiplicative depth is an important parameter in the CKKS model, determining the maximum depth of multiplication operations in a computation chain. It is not the total number of multiplications supported by the model, but rather a limit for a specific multiplication operation.

  For example, let's consider a computation $f(x, y) = x^2 + xy + y^2 + x + y$. This computation has a multiplicative depth of 1 but requires a total of 3 multiplications to be performed. In another case, we have a computation $g(x\_i) = x_1x_2x_3x_4$. This computation can be performed with a multiplicative depth of 3, such as $g(x\_i) = ((x_1x_2)x_3)x_4$, or with a multiplicative depth of 2, such as $g(x\_i) = (x_1x_2)(x_3x_4)$.

Choosing the appropriate multiplicative depth is crucial to ensure the efficiency and accuracy of the computation. The multiplicative depth in the CKKS model affects the accuracy of the computation results. As the multiplicative depth increases, allowing more multiplications to be performed in a computation chain before refreshing is required, the accuracy of the computation results also increases.

Refreshing is:

- In the CKKS (Cryptographically Secure Keyed Sum) model, "refreshing" is the necessary process to maintain the accuracy of the computation results. When computing in the CKKS model, multiplication operations lead to a loss of accuracy over time due to accumulated errors.
- To overcome this, the "refreshing" process needs to be performed after a specified multiplicative depth. In this process, values are normalized and adjusted to rebalance the accuracy. Refreshing may involve techniques such as noise redistribution or increasing the resolution.
- The refreshing process is important to ensure that the computation results in the CKKS model remain reliable and accurate after each computation chain. If refreshing is not performed correctly, accumulated errors can increase and lead to inaccurate results.

- **scaleModSize**

- In the CKKS model, real numbers are encoded as integers, and the scaling factor plays an important role in this process. The scaling factor is a parameter of the model and is used to encode real numbers as integers.
- The 'scaleModSize' parameter in CKKS determines the bit length of the scaling factor D, however, it does not directly determine the exact scaling factor. The exact scaling factor is determined by the specific implementation and may differ between ciphertexts in certain versions of CKKS (e.g., FLEXIBLEAUTO).
- The choice of the 'scaleModSize' value depends on the desired accuracy of the computation, as well as other parameters such as the multiplicative depth or security

requirements. These parameters determine the level of noise generated during the computation because the CKKS model is an approximate model, and each operation introduces a small amount of noise.

- The scaling factor should have a sufficiently large length to accommodate the noise and support computation results with the desired accuracy. Choosing the correct 'scaleModSize' is an important factor in ensuring the accuracy of computation results in the CKKS model.

- **batchSize**

- In the CKKS model, multiple plaintext values are packed into each ciphertext. The maximum number of slots depends on a security parameter called the ring dimension.
- In this case, we don't directly specify the ring dimension, but rather let the library choose it based on the security level we select, the desired multiplicative depth, and the size of the scaling factor.
- We can use the method GetRingDimension() to determine the exact ring dimension being used for these parameters. With a ring dimension of N, the maximum batch size is N/2, due to the way CKKS operates.

Implementation code

Function encCKKS(double x); for encryption using the CKKS scheme:

```
Ciphertext<DCRTPoly> encCKKS(double x)
{
        vector<double> r;
        r.push_back(x);
        Plaintext ptxt = cc->MakeCKKSPackedPlaintext(r);
        auto c = cc->Encrypt(keys.publicKey, ptxt);
        r.clear();
        return c;

}
```

Here is the code segment that maps the target labels to their corresponding values and stores them in the 'res' vector of type 'Ciphertext<DCRTPoly>>:

```
vector<Ciphertext<DCRTPoly>> res;
class OutputPrinter {
        private:
                ofstream fout;
        public:
                OutputPrinter(string filename) {
                        fout.open(filename,ios::app);
                        if(!fout) {
                                cout << filename << " file could not be opened\n";
                                exit(0);
                        }
                }

                string joinByTab(vector<string> row) {
                        for (const auto& labelValue : labelValueMap) {
                        if (labelValue.first==row[static_cast<int>(row.size()-1)])
                        {
                                res.push_back(labelValue.second);
                        }
                }
                        return row[static_cast<int>(row.size()-1)];
                }
                void addLine(string str) {
                        fout << str << endl;
                }
};
```

Here is the code segment for performing voting, which involves calculating the average of the output values from the decision trees, decrypting them, decoding, extracting the real part, and displaying it on the screen:

```
Plaintext result;
std::vector<std::complex<double>> finalResult;
if(static_cast<int>(res.size())!=0)
{
        Ciphertext<DCRTPoly> avg = res[0];
        for(int i=1;i < static_cast<int>(res.size()); i++) {
        avg = cc->EvalAdd(avg, res[i]);

        }
        avg=cc->EvalMult(avg, (1/double(static_cast<int>(res.size()))));
        cc->Decrypt(keys.secretKey, avg, &result);
        result->SetLength(batchSize);
        finalResult = result->GetCKKSPackedValue();
        std::cout << "Risk of heart disease with Random Forest_CKKS scheme: " << finalResult[0].real() << std::endl;
}
else
cout<<"res.size()==0....";
```

Result

In the case where the target is 0, after the computation, we obtain a probability of returning 0.4

```
ST slope = K2, fasting blood sugar = F0, cholesterol = E70, Label: L1
ST slope = K2, fasting blood sugar = F0, cholesterol = E80, Label: L1
ST slope = K2, fasting blood sugar = F1, Label: L1
ST slope = K3, exercise angina = I0, Label: L0
ST slope = K3, exercise angina = I1, Label: L1
<-- finish generating decision tree -->

Risk of heart disease with Random Forest_CKKS scheme: 0.4
Run time: 6205.93s

~/Downloads/openfhe-development/build/bin/examples/pke  main !6 ?9
```

In the case where the target is 1, after the computation, we obtain a probability of returning 0.8

```
ST slope = K2, fasting blood sugar = F0, cholesterol = E70, Label: L1
ST slope = K2, fasting blood sugar = F0, cholesterol = E80, Label: L1
ST slope = K2, fasting blood sugar = F1, Label: L1
ST slope = K3, exercise angina = I0, Label: L0
ST slope = K3, exercise angina = I1, Label: L1
<-- finish generating decision tree -->

Risk of heart disease with Random Forest_CKKS scheme: 0.8
Run time: 5781.97s

~/Downloads/openfhe-development/build/bin/examples/pke  main !6 ?9
```

3.4.2. Completion time

- The completion time for the case where the target is 0 is: 6205.93 seconds, equivalent to 1 hour and 44 minutes.
- The completion time for the case where the target is 1 is: 5781.97 seconds, equivalent to 1 hour and 36 minutes.

3.5. Random forest with BGV scheme

3.5.1. Initializing Parameters

```
CCParams<CryptoContextBGVRNS> parametersBGV;
CryptoContext<DCRTPoly> cryptoContextBGV;
KeyPair<DCRTPoly> keyPairBGV;
```

```
parametersBGV.SetMultiplicativeDepth(8);
parametersBGV.SetPlaintextModulus(65537);
parametersBGV.SetBatchSize(1);
cryptoContextBGV = GenCryptoContext(parametersBGV);
std::cout << "BGV scheme is using ring dimension " << cryptoContextBGV->GetRingDimension() << std::endl << std::endl;

cryptoContextBGV->Enable(PKE);
cryptoContextBGV->Enable(KEYSWITCH);
cryptoContextBGV->Enable(LEVELEDSHE);
keyPairBGV = cryptoContextBGV->KeyGen();


std::cout << "Public key BGV: " << keyPairBGV.publicKey << std::endl;
std::cout << "Secret key BGV: " << keyPairBGV.secretKey << std::endl;
```

parameters.SetMultiplicativeDepth(8);: This is the command to set the multiplicative depth of the BGV model. The multiplicative depth determines the number of multiplication layers that can be performed in the computation. In this case, a value of 8 for the multiplicative depth allows for a maximum of 8 multiplication layers in the computation.

parameters.SetPlaintextModulus(65537);: This is the command to set the plaintext modulus in the BGV model. The plaintext modulus is a large integer used to encode plaintext values in the BGV model. In this case, the value 65537 is chosen as the plaintext modulus. The choice of the plaintext modulus depends on the desired accuracy and computational requirements of the model.

Implementation code

Function encBGV(int64_t x); for encryption using the BGV scheme:

```
Ciphertext<DCRTPoly> encBGV(int64_t x)
{
        vector<int64_t> r;
        r.push_back(x);
        Plaintext ptxt = cryptoContextBGV->MakePackedPlaintext(r);
        auto c = cryptoContextBGV->Encrypt(keyPairBGV.publicKey, ptxt);
        r.clear();
        return c;

}
```

Here is the code segment that maps the target labels to their corresponding values and stores them in the 'res' vector of type 'Ciphertext<DCRTPoly>>:

```
vector<Ciphertext<DCRTPoly>> res;
class OutputPrinter {
        private:
                ofstream fout;
        public:
                OutputPrinter(string filename) {
                        fout.open(filename,ios::app);
                        if(!fout) {
                                cout << filename << " file could not be opened\n";
                                exit(0);
                        }
                }

                string joinByTab(vector<string> row) {
                        for (const auto& labelValue : labelValueMap) {
                        if (labelValue.first==row[static_cast<int>(row.size()-1)])
                        {
                                res.push_back(labelValue.second);
                        }
                }
                        return row[static_cast<int>(row.size()-1)];
                }
                void addLine(string str) {
                        fout << str << endl;
                }
};
```

Here is the code segment for performing voting, which involves calculating the average of the output values from the decision trees, decrypting them, decoding, extracting the real part, and displaying it on the screen:

```
Plaintext result;
Ciphertext<DCRTPoly> avg = res[0];
std::vector<int64_t> finalResult;
if (static_cast<int>(res.size()) != 0) {
    for (int i = 1; i < static_cast<int>(res.size()); i++) {
        avg = cryptoContextBGV->EvalAdd(avg, res[i]);
    }
    cryptoContextBGV->Decrypt(keyPairBGV.secretKey, avg, &result);
    result->SetLength(8);
    finalResult = result->GetPackedValue();
    std::cout << "Risk of heart disease with Random Forest_BGV scheme: " << double(finalResult[0])/ static_cast<int>(res.size()) << std::endl;
} else {
    std::cout << "res.size()==0...." << std::endl;
}
```

Result

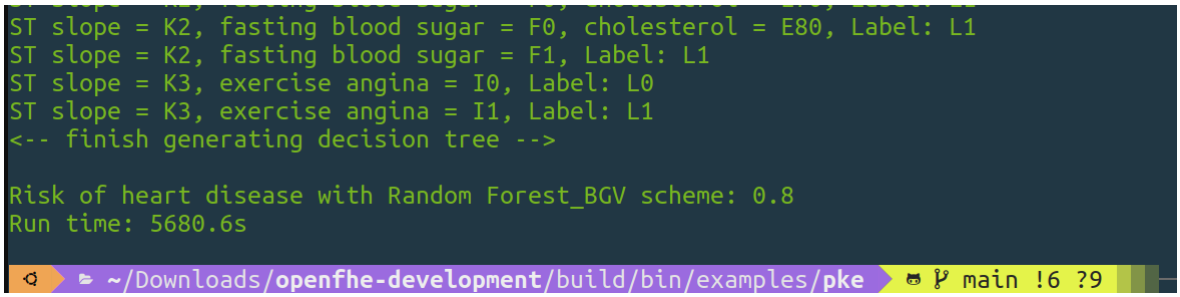
In the case where the target is 0, after the computation, we obtain a probability of returning 0.4

```
ST slope = K2, fasting blood sugar = F1, Label: L1
ST slope = K3, exercise angina = I0, Label: L0
ST slope = K3, exercise angina = I1, Label: L1
<-- finish generating decision tree -->

Risk of heart disease with Random Forest_BGV scheme: 0.4
Run time: 6102.39s

 ~/Downloads/openfhe-development/build/bin/examples/pke  main !6 ?9
```

In the case where the target is 1, after the computation, we obtain a probability of returning 0.8



3.5.2. Completion time

- The completion time for the case where target = 0 is: 6102.39s, equivalent to 1 hour and 41 minutes.
- The completion time for the case where target = 1 is: 5680.6s, equivalent to 1 hour and 34 minutes.

3.6. Comparison

	Target=0	Run time(s)	Target=1	Run time(s)
Random forest with CKKS scheme	0.4	6205.93	0.8	5781.97
Random forest with BGV scheme	0.4	6102.39	0.8	5680.6

The accuracy of the Random Forest model on the training set with 800 samples and the test set with 400 samples is:

3.7. Conclusion

Based on the given information, the CKKS scheme exhibits higher error and longer runtime compared to the BGV scheme. However, it's important to note that the dataset used for evaluation is relatively small compared to real-world datasets in hospitals, which may not clearly reflect the differences. In terms of error and runtime, the BGV scheme with integers is considered more favorable.

REFERENCES

1. Alotaibi, F. S. (2019). Implementation of machine learning model to predict heart failure disease. International Journal of Advanced Computer Science and Applications, 10(6)
2. Munjal, K., & Bhatia, R. (2022). A systematic review of homomorphic encryption and its contributions in healthcare industry. Complex & Intelligent Systems, 1-28