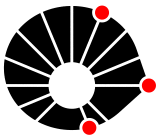

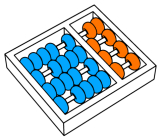

An Homomorphic Logistic Regression Inference Implementation

N. Takemoto *A. Guimarães* *E.Borin*

Relatório Técnico - IC-PFG-21-46
Projeto Final de Graduação
2021 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

A Study on Homomorphic Logistic Regression Implementation

Naomi Takemoto * Antonio Guimarães * Edson Borin *

Sumário

*State University of Campinas

Resumo

A criptografia homomórfica (*Fully Homomorphic Encryption* ou FHE) possibilita a realização de operações lógicas e aritméticas em dados encriptados que se refletem no dado decriptado. Por exemplo, sejam a e b dois números, e $\varepsilon()$ uma função de encriptação então: $\varepsilon(a) \bigcirc \varepsilon(b) = \varepsilon(a + b)$, onde $\bigcirc$ seria a operação análoga à adição para o conjunto de amostras encriptadas. Essa propriedade facilitaria o gerenciamento de dados para provedores de serviços em nuvem além de fornecer proteção para dados de clientes contra vazamentos.

Neste trabalho exploramos a implementação homomórfica da inferência para classificadores binários baseados em regressão logística. Empregamos o esquema TFHE [Chillotti et al., 2020] e utilizamos bibliotecas que já forneciam algumas operações lógicas e aritméticas, desenvolvendo outras operações conforme necessário. Utilizados dois conjuntos de dados, o primeiro referente a um censo americano [Kohavi and Becker, 1994] que nos serviu durante o processo de desenvolvimento e o segundo sobre o pós cirúrgico de câncer de mama [Haberman, 1976] com o qual reportamos nossos resultados. Implementamos dois modelos de inferência, um que operava com dados e pesos encriptados e outro com texto claro apenas. Este nos auxiliou durante o processo de escolha de parâmetros como precisão e número de bits reservados para parte fracionária do número de modo que o tempo de execução fosse menor, mas a acurácia fosse preservada. Com o modelo encriptado medimos o tempo de execução da inferência.

Identificamos que o fator mais importante para o desempenho do modelo era o número de atributos do dado de entrada, visto que ele determina o número de multiplicações homomórficas necessárias por amostra. Esta, a depender da precisão, se mostrou de 20 a 40 vezes mais demorada do que adição.

Abstract

Fully Homomorphic Encryption (FHE) makes it possible to perform logical and arithmetic operations on encrypted data that also happens in the underlying plaintext data. For example, let a and b be two numbers, and $\varepsilon()$ an encryption function, then: $\varepsilon(a) \bigcirc \varepsilon(b) = \varepsilon(a + b)$, where $\bigcirc$ would be the analogous addition operation for the set of encrypted samples. This property could facilitate data management for cloud service providers while protecting customers' data against leakage.

In this report, we explored the homomorphic encryption using the TFHE scheme [Chillotti et al., 2020] and applied it to the Logistic Regression inference using libraries that already provide a few logical and arithmetic operations, implementing others as needed. We used two sets of data, the first a 1994 US census [Kohavi and Becker, 1994] that we employed during the development process and the second a post-surgical breast cancer dataset [Haberman, 1976] for reporting our results.

We developed two models, the ciphertext model that worked with encrypted inputs and encrypted weights while the other, the cleartext, worked with cleartext only. This one we employed to find the best combination of precision for each ciphertext and the number of bits reserved for fraction so the execution time would be reduced but the inference accuracy preserved. With the ciphertext model, we focused on measuring the execution time.

We concluded that the number of features a sample from each dataset needs have a great impact on the overall performance since it sets the number of multiplications required in the inference. Finally, we observed that the multiplication implementation we used can be from 20 to 40 times costlier than one addition operation depending on the chosen precision.

1 Introduction

Cloud computing brings flexibility for clients allowing a faster and cost-effective scaling with a reduced operational burden. Security is usually a shared responsibility between the cloud user and the cloud provider. Choosing to delegate security configurations, like firewall, key management, etc, requires a certain level of trust on the cloud provider, which, depending on the nature of the application data, *e.g.*military, medial or financial, can face legal restrictions. These issues can restrict the cloud provider options a client can consider when moving their computation to the cloud.

Fully Homomorphic Encryption (FHE) allows arithmetic and logical operations to be performed in encrypted data instead of traditional encryption systems that require prior decryption. This property allows the client data to be processed on the cloud without exposing it to the provider. Also, suppose homomorphic encryption gets to the point that it can be implemented as a web protocol such as HTTPS. In that case, we could have an increased level of privacy on the web, making the user's data utterly opaque on the server-side. Some fields that deal with inherently confidential data, such as medical, financial, or military, would benefit. From the cloud providers point of view, an efficient FHE solution would simplify data handling. For instance, leakages on the server-side would have reduced (or no) impact. Dealing with governments and regulatory agencies data sovereignty policies would become more manageable.

In this work, we used the TFHE scheme [Chillotti et al., 2020] to implement the inference step of a binary classifier based on the Logistic Regression. We performed the input data processing, such as normalization and removal of invalid entries, and the training steps without cryptography. We employed the Gradient Descent algorithm, with the implementation described by [Brownlee, 2016] to minimize the Logistic Regression cost function and find the best set of weights. For the inference, we encrypted both the model and the weights.

We used the arithmetic and logical operations provided by [A. Guimarães, E. Borin, D. Aranha, *C-TFHE and UFHE libraries. Development version (Unpublished)*, 2021.] We also developed some additional homomorphic helper functions when necessary.

Because the execution time to compute the inference for the whole datasets is too high, aiming to facilitate the evaluation of our ciphertext model, we also developed a cleartext model using similar strategies to encode and operate on numbers, but without encryption, so testing different sets of parameters was less compute demanding.

With this work, we have found that having a clear text version of the UFHE library helped us to deal the difficulty of spotting and fixing errors enabling us to rapidly test different combinations of experimental parameters. We measured the correctness of our ciphertext model by comparing its inference accuracy with the baseline one developed in Python. We tried to minimize the absolute difference between the two mentioned models, also trying to reduce the precision to represent out data, so the memory required and time elapsed per inference would be lower. Related to the data representation, we explored the effects of two parameters, the *n_bits_precision*, the number of bits reserved to each number, and *n_bits_fraction*, the amount of bits reserved for the rational part of the numbers. For the [Haberman, 1976] dataset the best combination, minimal precision and difference from Python baseline, of the previous parameters was 16 and 8 bits respectively. We also

investigated the comparative cost of the homomorphic operations and we observed that the current implementation of the UFHE multiplication was from 20 to 40 times, depending on the precision, slower than the addition.

In this report, we will start presenting an overview of the main cryptographic concepts and terminology necessary for the reader, with some additional comments on the state of the art homomorphic schemes. Afterwards, we will introduce the TFHE data representation model and building block operations. We will also explain the Logistic Regression and how it can be used to perform binary classification and discussed some optimizations that can be made to reduce the number of computations. To describe our experiments methodology and resources used, we dedicated a section for methods and resources. In the Results section, we will present the performance we measured, based on time per inference, similarity between the homomorphic and the cleartext model's accuracy and amount of memory required during our experiments. Finally, with these results we analyzed we defined what could be done as future work.

2 Theory Overview

Throughout this text, we will need some terminology and concepts regarding the building blocks of cryptography. In this section, we will present a high-level explanation of them.

2.1 Terminology

Let's look at an analogy, suppose, for instance, that Alice wants to communicate with Bob over an insecure channel, *i.e.*, in this channel a message sent as is could be eavesdropped by a spy, let's say, Trudy. To avoid eavesdropping Alice can encrypt the message before sending it, and Bob must be able to read somehow it when he receives the encrypted message (ciphertext). Based on the Handbook of Applied Cryptography [Menezes et al., 2001], some of the key terms are:

- Plaintext or cleartext: it is an element of the set of allowed strings from an alphabet definition as is. It can be an English text, a computer program, etc. In our analogy, the original message Alice wants to send.
- Ciphertext: the output of the encryption algorithm. It is an element of the set of allowed strings from an alphabet definition that might, or not, be the same as the plaintext alphabet.

And, as a direct citation from Handbook of Applied Cryptography [Menezes et al., 2001]:

- K denotes a set called the key space. An element of K is called a *key*
- An encryption scheme consists of a set $\{E_e : e \in K\}$ of encryption transformations and a corresponding set $\{D_d : d \in K\}$ of decryption transformations with the property that for each $e \in K$ there is a unique key $d \in K$ such that $D_d = E_e^{-1}$; that is, $D_d(E_e(m)) = m$ for all $m \in M$. An encryption scheme is sometimes referred to as a cipher.

- The keys e and d in the preceding definition are referred to as a key pair and sometimes denoted by (e, d). Note that e and d could be the same.

In analogy, e would be Alice's key use for encryption, while d would be key used by Bob. If only Bob is supposed to be able of decrypting the ciphertext, then d is a secret key.

2.1.1 Symmetric Cryptography

In a symmetric cryptosystem, the two parts involved have to share a secret key used in both encryption and decryption. The key has to be exchanged between the parts prior to establishing secure communication. One industry standard example of a symmetric key encryption scheme is the 256-bit AES, proposed by [Daemen and Rijmen, 1999], which is recommended by the US government and leverages a 256-bit long secret key. The main challenge is to define how the parts will exchange it securely. Apart from that, AES can be used to handle large amounts of data efficiently thanks to hardware optimizations.

2.1.2 Asymmetric Cryptography

In public-key systems, there's a pair of keys, the *public key*, used for encryption, and the *private key*, for decryption. In a high-level view, suppose Alice wants to send a message that only Bob can read. Alice can use Bob's public key K_B^+ to encrypt the plaintext message m with it, obtaining the ciphertext $K_B^+(m)$. Then she can send that message to Bob, who is the only one having the private key K_B^- which allows the message to be decrypted, *i.e.*, Bob can perform $K_B^-(K_B^+(m)) = m$. An example of a widely used public-key system is RSA [Rivest et al., 1978], which relies its security on the integer factorization problem. Asymmetric cryptosystems are slower and usually used to exchange a small amount of data, such as keys of symmetric schemes.

2.2 FHE - Gentry

Before 2009, the existing homomorphic encryption schemes had limited usage because they did not support an unbounded number of additions and multiplications, which limited the number of circuits that could be represented. An example of the old generation is [Paillier, 1999] cryptosystem that allows an unbounded number of additions, but multiplication between two encrypted samples without using the private key is not possible. For LWE [Regev, 2010] based cryptosystems, the noise level grows with the number of homomorphic operations, and if it achieves a certain critical level, the correct decryption of the ciphertext is not possible, but eliminating the noise from samples was not an option because it could compromise security.

[Gentry, 2009] proposes the bootstrap operation, which allows the noise level of ciphertext to be "reset". To explain how this works, he proposes an analogy. In his story, Alice's Jewelry Store, the fictional character Alice runs a business in which she needs to assemble expensive materials into jewelery, but this task has to be performed by workers she distrusts. Therefore, she does not want to grant them direct access to the materials. The solution is to use a transparent box that has built-in gloves that allow a person to manipulate the

box's content, but since the box is locked with a key only Alice has access to, no one could retrieve its content.

To represent the issue of the growing noise in ciphertexts as more operations are performed, Gentry puts a restriction in the analogy: the boxes have limited usable time. So Alice has to find a way to pass the content of one box into another securely, *i.e.*, not getting the materials out of the protection of a box. The parallel in the cryptography problem is that we want to reduce the ciphertext noise without decrypting the ciphertext into plaintext in an untrusted host. What Alice can do is to get the content of a box A into box B while A is inside B, so the material that gets out of A is not exposed, *i.e.*, it is still protected by box B. In the cryptography problem, we would decrypt a message encrypted under a key K_A when $K_A(m)$ is under key K_B, so the message is never actually exposed, but the keys encrypting the message are switched, and the noise reset. This is known as the bootstrap operation.

Although the bootstrap operation can be used to reduce the noise, it is computationally costly. Therefore there are two approaches to deal with this trade-off:

- Leveled: the circuit (collection of logical and arithmetic operations necessary to model a problem) depth must be known in advance so the parameters, such as numeric precision, are set to guarantee that the decryption will be possible.
- Bootstrapped: bootstrap operations are performed, *e.g.*, at every operation - but not necessarily, to reduce the ciphertext noise allowing the implementation of arbitrarily deep circuits.

2.3 Homomorphic Schemes Today

Current homomorphic schemes are comprised of 5 stages:

1. Setup: in this stage steps are chosen the security parameters, the homomorphic scheme and functionality parameters.
2. Key Generation: all keys are generated, public and private, bootstrap, *etc.*
3. Encoding and Encryption: the process of converting the plaintext into ciphertext.
4. Evaluation: operations defined in the circuit are executed over the encrypted data.
5. Decryption: the private key is used, and the result of the computations made in the Evaluation step are retrieved.

Some popular homomorphic schemes are:

- BGV and BFV: BGV [Brakerski et al., 2011] was the first scheme based on the LWE problem [Regev, 2010]. In 2012 BFV [Lyubashevsky et al., 2010] brought improvements to BGV making computations based on the Ring-Learning with Errors (RLWE) problem also proposed by [Lyubashevsky et al., 2010], which is an extension of LWE. These schemes focus on exact computation, *e.g.*, without approximations;

- TFHE [Chillotti et al., 2020]: uses logical gates with data represented as bits, operations are implemented in logical circuits, also based on the LWE and RLWE problems; and
- CKKS [Cheon et al., 2017]: works with approximate computation on vectors of numbers aiming to gain efficiency. Because of this, it is capable of achieving practical performance.

2.3.1 Finite Fields

Finite Fields or Galois Fields is a key concept to understand the TFHE cryptosystem, which is Tori based. This section provides an overview intended to define some common terms that will be employed throughout this document. To do so, we will show the definitions of group and field as presented by [Paar and Pelzl, 2009] in the *Undestanding Cryptography* book that accompanies the online course taught by C. Paar.

Definition 2.1 (Group)**.** A group is a set of elements G together with an operation $\bigcirc$ that combines two elements of G. A group has the following properties:

1. The group operation $\bigcirc$ is closed. That is, for all $a, b, \in G$, it holds that $a \bigcirc b = c \in G$.
2. The group operation is associative. That is, $a \bigcirc (b \bigcirc c) = (a \bigcirc b) \bigcirc c$ for all $a, b, c \in G$.
3. There is an element $1 \in G$, called the neutral element (or identity element), such that $a \bigcirc 1 = 1 \bigcirc a = a$ for all $a \in G$.
4. For each $a \in G$ there exists an element $a^{-1} \in G$, called the inverse of a, such that $a \bigcirc a^{-1} = a^{-1} \bigcirc a = 1$.
5. A group G is abelian (or commutative) if, furthermore, $a \bigcirc b = b \bigcirc a$ for all $a, b \in G$.

Definition 2.2 (Field definition)**.** A field $F$is a set of elements with the following properties:

1. All elements of F form an additive group with the group operation "+" and the neutral element 0.
2. All elements of F except 0 form a multiplicative group with the group operation "$\times$" and the neutral element 1.
3. When the two group operations are mixed, the distributivity law holds, *i.e.*, for all $a, b, c \in F : a(b + c) = (ab) + (ac)$.

A field can be either infinite or finite, a finite field always has p^m elements, where p is a prime number and m is a positive integer. For instance a finite field with 11 elements is represented as $GF(11) = GF(11^1)$, in this case $p = 11$ and $m = 1$. There are two types of finite fields: the prime fields and the extended fields.

Prime fields are the case when $m = 1$ and its representations can be simplified to $GF(p)$ omitting the m. The elements in the prime field are the integers in the set $\{0, 1, ..., p - 1\}$. Let $a, b, c \in GF(p)$, the prime field operations are defined as:

1. Addition: $a + b \equiv c \mod p$;

2. Subtraction: $a - b \equiv d \mod p$; and

3. Multiplication: $a * b \equiv e \mod p$.

Due to the $\mod p$ reduction, the above operations are closed because their results belong to $\{0, 1, ..., p-1\}$. From the quotient remainder theorem, linked in the references, it's possible to show that the operations are associative and commutative. For the addition, the neutral element is 0, and the inverse operation is the subtraction, and vice versa - thus an additive group. For the multiplication, the neutral element is 1, and the inverse is defined to all values of the set, except 0, and can be computed with the extended euclidean algorithm , also mentioned in our references. Thus satisfying the second requirement of the field definition.

Extended fields happen for cases when $m > 1$, and its elements are not numbers but polynomials. For instance:

$$a_{m-1}x^{m-1} + ... + a_1x + a_0 = A(x), A(x) \in GF(2^m)$$

with $a_i \in GF(2) = \{0, 1\}$. Let $A(x), B(x) \in GF(2^m)$. The addition of two elements is defined:

$$C(x) = A(x) + B(x) = \sum_{i=0}^{m-1} c_i x^i, c \equiv a_i + b_i \mod 2 \tag{1}$$

And the subtraction:

$$C(x) = A(x) - B(x) = \sum_{i=0}^{m-1} c_i x^i, c \equiv a_i - b_i \mod 2 \tag{2}$$

Similarly to the Prime Fields case, we can show that the operation as defined above on the elements on the fields is an additive group. To define the multiplication, we will need a polynomial $P(x)$:

$$P(x)m \equiv \sum_{i=0}^{m} p_i x^i, p_i \in GF(2) \tag{3}$$

be irreducible, *i.e.*, cannot be factored into a product of two non-constant polynomials. Then the multiplication is:

$$C(x) \equiv A(x) * B(x) \mod P(x) \tag{4}$$

Here the inverse of the multiplicative operation can also be computed with the support of the extended euclidean algorithm.

2.4 Torus

A Torus is an abelian (commutative) group, but the multiplication between two members of the group is not defined. The TFHE relies on the real and the polynomial Torus. According to Chillotti and Leuven [Chillotti et al., 2020]: The **Real Torus** ($\mathbb{T}$) $(\mathbb{T}, +, \cdot)$ is: $\mathbb{T} = \mathbb{R}/\mathbb{Z} = \mathbb{R} \mod 1$. The addition is well defined for every two elements in the Torus, as well as the external product, *i.e.*, it is possible to multiply an element of the Torus by a constant $\mathbb{Z}$. Also, the **torus polynomials** is $\Re$-module, the set of polynomials with integer coefficients modulo 1, denoted $\mathbb{T}_{\mathbb{N}}[\mathbb{X}] = \mathbb{T}[X] \mod (X^N + 1)$. $(\mathbb{T}_N[X], +, \cdot)$ has the addition and the multiplication by a constant well defined, but the multiplication is not.

3 TFHE

TFHE is a lattice-based cryptosystem that relies extensively on the Learning With Errors (LWE) problem presented by [Regev, 2010], and its variant, the Ring Learning with Errors (RLWE in short), proposed by [Lyubashevsky et al., 2010]. In the next subsections, we will present the ciphertext types for LWE and RLWE problems mentioned earlier.

3.1 Ciphertext types

3.1.1 LWE

An LWE sample can encode a message m from the real torus. The ciphertext $c = (\vec{a}, b) \in \mathbb{T}^{n+1}$, where $\mathbb{T}^{n+1}$ can be represented by a vector of $n + 1$ elements and $\vec{a}$ is public. The secret key $\vec{s} = (s_0, s_1, ..., s_{n-1})$ is a vector of n elements, $s_i \in \{0, 1\}$. $\vec{a}$ and $\vec{s}$ are sampled from a uniform distribution, and:

$$b = \langle a, s \rangle + e + m \tag{5}$$

e is a scalar sampled from a Gaussian distribution with a 0 mean. The decryption consists in computing: $m' = \lfloor b - \langle a, s \rangle \rfloor$, if the decryption is correct then $m' \approx m$. The LWE sample supports addition and constant multiplication.

3.1.2 RLWE

A message in RLWE encodes a message $M(x)$ in the polynomial torus $\in \mathbb{T}_{\mathbb{N}}[\mathbb{X}]$. $M(x)$ and the secret key $S(x)$ have N binary coefficients. A ciphertext $C(x) = (A(x), B(x)) \in \mathbb{T}_N[X]^2$, with $A(x)$ being a polynomial with randomly sampled coefficients and $B(x) = A(X) \cdot S(X) + E(x) + M(x)$, $E(x) \in \mathbb{T}_{\mathbb{N}}[\mathbb{X}]$ sampled from a Gaussian distribution. Similarly to the LWE sample, the decryption is computed as: $\lfloor B(x) - A(x) \cdot S(x) \rfloor$. Addition and multiplication by constant are well defined.

3.2 Building blocks

In order to implement the homomorphic inference, we had to rely on the following operations, more details can be found in the [Chillotti et al., 2020] paper:

1. Key Switching: can be used to switch the encryption key, but also to change from one sample to another, for example, LWE to RLWE.

2. Sample Extraction: get an LWE sample from a packed RLWE.

3. Blind Rotation: is the core operation used to evaluate lookup tables. It works by rotating the polynomial $M(x)$, which encrypts an RLWE sample by an amount p (encrypted).

4. Functional Bootstrap [Boura et al., 2019]: the TFHE bootstrap uses the blind rotate operation to evaluate a lookup table that contains a discretized function. After the rotation, the Sample Extraction operation can be used to retrieve the result.

4 Logistic Regression

In this section we will present the Logistic Regression training and inference steps, the mathematical notation we used was based on [Géron, 2017]. Logistic regression is a widely used method to perform predictions and classifications in several fields, and its core is the sigmoid function $\sigma(t)$ which is defined as:

$$\sigma(t) = \frac{1}{1 + e^{-t}} \tag{6}$$

This function's output is bounded between 0 and 1 that can be interpreted as the probability of an instance to belong to a class (1) or not (0), which allows it to be used to perform binary classification.

Let $\hat{y}$ be the model's prediction, x the an input sample with a certain number of features, and θ the set of weights, then $\hat{p} = h_\theta(x)$, the probability of x to belong the positive class, is:

$$\hat{p} = h_\theta(x) = \sigma(\theta^T x) \tag{7}$$

Once we have the weights θ, the prediction criteria for the binary inference is:

$$\hat{y} = \begin{cases} 0 & \hat{p} < 0.5 \\ 1 & \hat{p} \geq 0.5 \end{cases}$$

Or simply round($\hat{p}$). Note that if $t < 0$ then $\sigma(t) > 0.5$ and when $t \geq 0$ then $\sigma(t) \geq 0.5$. This allow us to simplify the inference by simply checking whether the input is greater or equal 0, thus for the binary classification use case we can eliminate the sigmoid evaluation.

The set of weights θ is defined by a training process. This process is performed by minimizing the following cost function:

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} \log(p^{(i)}) + (1 - y^{(i)}) log(1 - \hat{p}^{(i)})] \tag{8}$$

This cost function tends to grow when the prediction fails. Given that the label y can be either 0 or 1, then the cost value is expressed as:

$$c(\theta) = \begin{cases} -\log(\hat{p}) & y = 1 \\ -\log(1 - \hat{p}) & y = 0 \end{cases}$$

When the label $y = 1$ and the prediction $\hat{p}$ is 0, then the cost is maximum. The same when the label $y = 0$ and the prediction $\hat{p}$ is one.

The Gradient Descent algorithm supports the minimization of the function, repeating the following steps iteratively:

1. Compute the first-order derivative of the cost function.
2. Adjust the current weights by making a step in the opposite step of the derivative, scaled by the learning rate α.

In short, repeat the following step until convergence (or a fixed number of iterations):

$$J(\theta) = J(\theta) - \alpha \frac{\partial J(\theta)}{\partial \theta} \tag{9}$$

5 Methods and Resources

In this section we will describe: the dataset, libraries, and tools that helped in our experiments. We will also present our inference models and strategies to evaluate them along with some preliminary results. We will discuss some implementation issues we faced and how they impacted on our decision to change datasets.

5.1 Census Dataset

We first used the logistic regression to implement a binary classifier for a Census problem; its goal was to predict whether a person's annual income was above or below $50,000 based on census data from 1994, available in UCI Machine Learning Repository [Kohavi and Becker, 1994]. The training data contains 32561 samples while the test had 16281, each sample containing 14 different attributes of both types, numerical and categorical.

For the training step, we reproduced part of the steps described by Agarwal [Agarwal, 2018] to prepare the dataset. We first removed 2399 samples that had missing features of the training dataset, which represents 7.37% of its total size. Then, we used the data processing pipeline provided by A. Agarwal to transform categorical data to numeric and to normalize it. After this procedure, the number of features increased from 14 to 81 due to the One Hot Encoding [scikit learn, 2021] technique that creates a new numerical feature for each different class of a categorical one. Then we saved the logistic regression's weight that would be used to implement the inference step homomorphically.

The training was performed using Gradient Descent as the optimization algorithm. The learning rate was set to 0.001, and the number of epochs was 1000. The accuracy on

the train set was 84.40%, and in the test set was 84.71%, similar to A. Agarwal 85.00% and 85.2% respectively. Agarwal's results differ from ours because we decided to drop the samples with missing values while Agarwal filled and kept them for the training set. These baseline accuracy values were obtained from an inference model implemented in Python with the support of the Numpy library, a well-known package for scientific computing. This baseline inference is shown:

```
1 def predict(w, x):
2     p_hat = sigmoid(np.dot(x, w))
3     binary_preditions = np.where(y_hat < 0.5, 0.0, 1.0)
4     return binary_preditions
```

It is worth noticing that the np.dot() function behaviour when called for two multi-dimentional arrays, as in our case, results in a array containing the dot product between w and each of the samples in x as explained in the [Numpy and Documenation, 2021].

This is the core inference we wanted to implement homomorphically, and its comprised of 3 steps:

1. Compute the dot product between the weights w and the input x.
2. Compute the sigmoid for the dot product obtained - this and the next step could be skipped by checking whether the dot product is greater or equal than 0 as explained earlier.
3. Since we are dealing with a binary classification problem, compute the result by rounding the above's output to 0 or 1.

5.2 CTFHE and UFHE libraries

We used the CTFHE and UFHE [Guimarães et al., 2021] C libraries, which provide homomorphic implementations of basic arithmetic and logical operations and integrated with our C++ code. UFHE library operates upon numbers encoded as ufhe_integer, this type supports both signed, and unsigned integers, represented in base 4 (each digit can encode up to 4 different values), and its precision can be parameterized. Each digit is an LWE sample represented by the scalar b, and the vector a size 2048 (same size as the secret key), b and a's definitions are described in the LWE sample subsection.

To adjust our original plaintext data, 64 bits doubles to the ufhe_integer representation, we had to:

1. Find a way to use C/C++ integers to represent real numbers;
2. Convert C/C++ integers to ufhe_integer.

To address the first issue, we chose to convert the doubles from float point representation to fixed-point, which is equivalent to multiplying the double by a power of 2, then truncating it to an integer. For the second issue, we employed the functions provided by UFHE library to encrypt the numbers, using a private key also generated by the library.

5.3 Strategy for evaluating performance and accuracy

The execution time and memory required to handle operations grow with the chosen precision. Thus we decided to investigate what was the minimum precision necessary to represent our data while preserving the accuracy of the inference. To illustrate the amount of memory required in the Census problem:

1. Input samples x: each sample consists of 81(number of features) ufhe_integers.
2. Weights w: also a vector with 81 ufhe_integers.
3. Label y: 1 ufhe_integer.

Let's say that each ufhe_integer needs 32 bits precision to avoid information loss; it means that the number of digits $d = 32/2 = 16$, because 2 is the number of bits per digit. Each digit is represented by an LWE sample in $\mathbb{T}^{n+1}$ with $n = 2048$. To store only one sample of x and y and the weights, we would need 0.04 GB of memory without including the extra memory for auxiliary variables used in UFHE operations. If the whole train dataset (30162 samples) was in memory, it would be necessary at least 1194.0 GB. Even if we processed each sample at a time, loading the encrypted data from a file as necessary, which was our chosen approach, making the inference for the whole train (or test) dataset is infeasible to be done in a domestic computer because of high disk usage and execution time.

Also, the homomorphic inference is harder to debug, thus we implemented two models:

1. Cleartext: we used to measure the minimum precision necessary to store the data and perform operations and find the best shift amount for converting a C++ double to fixed point precision. This model is suitable for rapidly evaluating the accuracy and spotting bugs in the implementation.
2. Ciphertext: we used to measure the performance in terms of time per inference. Since the inference takes longer and the memory demands are high with this setup, it is not suitable for rapidly evaluating the accuracy of the whole dataset (train or test).

5.4 Cleartext Model

We implemented the UFHE alike functions in C++, but with cleartext samples, using arrays to represent numbers in base 4 - the algorithms implemented in this work are described by [Guimarães et al., 2021]. For instance, the number 6_{10} (6 in base 10) in our cleartext model would be multiplied by a power of 2, in our case 2^{16}, resulting in 393216, then converted to base 4, resulting in 1200000000_4 and each digit assigned to a position on an integer vector, the least significant digit in the least significant position and the most significant ones were filled with 0's. Negative numbers were handled with Two's complement. This model can be tunned by the following parameters:

1. n_bits_precision: number of bits each number would use, for instance if it was set to 16, then the number of digits d would be $d = n\_bits\_precision/log_2(base)$, in this case as $base = 4$, the number of digits would be 8.

2. n_bits_fraction: number of bits reserved to represent the rational part of the number. Similarly to the above, if, for instance, it was set to 8, then the number of digits would be 4.

We generated the sigmoid table with the discretization of the sigmoid function with the following parameters, so we were able to investigate if there was a difference in the required precision if we used a sigmoid table:

1. x_max and x_min: for $x \in [x_{max}, x_{min})$, the value of $\sigma(x)$ can be approximated by a value present in the discretization of the sigmoid function.

2. sigmoid_size: the number of samples in the sigmoid table.

5.4.1 Find the minimum precision for inference

The predict function listed below computes the equivalent of $\sigma(\theta^T x)$, *i.e.*, it performs the dot product between the weights and input samples vector, the output is a vector, where each sample is a number (between 0 and 1) represented in fixed-point precision and base 4, stored in an array (Number is an alias to the type std::vector <int>).

```
1  std::vector<Number> predict(std::vector<Number> &w, std::vector<
       std::vector<Number> > &x) {
2    std::vector<Number> result;
3    int n_samples = x.size();
4    int n_features = x[0].size();
5    int n_digits = config::DEFAULT_CONFIG.n_bits_precision / 2; //
       base 4 needs 2 bits
6    Number temp = Number(n_digits, 0);
7
8    Number p_hat = NUMBER_ZERO;
9    for (int row = 0; row < n_samples; row++) {
10     acc = NUMBER_ZERO;
11     for (int col = 0; col < n_features; col++) {
12       Number temp = lookup::multiply(w[col], x[row][col]);
13       acc = lookup::add(temp, acc);
14     }
15     p_hat = sigmoid(acc);
16     result.push_back(line_result);
17   }
18
19   return result;
20 }
```

To help us estimate the minimum precision required, we measured the min and max values obtained in the predict function: the min and max in the training samples were -1.3940 and 1.5562, and for after the dot product, the min and max were: -7.870 and 23.712

respectively. From this, we found that to represent the numbers without loss, the integer part of the fixed-point number requires at least four digits, because $24_{10} = 0120_4$. This means we need $4 * number\_of\_bits\_per\_digit = 8$ bits, so we focused on trying to find the number of bits necessary to represent the rational part of the number. In the results of Table 1, the sigmoid table size was set to 16, so we would not need higher precision to store the table's index during our computations.

Table 1: Cleartext model's accuracy when used a discrete sigmoid table vs. the optimized version which did not use sigmoid table with 16 entries. The column distance from baseline contains the absolute difference between the accuracy of the model on the left and the baseline on train dataset (84.4009 %).

number precision [bits]	**fraction precision [bits]**	**sigmoid model - accuracy**	**distance from baseline**	**model without sigmoid - accuracy**	**distance from baseline**
32	16	0.843976	3.30E-05	0.843976	3.30E-05
16	8	0.844175	1.66E-04	0.844042	3.30E-05
16	6	0.755056	8.90E-02	0.842418	1.59E-03
16	4	0.751078	9.29E-02	0.580764	2.63E-01
12	8	0.634142	2.10E-01	0.612194	2.32E-01
12	6	0.839202	4.81E-03	0.838704	5.31E-03
12	4	0.754128	8.99E-02	0.835488	8.52E-03
12	2	0.751078	9.29E-02	0.248922	5.95E-01
12	0	0.248922	5.95E-01	0.248922	5.95E-01

The first two rows of Table 1, (number precision, bits for fraction) pairs: (32, 16) and (16, 8) respectively, have shown the best distance between the baseline accuracy for both models, with and without sigmoid. In particular, the model without sigmoid preserved accuracy when reducing from 32 to 16 bits precision, which did not happen for the model with the sigmoid. This is probably related to the extra computations necessary to evaluate the sigmoid table. When considering the 16 bits precision, using 8 bits for fraction had the minimum error. These results led us to choose the model with 16 bits, with 8 bits for the fraction part, without sigmoid, as this setup preserved the baseline accuracy while having the lowest memory requirement.

5.4.2 Ciphertext Model

To help us estimate the minimum precision required, we developed a ciphertext model with encrypted weights and input. We encoded these data into fixed point precision and base four then converted them into ufhe_integers. This model was divided into two, the client-side and the server-side. The client-side, encrypt_data.cpp, is responsible for generating the required keys, encrypting the input data, x and y, and the model weights w using the private key, and save to a file that becomes the input for the inference program (inference.cpp). The server-side inference was implemented using the UFHE version of the arithmetic and logical

operations used in the cleartext model, and it returns the number of matches, encrypted, between the model's prediction and the labels y. Here the private key was not used for computations.

To evaluate the inference, we randomly chose ten samples from the training dataset and used the encrypt_data.cpp program to generate and encrypt the necessary information, then used this information in the inference program. The inference demanded 4h25 minutes to run with 21.7 GB memory allocated. In another experiment with a single sample, the time elapsed was 1634982 ms or approximately 27 minutes. At this point, we had two major issues:

1. The time necessary to evaluate the inference made it hard to identify issues with implementations/parameter choice;
2. The amount of memory demanded.

For 1, we observed that the multiplication was the most time-consuming operation and, as in the clear text model, 5.4.1 line 12, it is executed under two for loops to compute the dot product between the weights and the features of each x sample. Since each of our x samples have 81 features, the number of multiplications per sample was 81. Table 2 shows the average execution time for the multiplication and the addition operations and the comparison between the two. For the precision values tested, we found that the multiplication can be from 20 to 40 times slower than the addition operation. This can be explained by its implementation, instead of a "native" multiplication using RGSW [Gentry et al., 2013] sample, it was done with lookup tables that encoded the possible values for every two-digit product in base 4. Then used the Long multiplication Algorithm [Multiplication and Algorithm,], with complexity $O(n^2)$, n the number of digits, to compute the product for the entire ufhe_integer.

Table 2: Comparing multiplication vs addition time.

Precision	**Multiplication [ms]**	**Addition [ms]**	**Times Slower [ms]**
10	9743.7	470.0	20.8
16	19580.3	822.0	23.8
32	69794	1535 .0	45.5

We tried to reduce the number of multiplications by retraining our logistic regression model, now dropping some features. But we could not do so without decreasing the accuracy. The high number of features was a consequence of the categorical features. For each possible categorical value, the One Hot Encoding technique creates a new feature. For example: for the "occupation" feature, there are 14 possible classes: Adm-clerical, Armed-Forces, Craft-repair, *etc.*. For "occupation" features alone, with One Hot Encoding, it would be converted to 14 new features, with values 0 (not belonging) or 1 (belonging), thus increasing the number of multiplications necessary in the inference step. One of our attempts to reselect the features was choosing only the numerical ones and retraining from the logistic

regression model. Though the accuracy for inference in the train set (measured in cleartext in the Python script) was 79.03%, the accuracy for the validation set was: 9.022% indicating the model wasn't able to generalize. The difficulty in reducing the dimension of the features aligned with the issues with memory and time for inference led us to change the dataset so we could reliably evaluate the ciphertext model, more details on the new dataset in the results section.

5.4.3 Implementation issues

Using homomorphic encryption increases memory usage because of the LWE/RLWE samples needs. During our first implementation approach, we did not reuse the ufhe_integer variables, choosing to allocate (malloc) and deallocate (free) dynamically. This approach was problematic because of the common behaviour of operating systems memory management strategy. When the memory gets freed, it does not return to OS, and in our case, while more memory got allocated, the program's memory usage kept growing despite the variables were being freed. One strategy to deal with this issue would be using the fork() system call to create another process that, at the end of its execution the exit() system call. With the process termination, the memory would return to the operating system.

5.4.4 Binary Inference Optimization Details

As mentioned before, because we are using logistic regression for binary classification, we could replace the sigmoid evaluation with a check for whether the number is greater than zero. For the ciphertext model, we opted for this strategy since its development was simpler, and the execution time was also reduced.

We had to create some functions, one of them determines whether a number is greater than zero. We presented the high-level view of the function below.
ufhe_greater_than_zero(ufhe_integer out, ufhe_integer in):

1. Extract the sign digit, the most significative digit, of the input number (*in*);
2. Initialize the output (*out*) to 0 - using an ufhe function that allows encoding a cleartext value into an ufhe_integer;
3. Program the lookup table (lut), encoded as a RLWE sample with entries that contains the logic: *if sign is 0 than return 1, else return 0.* A plaintext analogy would be the vector lut = [1, 0, 0, 0], when the selector is 0, here interpreted as index, then return 1 else return 0; and
4. Use the functional bootstrap to perform the lookup operation, setting *out* variable to the correct value.

We needed a function to increase a counter (n_matches) when the prediction was equal to the label. To do so we used the *binary_inference(ufhe_integer n_matches, ufhe_integer actual, ufhe_integer label*) function that encodes the logic *if actual == label then n_matches += 1.*

1. Use the ufhe_cmp_integer to compare the prediction with the expected label. The function returns 0 if the first argument is less than the second, 1 is they are equal and 2 if the second is greater than the first;

2. Create a lookup table [0,1,0,0]. Use the ufhe_lut_integer function from the UFHE library to perform a lookup table evaluation (the table is provided in cleartext) while the selector, an encrypted ufhe_integer, is the result of ufhe_cmp_integer. When the selector is 1 (index 1 of the table), the lookup returns 1, else 0; and

3. Add the result to an accumulator (n_matches).

5.5 Platform information

We did the experiments on a machine with the following specs: 2.6 GHz 6-Core Intel Core i7 with 16 GB 2667 MHz DDR4. Apple clang version 12.0.0 (clang-1200.0.31.1). When measuring execution time, the computer was set to not sleep, and all unnecessary programs, such as the browser and other apps, were closed. Memory usage was measured with the htop utility.

6 Results

As explained in the previous section, the Census dataset was not a good fit for our experiments due to a large number of features. Our new dataset was Haberman's Survival[Haberman, 1976], which contains data of patients who have undergone breast cancer surgery. It is composed of 4 features: age of the patient, operation year, number of nodes detected and survival (whether the patient survived the surgery 5 years or longer after the procedure). The total number of samples was 306, and we reserved 20% (62 samples) for the test set. We trained the logistic regression model as before. It is worth noticing that we normalized train and test data by dividing them by the mean of each feature computed from the training dataset. Then we minimized the cost function with Gradient Descent. The baseline accuracy on the train set was 76.23% and 70.97% in the test set.

We tried to find the best combination of n_bits_precision and n_bits_fraction using the results presented in Table 3. The goal was to reduce the difference between the baseline accuracy computed with our Python script instead of increasing the overall accuracy. The rationale is: by just aiming for the model's accuracy, we could favour the bias in the dataset - 73.53% approximate label 1 and the other 26.47% label 0. Table 3 indicates that the best setup for precision and the number of bits reserved to the rational part of the number is the pair (16, 8), same as the Census dataset.

We made the inference for train and test datasets as shown in tables 4 and 5. At first, we observed that the best precision we chose based on the cleartext model did not result in the expected accuracy. We then increased the precision by 2 bits and re-ran the experiments. This improved the accuracy of the ciphertext model inference but increased the inference time per sample by 13 seconds. From the implementation details of UFHE library, we know that addition and multiplication operations can increase the sample's noise, though

Table 3: Finding the minimum precision. Column 1 indicates how many bits an ufhe_integer can represent (of a plaintext data). Column 2, the shift amount used to convert a double to a fixed point representation, *e.g.*, 8 bits mean the double number was multiplied by 2^8, reserving 8 bits for the rational part. Column 3, the cleartext model's accuracy (ranging from 0 to 1). Column 4, is the distance between the baseline accuracy measured by our Python script and the accuracy measured in our plaintext model (76.23 %).

Precision	**Number of bits for fraction**	**Ciphertext model accuracy**	**Absolute distance from baseline**
32	16	0.762295	8.20E-08
16	8	0.762295	8.20E-08
16	6	0.762295	8.20E-08
16	4	0.770492	8.20E-03
16	2	0.262295	5.00E-01
14	8	0.758197	4.10E-03
14	6	0.758197	4.10E-03
14	4	0.77459	1.23E-02
14	2	0.262295	5.00E-01
12	8	0.704918	5.74E-02
12	6	0.758197	4.10E-03
12	4	0.77459	1.23E-02
12	2	0.262295	5.00E-01
12	0	0.262295	5.00E-01

the noise growth per digit would not decrease with the precision increase. We still have not identified the root cause of this behaviour, but we believe it might be related to the multiplication overflowing that could be mitigated by an increase of precision for the most significant ufhe_integer digits.

Table 4: Time and accuracy for ciphertext model in training dataset

Precision	**Bits for fraction**	**Total Time Elapsed**	**Time per Sample**	**Ciphertext Accuracy**	**Cleartext Accuracy**
16	8	3h52 min	57 s	0.5615	0.7623
18	8	4h48 min	70 s	0.7623	0.7623

Increasing the precision from 16 to 18, we obtained the expected accuracy results, as shown in table 4 and 5.

Table 5: Time and accuracy for ciphertext model in test dataset

Precision	**Bits for fraction**	**Total Time Elapsed**	**Time per Sample**	**Ciphertext Accuracy**	**Cleartext Accuracy**
16	8	59min	57 s	0.5	0.7097
18	8	1h12min 48 min	70 s	0.7258064516	0.7097

7 Conclusion

For our currently homomorphic implementation of the logistic regression inference, the key to minimizing the execution time is to reduce the number of features in the input data, so the number of multiplications is also reduced. Also, when converting C/C++ doubles to fixed point implementation, the amount of precision dedicated to represent the rational part of the number can impact the model's accuracy. One key point to support us in the development process had a cleartext model that helped us to debug and experiment with different parameters faster than we could with only the ciphertext model. The cleartext model, similarly to the UFHE library, could be used in other applications with different computation circuits supporting the parameters tuning of the ciphertext model. We also found out that different operations, such as multiplication and addition can have large differences in the execution times due to the current UFHE implementation that has complexity $O(n^2)$, for n the number of digits of the number.

8 Future Work

A useful tool to automate parameter setting would be a mechanism that automatically chooses the best precision for a given circuit and input data ranges. Also, finding methods to improve feature reduction for training so that we need fewer multiplications would improve the inference time per sample. Using sub-processes is an alternative to enhance memory management for the program and fix the issues we discussed earlier.

In our work, we used the model's weights as ciphertext, and this forced us to employ a multiplication algorithm that emulated an LWE $\times$ LWE operation. If the weights are cleartext, we could replace that multiplication with a scaling function that would multiply a cleartext value (weight) by an encrypted one (x sample), thus reducing the cost of the dot product. Another alternative to reduce the multiplication cost is to explore BVF-like multiplications that improves noise management and performance.

References

[Agarwal, 2018] Agarwal, A. (2018). Logistic regression classifier on census income data.

[Boura et al., 2019] Boura, C., Gama, N., Georgieva, M., and Jetchev, D. (2019). Simulating Homomorphic Evaluation of Deep Learning Predictions. In Dolev, S., Hendler, D.,

Lodha, S., and Yung, M., editors, *Cyber Security Cryptography and Machine Learning*, pages 212–230, Cham. Springer International Publishing.

[Brakerski et al., 2011] Brakerski, Z., Gentry, C., and Vaikuntanathan, V. (2011). (leveled) fully homomorphic encryption without bootstrapping.

[Brilliant.org, 2021] Brilliant.org (2021). Extended euclidean algorithm.

[Brownlee, 2016] Brownlee, J. (2016). Gradient descent for machine learning.

[Cheon et al., 2017] Cheon, J. H., Kim, A., Kim, M., and Song, Y. S. (2017). Homomorphic encryption for arithmetic of approximate numbers. In Takagi, T. and Peyrin, T., editors, *ASIACRYPT (1)*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer.

[Chillotti et al., 2020] Chillotti, I., Gama, N., Georgieva, M., and Izabachène, M. (2020). TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91.

[Daemen and Rijmen, 1999] Daemen, J. and Rijmen, V. (1999). Aes proposal: Rijndael.

[Gentry, 2009] Gentry, C. (2009). *A fully homomorphic encryption scheme*. PhD thesis, Stanford University. `crypto.stanford.edu/craig`.

[Gentry et al., 2013] Gentry, C., Sahai, A., and Waters, B. (2013). Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Canetti, R. and Garay, J. A., editors, *Advances in Cryptology – CRYPTO 2013*, pages 75–92, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Guimarães et al., 2021] Guimarães, A., Borin, E., and Aranha, D. (2021). C-TFHE and UFHE libraries. Development version (Unpublished), 2021.).

[Géron, 2017] Géron, A. (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Sebastopol, CA.

[Haberman, 1976] Haberman, S. J. (1976). Haberman's survival data set.

[khanacademy, 2021] khanacademy (2021). The quotient remainder theorem.

[Kohavi and Becker, 1994] Kohavi, R. and Becker, B. (1994). Census income data set.

[Lyubashevsky et al., 2010] Lyubashevsky, V., Peikert, C., and Regev, O. (2010). On ideal lattices and learning with errors over rings. In Gilbert, H., editor, *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Menezes et al., 2001] Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. (2001). *Handbook of Applied Cryptography*. CRC Press.

[Multiplication and Algorithm,] Multiplication and Algorithm (-). Multiplication algorithm.

[Numpy and Documenation, 2021] Numpy and Documenation (2021).

[Paar and Pelzl, 2009] Paar, C. and Pelzl, J. (2009). *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st edition.

[Paillier, 1999] Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In Stern, J., editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Regev, 2010] Regev, O. (2010). The learning with errors problem (invited survey). In *2010 IEEE 25th Annual Conference on Computational Complexity*, pages 191–204.

[Rivest et al., 1978] Rivest, R. L., Shamir, A., and Adleman, L. M. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126.

[scikit learn, 2021] scikit learn (2021). One hot encoding.