Evaluation of Privacy-aware Support Vector Machine (SVM) Learning using Homomorphic Encryption

William J Buchanan[1] and Hisham Ali[1]

Blockpass ID Lab, Edinburgh Napier University
b.buchanan@napier.ac.uk

Abstract. The requirement for privacy-aware machine learning increases as we continue to use PII (Personally Identifiable Information) within machine training. To overcome these privacy issues, we can apply Fully Homomorphic Encryption (FHE) to encrypt data before it is fed into a machine learning model. This involves creating a homomorphic encryption key pair, and where the associated public key will be used to encrypt the input data, and the private key will decrypt the output. But, there is often a performance hit when we use homomorphic encryption, and so this paper evaluates the performance overhead of using the SVM machine learning technique with the OpenFHE homomorphic encryption library. This uses Python and the scikit-learn library for its implementation. The experiments include a range of variables such as multiplication depth, scale size, first modulus size, security level, batch size, and ring dimension, along with two different SVM models, SVM-Poly and SVM-Linear. Overall, the results show that the two main parameters which affect performance are the ring dimension and the modulus size, and that SVM-Poly and SVM-Linear show similar performance levels.

Keywords: homomorphic encryption, Support Vector Machine, privacy-aware

1 Introduction

The rise in machine learning (ML) has caused an increasing demand for data to be used in creating data for learning. Unfortunately, this data can also include PII, and where it is often needed to be protected before it is shared. While data can be protected *over-the-air* and *at-rest*, we often do not protect data *in-process*. For this, we can use homomorphic encryption to process encrypted data. This can either be Partial Homomorphic Encryption (PHE) or Fully Homomorphic Encryption (FHE). With FHE, we can implement all of the arithmetic operations, while PHE only implements a reduced number of operations. With this, FHE typically uses lattice cryptography, and which often has an increased processing requirement for its implementation. This paper thus makes a core contribution in applying FHE to the SVM (Support Vector Machine) models, and then evaluates the performance of this using a range of parameters using within the OpenFHE library [1].

2 Background

Homomorphic encryption supports mathematical operations on encrypted data. In 1978, Rivest, Adleman, and Dertouzos [2] were the first to define the possibilities of implementing a homomorphic operation and used the RSA method. This supported multiply and divide operations [3], but does not support addition and subtraction. Overall, PHE supports a few arithmetic operations, while FHE supports add, subtract, multiply, and divide.

Since Gentry defined the first FHE method [4] in 2009, there have been four main generations of homomorphic encryption:

– 1st generation: Gentrys method uses integers and lattices [5] including the DGHV method.
– 2nd generation. Brakerski, Gentry and Vaikuntanathans (BGV) and Brakerski/ Fan-Vercauteren (BFV) use a Ring Learning With Errors approach [6]. The methods are similar to each other, and there is only a small difference between them.
– 3rd generation: These include DM (also known as FHEW) and CGGI (also known as TFHE) and support the integration of Boolean circuits for small integers.
– 4th generation: CKKS (Cheon, Kim, Kim, Song) and which uses floating-point numbers [7].

Generally, CKKS works best for real number computations and can be applied to machine learning applications as it can implement logistic regression methods and other statistical computations. DM (also known as FHEW) and CGGI (also known as TFHE) are useful in the application of Boolean circuits for small integers. BGV and BFV are generally used in applications with small integer values.

2.1 Public key or symmetric key

Homomorphic encryption can be implemented either with a symmetric key or an asymmetric (public) key. With symmetric key encryption, we use the same key to encrypt as we do to decrypt, whereas, with an asymmetric method, we use a public key to encrypt and a private key to decrypt. In Figure 1 we use asymmetric encryption with a public key (pk) and a private key (sk). With this Bob, Alice and Peggy will encrypt their data using the public key to produce ciphertext, and then we can operate on the ciphertext using arithmetic operations. The result can then be revealed by decrypting with the associated private key. In Figure 2 we use symmetric key encryption, and where the data is encrypted with a secret key, and which is then used to decrypt the data. In this case, the data processor (Trent) should not have access to the secret key, as they could decrypt the data from the providers.

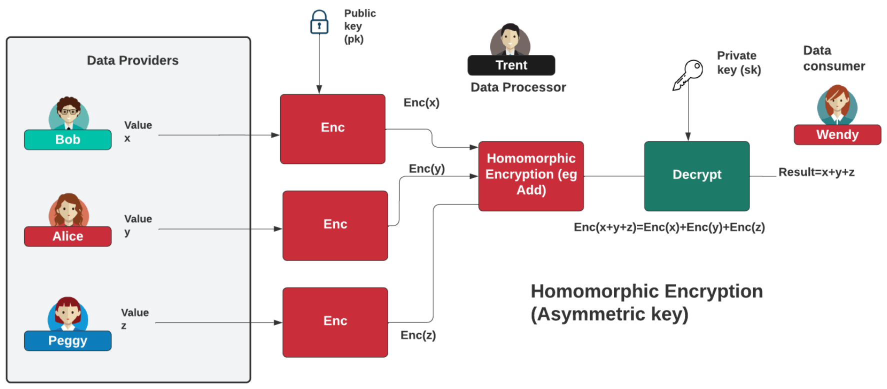

Fig. 1. Asymmetric encryption (public key)

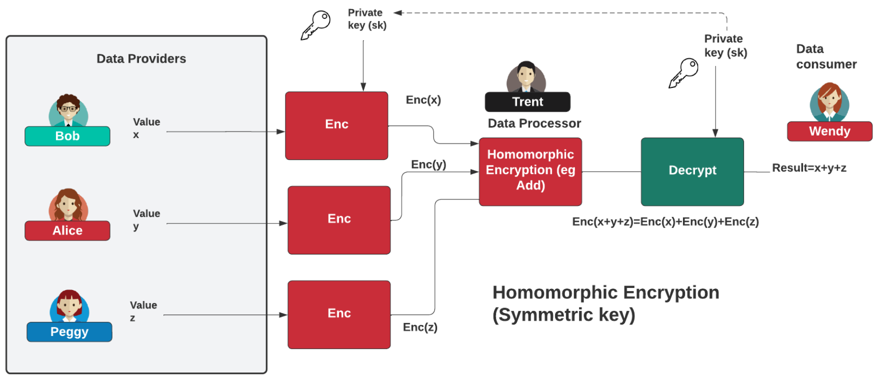

Fig. 2. Symmetric encryption

2.2 Homomorphic libraries

There are several homomorphic encryption libraries that support FHE, including ones that support CUDA and GPU acceleration, but many have not been kept up-to-date with modern methods or have only integrated one method. Overall, the native language libraries tend to be the most useful, as they allow the compilation to machine code. The main languages used for this are C++, Golang, and Rust, although some Python libraries exist through wrappers of C++ code. This includes HEAAN-Python, and its associated HEAAN library.

One of the first libraries which supported a range of methods is Microsoft SEAL [8], SEAL-C# and SEAL-Python. While it supports a wide range of methods, including BGV/BFV and CKKS, it has lacked any real serious development for the past few years. It does have support for Android and has a Node.js port [9]. Wood et al. [10] define a full range of libraries. One of the most extensive libraries is PALISADE, and which has now developed into OpenFHE. Within OpenFHE. The main implementations is this library are:

– Brakerski/Fan-Vercauteren (**BFV**) scheme for integer arithmetic
– Brakerski-Gentry-Vaikuntanathan (**BGV**) scheme for integer arithmetic
– Cheon-Kim-Kim-Song (**CKKS**) scheme for real-number arithmetic (includes approximate bootstrapping)
– Ducas-Micciancio (**DM**) and Chillotti-Gama-Georgieva-Izabachene (**CGGI**) schemes for Boolean circuit evaluation.

2.3 Bootstrapping

A key topic within fully homomorphic encryption is the usage of bootstrapping. Within a learning with-errors approach, we add noise to our computations. For a normal decryption process, we use the public key to encrypt data and then the associated private key to decrypt it. Within the bootstrap version of homomorphic encryption, we use an encrypted version of the private key that operates on the ciphertext. In this way, we remove the noise which can build up in the computation. Figure 3 outlines that we perform an evaluation on the decryption using an encrypted version of the private key. This will remove noise in the ciphertext, after which we can then use the actual private key to perform the decryption.

The main bootstrapping methods are CKKS [7], DM [11]/CGGI, and BGV/BFV. Overall, CKKS is generally the fastest bootstrapping method, while DM/CGGI is efficient with the evaluation of arbitrary functions. These functions approximate math functions as polynomials (such as with Chebyshev approximation). BGV/BFV provides reasonable performance and is generally faster than DM/CGGI but slower than CKKS.

2.4 Arbitrary smooth functions

With approximation theory, it is possible to determine an approximate polynomial $p(x)$ that is an approximation to a function $f(x)$. A polynomial takes the

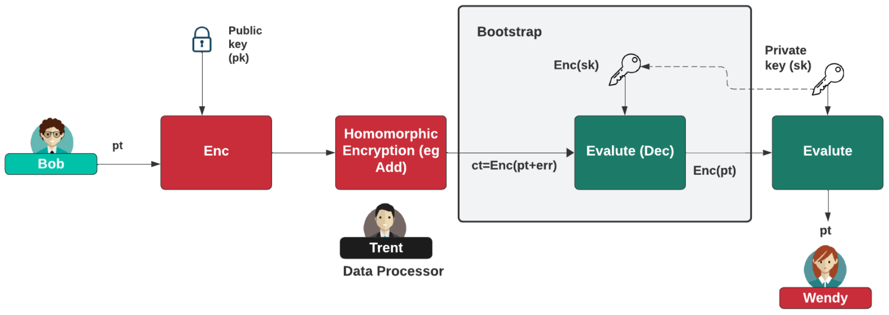

Fig. 3. Bootstrap

form of $p(x) = a_n.x^n + a_{n-1}.x^{n-1} + ... + a_1.x + a_0$, and where a_0... a_n are the coefficients of the powers, and n is the maximum power of the polynomial.

For this, we can define arbitrary smooth functions for CKKS using Chebyshev approximation [12]. These were initially created by Pafnuty Lvovich Chebyshev. This method involves the approximation of a smooth function using polynomials. Examples of these functions include log_{10}, log_2, log_e, and e^x [13].

2.5 Plaintext slots

With many homomorphic methods, we can encrypt multiple plaintext values into ciphertext in a single operation. This is defined as the number of plaintext slots, and is illustrated in Figure 4.

2.6 BGV and BFV

With BGV and BFV, we use a Ring Learning With Errors (LWE) method [6]. With BGV, we define a moduli (q), which constrains the range of the polynomial coefficients. Overall, the methods use a moduli, which can be defined within different levels. We then initially define a finite group of $\mathbb{Z}_q$, and then make this a ring by dividing our operations with $(x^n + 1)$ and where $n - 1$ is the largest power of the coefficients. The message can then be represented in binary as:

$$m = a_{n-1}a_{n-2}...a_0 \tag{1}$$

This can be converted into a polynomial with:

$$\mathbf{m} = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_1x + a_0 \pmod q \tag{2}$$

The coefficients of this polynomial will then be a vector. Note that for efficiency, we can also encode the message with ternary (such as with -1, 0 and 1). We then define the plaintext modulus with:

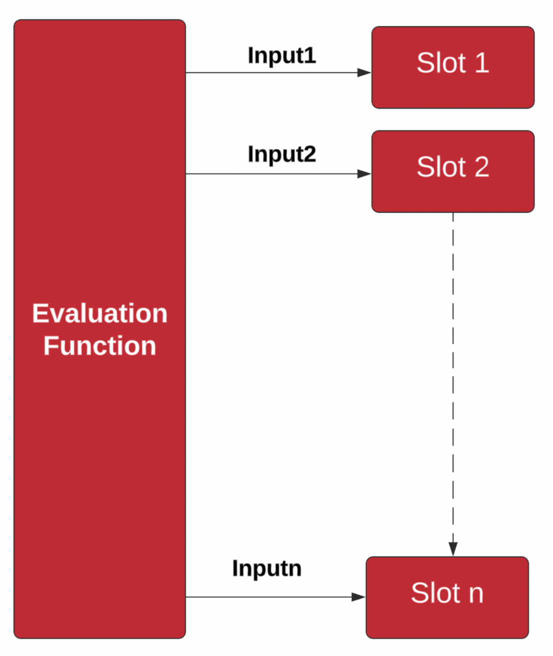

Fig. 4. Slots for plaintext

$$t = p^r \tag{3}$$

and where p is a prime number and r is a positive number. We can then define a ciphertext modulus of q, and which should be much larger than t. To encrypt with the private key of $\mathbf{s}$, we implement:

$$(c_0, c_1) = \left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s} + e, -\mathbf{a}\right) \mod q \tag{4}$$

To decrypt:

$$m = \lfloor \frac{t}{q}(c_0 + c_1).\mathbf{s} \rceil \tag{5}$$

This works because:

$$m_{recover} = \lfloor \frac{t}{q}\left(\frac{q}{t}.\mathbf{m} + \mathbf{a}.\mathbf{s} + e - \mathbf{a}.\mathbf{s}\right) \rceil \tag{6}$$

$$= \lfloor \left(\mathbf{m} + \frac{t}{q}.e\right) \rceil \tag{7}$$

$$\approx m \tag{8}$$

$$\tag{9}$$

For two message of m_1 and m_2, we will get:

$$Enc(m_1 + m_2) = Enc(m_1) + Enc(m_2) \tag{10}$$

$$Enc(m_1.m_2) = Enc(m_1).Enc(m_2) \tag{11}$$

Noise and computation But each time we add or multiply, the error also increases. Thus bootstrapping is required to reduce the noise. Overall, addition and plaintext/ciphertext multiplication is not a time-consuming task, but ciphertext/ciphertext multiplication is more computationally intensive. The most computational task is typically the bootstrapping process, and the ciphertext/-ciphertext multiplication process adds the most noise to the process.

Parameters We thus have a parameter of the ciphertext modulus (q) and the plaintext modulus (t). Both of these are typically to the power of 2. An example of q is 2^{240} and for t is 65,537. As the value of 2^q is likely to be a large number, we typically define it as a log_q value. Thus, a ciphertext modulus of 2^{240} will be 240 as defined as a log_q value.

2.7 CKKS

HEAAN (Homomorphic Encryption for Arithmetic of Approximate Numbers) defines a homomorphic encryption (HE) library proposed by Cheon, Kim, Kim and Song (CKKS). The CKKS method uses approximate arithmetics over complex numbers [7]. Overall, it is a levelled approach that involves the evaluation of arbitrary circuits of bounded (pre-determined) depth. These circuits can include ADD (X-OR) and Multiply (AND).

HEAAN uses a rescaling procedure to measure the size of the plaintext. It then produces an approximate rounding due to the truncation of the ciphertext into a smaller modulus. The method is especially useful in that it can be applied to carry out encryption computations in parallel. Unfortunately, the ciphertext modulus can become too small, and where it is not possible to carry out any more operations.

The HEAAN (CKKS) method uses approximate arithmetic over complex numbers ($\mathbb{C}$) and is based on Ring Learning With Errors (RLWE). It focuses on defining an encryption error within the computational error that will happen within approximate computations. We initially take a message (M) and convert it to a cipher message (ct) using a secret key sk. To decrypt ($[ct,sk]q$), we produce an approximate value along with a small error (e).

Craig Gentry [14] has outlined three important application areas within privacy-preserving genome association, neural networks, and private information retrieval. Along with this, he proposed that the research community should investigate new methods which did not involve the usage of lattices.

Chebyshev approximation With approximation theory, it is possible to determine an approximate polynomial $p(x)$ that is an approximation to a function $f(x)$. A polynomial takes the form of $p(x) = a_n.x^n + a_{n-1}.x^{n-1} + a_1.x + a_0$, and where $a_0...a_n$ are the coefficients of the powers, and n is the maximum power of the polynomial. In this case, we will evaluate arbitrary smooth functions for CKKS and use Chebyshev approximation. These were initially created by Pafnuty Lvovich Chebyshev. This method involves the approximation of a smooth function using polynomials.

Overall, with polynomials, we convert our binary values into a polynomial, such as 101101 is:

$$x^5 + x^3 + x^2 + 1 \tag{12}$$

Our plaintext and ciphertext are then represented as polynomial values.

Approximation theory With approximation theory, we aim to determine an approximate method for a function f(x). It was Pafnuty Lvovich Chebyshev who defined a method of finding a polynomial p(x) that is approximate for f(x). Overall, a polynomial takes the form of:

$$p(x) = a_n.x^n + a_{n-1}.x^{n-1} + a_1.x + a_0 \tag{13}$$

and where $a_0...a_n$ are the coefficients of the powers, and n is the maximum power of the polynomial. Chebyshev published his work in 1853 as "Theorie des mecanismes, connus sous le nom de parallelogrammes". His problem statement was to determine the deviations which one has to add to get an approximated value for a function f, given by its expansion in powers of $x - a$, if one wants to minimise the maximum of these errors between $x = a - h$ and $x = a + h$, h being an arbitrarily small quantity".

2.8 Polynomial evaluations

A polynomial takes the form form of $p(x) = a_n.x^n + a_{n-1}.x^{n-1} + a_1.x + a_0$, and where $a_0...a_n$ are the coefficients of the powers, and n is the maximum power of the polynomial. With CKKS in OpenFHE, we can evaluate the result of a polynomial for a given range of x values. For example, if we have $p(x) = 5.x^2 + 3.x + 7$ will give a result of $p(2) = 33$.

3 Related work

Homomorphic encryption supports the usage of machine learning methods, and some core features include a dot product operation with an encrypted vector and logistic functions. With this, OpenFHE supports a range of relevant methods and even has a demonstrator for a machine learning method.

3.1 State-of-the-art

Iezzi et al. [15] define two methods of training with homomorphic encryption:

– Private Prediction as a Service (PPaaS). This is where the prediction is outsourced to a service provider who has a pre-trained model and where encrypted data is sent to the service provider. In this case, the data owner does not learn the model used.
– Private Training as a Service (PTaaS). This is where the data owner provides data to a service provider and who will train the model. The service provider can then provide a prediction for encrypted data.

Wood et al [10] adds models of:

– Private outsourced computation. This involves moving computation into the cloud.
– Private prediction. This involves homomorphic data processed into the cloud, and not having access to the training model.
– Private training. This is where a cloud entity trains a model based on the client's data.

3.2 Basic primitives

Logistic Function With homomorphic encryption, we can represent a mathematical operation in the form of a homomorphic equation. One of the most widely used methods is to use Chebyshev polynomials, and which allows the mapping of the function to a Chebyshev approximation. A core application of the logistic function - also known as the sigmoid function - is within machine learning. With this, an artificial neural network is created with weighted summation and a sigmoid function (Figure 5). Mathematically, this is defined as:

$$f(x) = \frac{1}{1+e^{-x}} \tag{14}$$

This is supported in OpenFHE, and which implements Chebyshev approximation. For this, we can use the function of [16] and which evaluates 1/(1 + exp(-x)) for f(x), and where x is a range of coefficients with ciphertext. The value of a is the lower bound of the coefficients, and b is the upper bound. The degree value is the desired degree of approximation.

Logistic regression is often used to predict binary outcomes of whether patients need treatment in medical applications, such as with diabetic patients [17].

Inner product The inner product of two vectors of a and b is represented by a, b. It is the dot product of two vectors and represented as $a, b = |a|.|b|.cos(\theta)$, where θ is the angle between the two vectors. This operation is supported in OpenFHE [18].

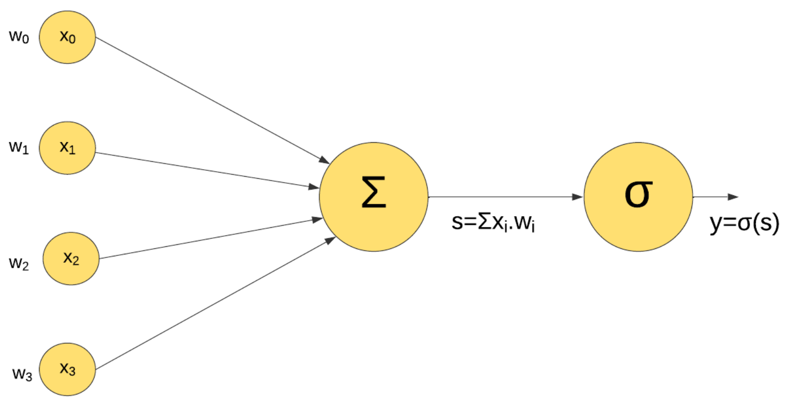

Fig. 5. Sigmoid function

Matrix operations We can perform matrix operations with an encrypted input vector from OpenFHE [19]. For this, if we have a vector of the form:

$$v_1 = \begin{bmatrix} x_1 \ x_2 \ x_3 \end{bmatrix} \tag{15}$$

and a matrix of:

$$m_1 = \begin{bmatrix} w_{11} \ w_{21} \ w_{31} \\ w_{12} \ w_{22} \ w_{32} \\ w_{13} \ w_{23} \ w_{33} \end{bmatrix} \tag{16}$$

We now get:

$$v_1.m_1 = \begin{bmatrix} x_1 \ x_2 \ x_3 \end{bmatrix} \begin{bmatrix} w_{11} \ w_{21} \ w_{31} \\ w_{12} \ w_{22} \ w_{32} \\ w_{13} \ w_{23} \ w_{33} \end{bmatrix} \tag{17}$$

and:

$$v_1.m_1 = \begin{bmatrix} x_1.w_{11} + x_2.w_{21} + x_3.w_{31} \ x_1.w_{12} + x_2.w_{22} + x_3.w_{32} \ x_1.w_{13} + x_2.w_{23} + x_3.w_{33} \end{bmatrix} \tag{18}$$

Thus we get:

$$y_1 = x_1.w_{11} + x_2.w_{21} + x_3.w_{31} y_2 = x_1.w_{12} + x_2.w_{22} + x_3.w_{32} y_3 = x_1.w_{13} + x_2.w_{23} + x_3.w_{33} \tag{19}$$

Figure 6 shows this setup.

3.3 GWAS

Blatt et al. [20] implemented the Genome-wide association study (GWAS) and which is a secure large-scale genome-wide association study using homomorphic encryption.

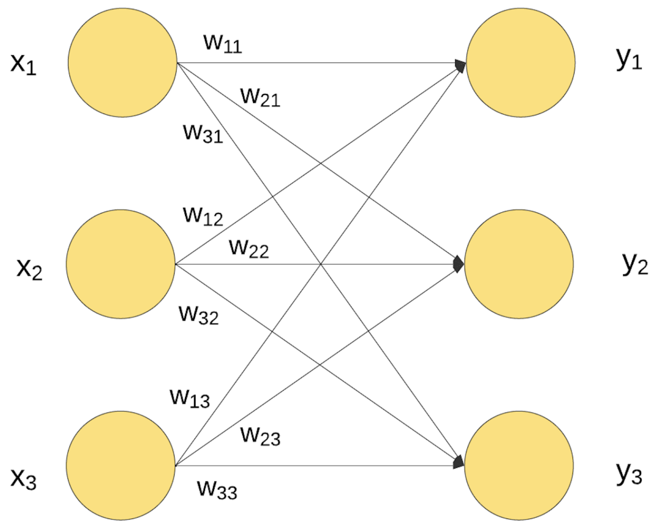

Fig. 6. Neural network

Chi-Square GWAS The Chi-squared GWAS test has been implemented in OpenFHE Here. With this, each of the participants in the student group is given a public key from a GWAS (Genome-wide association studies) coordinator, who then encrypts the data with CKKS and sends it back for processing. The computation includes association statistics using full logistic regression on each variant with sex, age, and age squared as covariates. Pearsons chi-square test uses categories to determine if there is a significant difference between sets of data [1].

$$\tilde{\chi}^2 = \frac{1}{d} \sum_{k=1}^{n} \frac{(O_k - E_k)^2}{E_k} \tag{20}$$

and where:

- $\tilde{\chi}^2$ is the chi-square test statistic.
- O is the observed frequency.
- E is the expected frequency.

Overall, the implementation involved a dataset of 25,000 individuals, and it was shown that 100,000 individuals and 500,000 single-nucleotide polymorphisms (SNPs) could be evaluated in 5.6 hours on a single server [20].

[1] It implements as RunChi2 from https://github.com/openfheorg/openfhe-genomic-examples/blob/main/demo-chi2.cpp

Linear Regression The GWAS method is also implemented with linear regression for homomorphic encryption (See RunLogReg in Here). The results show that the accuracy of both the Chi-squared and linear regression tests was good. The run time varied linearly with the number of participants in the test.

3.4 Support Vector Machines (SVM)

With the SVM (Support Vector Machine) model, we have a supervised learning technique. Overall, it is used to create two categories (binary) or more (multi) and will try to allocate each of the training values into one or more categories. Basically, we have points in a multidimensional space and try to create a clear gap between the categories. New values are then placed within one of the two categories.

Overall, we split out the input data into training and test data and then train with a sklearn model with unencrypted values from the training data. The output from the model is the weights and intercepts. Next, we can encrypt the test data with the homomorphic public key and then feed this into the SVM model. The output values can then be decrypted by the associated private key, as illustrated in Figure 7.

CKKS and SVM The CKKS scheme is a homomorphic encryption method designed for encrypted arithmetic operations. For a given plaintext feature vector of:

$$\mathbf{x} = (x_1, x_2, \ldots, x_n) \tag{21}$$

and a public key of pk, the encryption function is:

$$\mathsf{Enc}(\mathbf{x}, \mathsf{pk}) = c_x \tag{22}$$

and where c_x is the encrypted representation of $\mathbf{x}$ [7]. For Support Vector Machine (SVM) classification with Linear SVM, we use a **linear decision function** of:

$$f_{\text{lin}}(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b \tag{23}$$

and where $\mathbf{x}$ is the feature vector, $\mathbf{w}$ is the weight vector, and b is the bias term.

For classification:

$$y = \text{sign}(f_{\text{lin}}(\mathbf{x})) \tag{24}$$

Using FHE, the computation is performed on encrypted values [7]:

$$\mathsf{Enc}(f_{\text{lin}}(\mathbf{x})) = \mathsf{Enc}(\mathbf{w}^T \mathbf{x} + b) \tag{25}$$

A Polynomial Kernel SVM extends the decision function to:

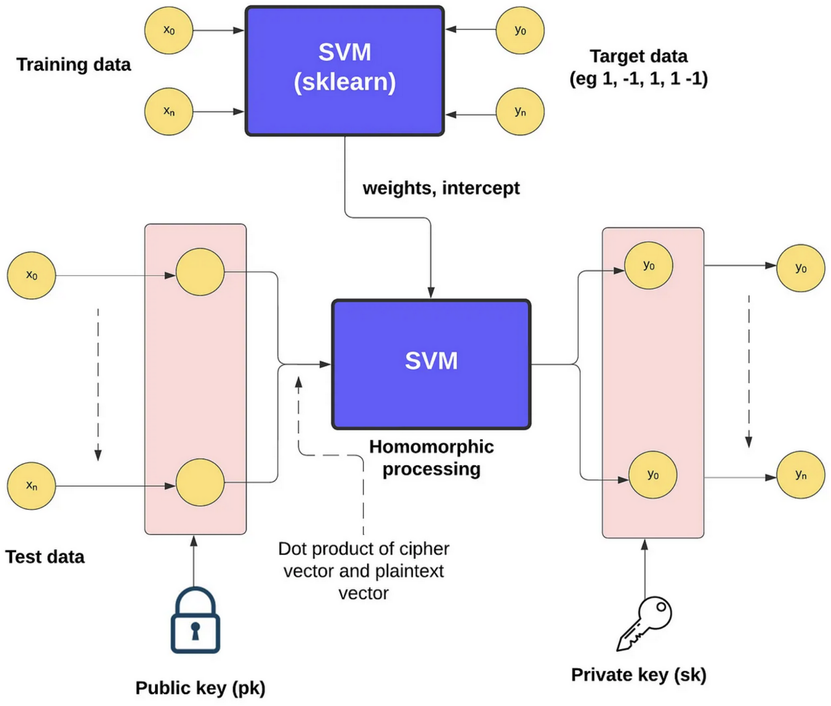
Training data
x0
xn
SVM (sklearn)
y0
yn
Target data (eg 1, -1, 1, 1 -1)
weights, intercept
x0
xn
Test data
SVM
Homomorphic processing
Dot product of cipher vector and plaintext vector
Public key (pk)
y0
y0
y0
yn
Private key (sk)

Fig. 7. SVM

$$f_{\text{poly}}(\mathbf{x}) = (\mathbf{w}^T \mathbf{x} + b)^d \tag{26}$$

where d is the polynomial degree, and where the rest of the parameters are the same as the Linear SVM. For classification:

$$y = \text{sign}(f_{\text{poly}}(\mathbf{x})) \tag{27}$$

With homomorphic encryption, we compute this function without decrypting:

$$\mathsf{Enc}(f_{\text{poly}}(\mathbf{x})) = \mathsf{Enc}((\mathbf{w}^T \mathbf{x} + b)^d) \tag{28}$$

and which follows prior encryption-based SVM work [14,7]. After computation, the result is decrypted using the **private key** (sk:

$$\mathsf{Dec}(c_f, \mathsf{sk}) = f(\mathbf{x}) \tag{29}$$

The final classification is:

$$y = \text{sign}(\mathsf{Dec}(c_f, \mathsf{sk})) \tag{30}$$

4 Methodology

This paper explores the integration of Fully Homomorphic Encryption (FHE) with Support Vector Machines (SVM) for privacy-preserving machine learning. The proposed framework employs the CKKS encryption scheme, implemented via the OpenFHE library, to enable encrypted inference while maintaining classification accuracy. Homomorphic encryption allows computations to be performed directly on encrypted data without decryption, ensuring data privacy throughout the machine-learning pipeline [7]. OpenFHE is an open-source library that provides implementations of lattice-based encryption schemes, including CKKS, which supports approximate arithmetic operations on encrypted data [21].

The dataset used for this experiment is the Iris dataset, which contains measurements of iris flowers, including sepal length, sepal width, petal length, and petal width [22]. This dataset, originally introduced by Fisher [23], is widely used in machine learning research due to its simplicity and well-separated class distributions. The objective of using this dataset is to evaluate the effectiveness of the encrypted SVM framework in classifying iris species while ensuring data privacy. The dataset is publicly available from the UCI ML Repository [24].

The methodology consists of environment setup and data preprocessing. The SVM model, originally introduced by Cortes and Vapnik [25], is trained on plaintext data before being used for encrypted inference. To analyse the trade-offs between encryption security, computational efficiency, and model accuracy, the implementation is conducted using *OpenFHE* within a Python-based machine-learning pipeline, leveraging libraries such as Scikit-Learn and NumPy [26].

4.1 Environment Setup

The encryption parameters in the Fully Homomorphic Encryption (FHE) framework are essential for balancing security, computational efficiency, and model accuracy. The following section outlines the installation process, provides a detailed explanation of each parameter, and presents the system specifications, as shown in Table 1.

Experimental Setup The implementation of the encrypted SVM framework followed a structured approach to ensure efficiency and reproducibility. The setup commenced with the installation of *openfhe-python*, adhering strictly to the official guidelines [1]. This library provided the essential cryptographic primitives required for executing encrypted computations securely.

Following the installation, the model training phase was conducted using the *model_training.py* script. This process involved training an SVM classifier and saving the learned parameters, which were subsequently utilised for encrypted inference. The trained model served as the foundation for performing secure classification without exposing sensitive data.

To facilitate a standardised evaluation, the dataset was organised within the *data/* directory. If necessary, the dataset could be regenerated by executing the *get_data.py* script, ensuring consistency and reproducibility across experiments.

For encrypted inference, two dedicated scripts were employed: *encrypted_svm_linear.py* and *encrypted_svm_poly.py*. These scripts enabled inference using linear and polynomial kernel SVM models, respectively, allowing for a comprehensive assessment of encrypted classification performance under different kernel settings. Through this structured approach, the framework effectively demonstrated the feasibility of privacy-preserving machine learning using homomorphic encryption.

Encryption Parameters Homomorphic encryption relies on several key parameters that impact computational efficiency, security, and accuracy [7]. The primary encryption parameters used in this study are:

- **Ring Dimension** (N): Defines the size of the polynomial ring used in encryption. A larger N increases security but also raises computational cost [27]. Typical values include $N = 2^{10}, 2^{12}, 2^{14}, \ldots$.
- **Multiplication Depth** (D): Represents the number of consecutive multiplications a ciphertext can undergo before noise accumulation becomes a limiting factor [14]. Higher D enables more complex computations, which is essential for polynomial kernel approximation in SVM.
- **Scaling Factor** (S): Determines the precision of fixed-point arithmetic in CKKS encryption. A higher S improves numerical accuracy but increases computational complexity [28].
- **First Modulus Size** (M): Defines the initial modulus size, impacting ciphertext precision and computational overhead [29].

- **Security Level (L)**: Specifies the cryptographic strength of encryption (e.g., 128-bit, 192-bit, 256-bit security). A higher L enhances security but introduces additional computational costs [30].
- **Batch Size (B)**: Represents the number of encrypted values processed in parallel. A larger B improves computational efficiency, particularly for batch inference [28].

These parameters significantly impact the feasibility of encrypted machine learning. In our experiments, we analyse their influence on classification accuracy, encryption overhead, and inference efficiency.

System Specifications The system was deployed on an AWS EC2 t3.medium instance, equipped with two virtual CPUs (Intel Xeon 3.1 GHz) and 4 GB of RAM, providing a balanced environment for machine learning and encrypted computations. For software, Python was used as the primary programming language, enabling seamless integration between machine learning and encryption frameworks. scikit-learn then supports the training and evaluation of traditional SVM models, ensuring a robust baseline for comparison. Meanwhile, OpenFHE is used to perform encryption, ciphertext operations, and homomorphic inference, thus enabling secure computation without compromising model performance.

Table 1: Experimental Setup

Component	**Description**
Compute Environment	AWS EC2 `t3.medium` (Two vCPUs, Intel Xeon 3.1 GHz, 4 GB RAM)
Operating System	Ubuntu 20.04
Programming Language	Python 3.x
ML Library	`scikit-learn` (for SVM training and evaluation) [31]
HE Library	OpenFHE (CKKS scheme for encrypted inference) [21]
Dataset	Iris Dataset (150 samples, four features) [23]
Preprocessing	Standardisation (zero mean, unit variance), Train-Test Split (80%-20%)
Encryption Parameters	N, D, S, M, L, B (Ring Dim, Mult Depth, Scaling Factor, Modulus Size, Sec Level, Batch Size)
SVM Models	Linear SVM, Polynomial SVM (homomorphic kernel approximation) [25]
Performance Metrics	Classification Accuracy, Encryption Overhead, Inference Time, Memory Usage, Scalability

4.2 Data Preprocessing

Data preprocessing is a crucial step to ensure reliable and efficient machine learning, particularly when incorporating Homomorphic Encryption [32]. In this study, we use the Iris dataset (150 samples, four features) and apply standardisation to achieve zero mean and unit variance, improving model stability. The data is then split into 80% training and 20% testing to enable fair evaluation. Given the constraints of encrypted computation, categorical features are appropriately encoded. These steps help maintain accuracy while minimising computational overhead in secure inference.

Dataset Overview The Iris dataset is a widely recognised benchmark in machine learning, frequently employed for evaluating classification algorithms [31]. It provides a structured framework for distinguishing between different iris flower species based on their physical attributes. The dataset consists of 150 samples, each representing an individual iris flower, and is characterised by four key features: sepal length, sepal width, petal length, and petal width, all measured in centimetres. These features enable effective classification by capturing the morphological differences among species.

The dataset comprises three distinct classes, each containing 50 samples, corresponding to three species: *Iris setosa* (label ‘0‘), *Iris versicolor* (label ‘1‘), and *Iris virginica* (label ‘2‘). It is well-structured, balanced, and contains no missing values, making it particularly suitable for both educational purposes and experimental evaluations in machine learning research.

Due to its simplicity and interpretability, the Iris dataset is commonly used for demonstrating data preprocessing techniques, exploratory data analysis, and classification models, including Support Vector Machines (SVM) and decision trees [25]. Furthermore, its features can be visualised through pair plots, enabling an intuitive understanding of feature relationships and class separability.

In this study, the Iris dataset serves as a controlled environment for analysing the effects of homomorphic encryption on SVM classification. By leveraging its structured nature, we facilitate a reliable comparison between traditional and encrypted inference methods, allowing for a comprehensive assessment of computational performance and classification accuracy.

The Iris dataset is a classic benchmark for machine learning algorithms, favored for its simplicity and accessibility. It is widely used in educational settings and can be easily accessed through libraries like `scikit-learn` in Python.

Data Preprocessing and Feature Encoding To ensure robust and efficient encrypted classification, the data set was subjected to a systematic preprocessing pipeline implemented by `get_data.py`. This process involved feature selection, transformation, and structuring to optimise the data for FHE-based machine learning.

The dataset contains 150 rows (samples) and four columns (four predictive features): sepal length, sepal width, petal length, petal width. There is one target column of species classification. Standardisation was applied to normalise values, ensuring a mean (μ) of approximately zero, and a standard deviation (σ) of approximately unity. This improves model performance by placing features on a similar scale. The data were split into 120 training samples and 30 test samples, ensuring a well-balanced split for model evaluation. Furthermore, categorical labels were encoded into numerical representations to facilitate seamless integration into the machine learning framework.

This preprocessing stage establishes a structured and standardised foundation for encrypted SVM training. Harmonising feature distributions, optimising data representation, and preparing the dataset for secure computation enhance both the accuracy and efficiency of privacy-preserving machine learning.

5 Implementation

This section presents the approach used to implement privacy-preserving classification using FHE. The Support Vector Machine (SVM) model is adapted to operate on encrypted data using the CKKS encryption scheme [7].

The implementation consists of dataset preprocessing, encryption of feature vectors, SVM training, and encrypted classification. The process follows established principles from privacy-preserving machine learning [33]. The overall workflow is visualised in Figure 8.

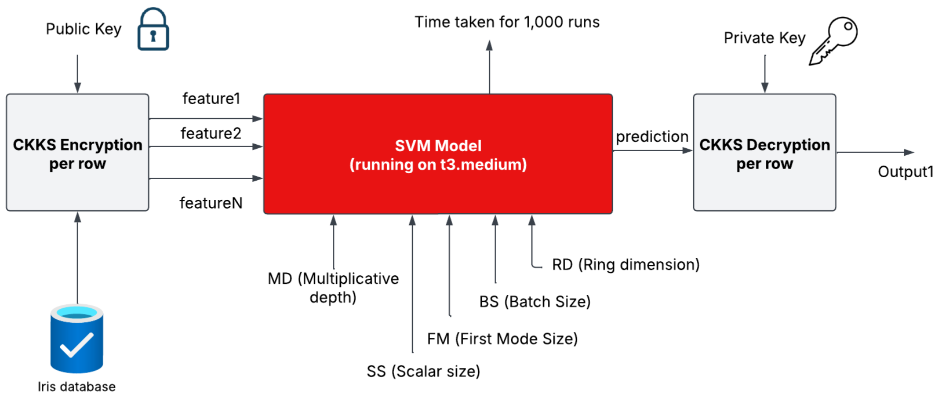

Fig. 8. Experimental setup for encrypted classification. The pipeline includes data encryption, encrypted inference, and decryption of results.

The evaluation of the impact of homomorphic encryption on machine learning performance involves a number of experiments measuring key performance metrics. Classification accuracy was assessed by comparing encrypted and non-encrypted inference, with the SVM model achieving high accuracy. Computation time was also analysed, including encryption, inference, and decryption durations. Additionally, the scale-up runtime was examined by calculating the ratio of non-encrypted to encrypted execution times.

Memory overhead was evaluated to determine the effect of homomorphic encryption on resource consumption, particularly memory usage. Finally, scalability was assessed by analysing performance variations as the ring dimension size and multiplication depth increased.

5.1 Encryption Using CKKS

The CKKS encryption scheme was employed to encrypt feature vectors, allowing privacy-preserving computations on floating-point values [7]. CKKS supports approximate arithmetic operations, making it well-suited for machine learning applications. The encryption parameters used in our experiments were selected based on a balance between computational efficiency and security. The multiplicative depth (D) ranged from 1 to 7, scaling factor (S) values varied between 10 and 50, and the first modulus size (M) was tested at 20, 30, 40, 50, and 60. The security level (L) was evaluated at 128-bit, 192-bit and 256-bit configurations. Batch sizes (B) included 128, 256, 512, 1024, 2048, and 4096. The ring dimension (N) was tested at 2^{14} (16,384), 2^{15} (32,768), 2^{16} (65,536), and 2^{17} (131,072), providing insights into the scalability of homomorphic encryption in machine learning.

5.2 Encrypted Classification Algorithm

In this work, we propose an FHE-based approach for SVM classification that supports both linear and polynomial kernels. The classification process involves encrypting the feature vector and model parameters, performing homomorphic computations to evaluate the decision function, and decrypting the result to obtain the classification outcome. The detailed steps are outlined in Algorithm 1, which describes the encrypted inference procedure for both linear and polynomial SVM models.

5.3 Model training

The model training example is shown in Appendix B.

6 Results

The experimental results provide a comprehensive evaluation of the impact of homomorphic encryption on SVM inference. A key observation is the trade-off between encryption depth and computational efficiency, where higher security parameters lead to increased execution time and memory consumption. This behaviour is consistent with the theoretical complexity of homomorphic encryption, which introduces overhead due to polynomial arithmetic and ciphertext expansion.

Algorithm 1 Homomorphic SVM Classification [14,7]

```
Require: Feature vector $\mathbf{x}$, public key $\mathsf{pk}$, private key $\mathsf{sk}$, degree $d$ (for polynomial SVM)
Ensure: Classification result $y$
 1: Encrypt Features: $c_x \leftarrow \mathsf{Enc}(\mathbf{x}, \mathsf{pk})$ (Equation 22)
 2: Encrypt Model Parameters:
 3:   $c_w \leftarrow \mathsf{Enc}(\mathbf{w}, \mathsf{pk})$
 4:   $c_b \leftarrow \mathsf{Enc}(b, \mathsf{pk})$
 5: Compute Encrypted Decision Function:
 6: if Linear SVM then
 7:     $c_f \leftarrow \mathsf{Enc}(\mathbf{w}^T\mathbf{x} + b)$ (Equation 25)
 8: elsePolynomial SVM
 9:     $c_f \leftarrow \mathsf{Enc}((\mathbf{w}^T\mathbf{x} + b)^d)$ (Equation 28)
10: end if
11: Decrypt the Result: $f(\mathbf{x}) \leftarrow \mathsf{Dec}(c_f, \mathsf{sk})$ (Equation 29)
12: Classify Output:
        $$y \leftarrow \begin{cases} 1, & f(\mathbf{x}) \geq 0 \\ -1, & f(\mathbf{x}) < 0 \end{cases} \quad \text{(Equation 30)}$$
13: return $y$
```

Beyond computational cost, the study examines the extent to which encrypted inference preserves classification accuracy. By systematically tuning encryption parameters, the analysis explores the balance between security and performance, offering insights into optimising privacy-preserving machine learning. The following sections present a detailed discussion of these findings, grounded in both empirical observations and theoretical considerations.

Tables 3 and 2 data gathered. MD is multiplicative depth, SS is scalar size, FM is first mod size, BS is batch size and RD is the ring dimension. AEA is Average Encryption Accuracy, NEA is Non-Encrypted Accuracy, AET is Average Encryption Time, and ANT is Average Non-Encryption Time.

Classification Accuracy Table 4 presents the classification accuracy of plaintext and encrypted SVM models. The results show that, in this experiment, homomorphic encryption has no significant impact on model accuracy, as both versions achieve similar performance. This confirms the effectiveness of the CKKS encryption scheme in preserving the integrity of machine learning inference.

Computational Overhead Homomorphic encryption introduces additional computational costs due to encryption, encrypted inference, and decryption steps. Table 5 compares execution times for plaintext and encrypted models.

The encrypted inference process is around 1,000 times slower than plaintext execution, primarily due to polynomial evaluations performed under encryption.

Table 2: Results for SVM-Linear

MD	SS	FM	SL	BS	RD	AEA	NEA	AET	ANT	Scale up
1	30	60	128	1,024	16,384	0.967	0.967	0.643458	0.000623	1,032.838
2	30	60	128	1,024	16,384	0.967	0.967	0.782	0.000067	1,172.735
3	30	60	128	1,024	16,384	0.967	0.967	0.924	0.000161	1,397.215
4	30	60	128	1,024	16,384	0.967	0.967	1.101	0.000613	1,794.548
5	30	60	128	1,024	16,384	0.967	0.967	1.283	0.000624	2,056.393
6	30	60	128	1,024	16,384	0.967	0.967	1.391	0.000627	2,097.537
7	30	60	128	1,024	16,384	0.967	0.967	1.530	0.000658	2,324.503
1	10	60	128	1,024	16,384	0.817	0.967	0.649	0.000067	9,697.24
1	20	60	128	1,024	16,384	0.967	0.967	0.672	0.000065	10,332.49
1	30	60	128	1,024	16,384	0.967	0.967	0.651	0.000073	8,919.69
1	40	60	128	1,024	16,384	0.967	0.967	0.650	0.000029	22,431.52
1	50	60	128	1,024	16,384	0.967	0.967	0.650	0.000027	24,084.56
1	30	20	128	1,024	16,384	0.967	0.967	0.627	0.000076	8,251.12
1	30	30	128	1,024	16,384	0.967	0.967	0.632	0.000067	9,434.24
1	30	40	128	1,024	16,384	0.967	0.967	0.635	0.000074	8,573.4
1	30	50	128	1,024	16,384	0.967	0.967	0.643	0.000065	9,905.05
1	30	60	128	1,024	16,384	0.967	0.967	0.641	0.000067	9,574.09
1	30	60	192	1,024	16,384	0.967	0.967	0.204	0.000184	1,108.59
1	30	60	256	1,024	16,384	0.967	0.967	0.197	0.000188	1,050.52
1	30	60	512	1,024	16,384	0.967	0.967	0.201	0.000191	1,050.31
1	30	60	1,024	1,024	16,384	0.967	0.967	0.198	0.000193	1,023.06
1	30	60	2,048	1,024	16,384	0.967	0.967	0.202	0.000193	1,048.5
1	30	60	4,096	1,024	16,384	0.967	0.967	0.647	0.000711	910.78
1	30	60	128	128	16,384	0.967	0.967	0.198	0.000109	1,817.4
1	30	60	128	256	16,384	0.967	0.967	0.195	0.000107	1,822.5
1	30	60	128	512	16,384	0.967	0.967	0.196	0.000109	1,808.7
1	30	60	128	1,024	16,384	0.967	0.967	0.641	0.000067	9,574.88
1	30	60	128	2,048	16,384	0.967	0.967	0.206	0.000109	1,878.6
1	30	60	128	4,096	16,384	0.967	0.967	0.213	0.000109	1,945.3
1	30	60	128	1,024	16,384	0.967	0.967	0.638	0.000662	963.82
1	30	60	128	1,024	32,768	0.967	0.967	1.265	0.000624	2,026.52
1	30	60	128	1,024	65,536	0.967	0.967	2.583	0.00067	3,646.68
1	30	60	128	1,024	131,072	0.967	0.967	5.103	0.00065	8,245.34

Table 3: Results for SVM-poly

MD	SS	FM	SL	BS	RD	AEA	NEA	AET	ANT	Scale up
1	30	60	128	1,024	16,384	0.967	0.967	0.648	0.000708	915.2
2	30	60	128	1,024	16,384	0.967	0.967	0.783	0.000730	1,093.3
3	30	60	128	1,024	16,384	0.967	0.967	0.919	0.000763	1,405.0
4	30	60	128	1,024	16,384	0.967	0.967	1.097	0.000629	1,527.7
5	30	60	128	1,024	16,384	0.967	0.967	1.288	0.000773	1,665.7
6	30	60	128	1,024	16,384	0.967	0.967	1.399	0.000712	1,954.9
7	30	60	128	1,024	16,384	0.967	0.967	1.562	0.000715	2,248.4
1	10	60	128	1,024	16,384	0.784	0.967	0.649	0.000067	973.8
1	20	60	128	1,024	16,384	0.967	0.967	0.671	0.000065	1,032.4
1	30	60	128	1,024	16,384	0.967	0.967	0.651	0.000073	889.4
1	40	60	128	1,024	16,384	0.967	0.967	0.650	0.000029	1,033.9
1	50	60	128	1,024	16,384	0.967	0.967	0.650	0.000027	1,036.4
1	30	20	128	1,024	16,384	0.967	0.967	0.627	0.000076	927.7
1	30	30	128	1,024	16,384	0.967	0.967	0.632	0.000067	940.1
1	30	40	128	1,024	16,384	0.967	0.967	0.634	0.000074	941.8
1	30	50	128	1,024	16,384	0.967	0.967	0.643	0.000065	998.1
1	30	60	128	1,024	16,384	0.967	0.967	0.641	0.000067	961.3
1	30	60	192	1,024	16,384	0.967	0.967	0.203	0.000184	1,090.9
1	30	60	256	1,024	16,384	0.967	0.967	0.197	0.000188	1,049.6
1	30	60	512	1,024	16,384	0.967	0.967	0.201	0.000191	1,052.2
1	30	60	1,024	1,024	16,384	0.967	0.967	0.197	0.000193	1,023.7
1	30	60	2,048	1,024	16,384	0.967	0.967	0.202	0.000193	1,048.9
1	30	60	4,096	1,024	16,384	0.967	0.967	0.646	0.000711	908.7
1	30	60	128	128	16,384	0.967	0.967	0.641	0.000067	961.3
1	30	60	128	256	16,384	0.967	0.967	0.204	0.000184	1,090.9
1	30	60	128	512	16,384	0.967	0.967	0.197	0.000188	1,049.6
1	30	60	128	1,024	16,384	0.967	0.967	0.201	0.000191	1,052.2
1	30	60	128	2,048	16,384	0.967	0.967	0.198	0.000193	1,023.7
1	30	60	128	4,096	16,384	0.967	0.967	0.202	0.000193	1,048.9
1	30	60	128	1,024	16,384	0.967	0.967	0.680	0.000747	910.6
1	30	60	128	1,024	32,768	0.967	0.967	1.269	0.000668	1,951.5
1	30	60	128	1,024	65,536	0.967	0.967	2.549	0.000604	3,935.1
1	30	60	128	1,024	131,072	0.967	0.967	5.245	0.000687	7,716.4

Table 4: Classification Accuracy Comparison

Model	**Accuracy (%)**
SVM (Plaintext)	96.7
SVM (Encrypted)	96.7

Table 5: Runtime Analysis (sec)

Operation	**Plaintext**	**Encrypted**
Feature Encryption	-	0.2029
Inference	0.0002	0.2029
Decryption	-	0.0001

Scalability and Resource Utilisation Scalability ensures stable performance as data grows, while resource utilisation optimises computational efficiency. Balancing encryption parameters helps maintain security, accuracy, and performance.

Impact of Ring Dimension Size The effect of increasing the ring dimension on encrypted inference time is shown in Table 6 and Figure 9. Larger ring dimensions increase computation time due to expanded ciphertext size.

Table 6: Homomorphic Scale-up with Varying Ring Dimensions

MD	**SS**	**FM**	**SL**	**BS**	**RD**	**SVM-Linear**	**SVM-Poly**
1	30	60	128	1024	16K	963.8	910.6
1	30	60	128	1024	32K	2,026.5	1,851.7
1	30	60	128	1024	64K	3,646.7	3,935.1
1	30	60	128	1024	128K	8,245.3	7,716.4

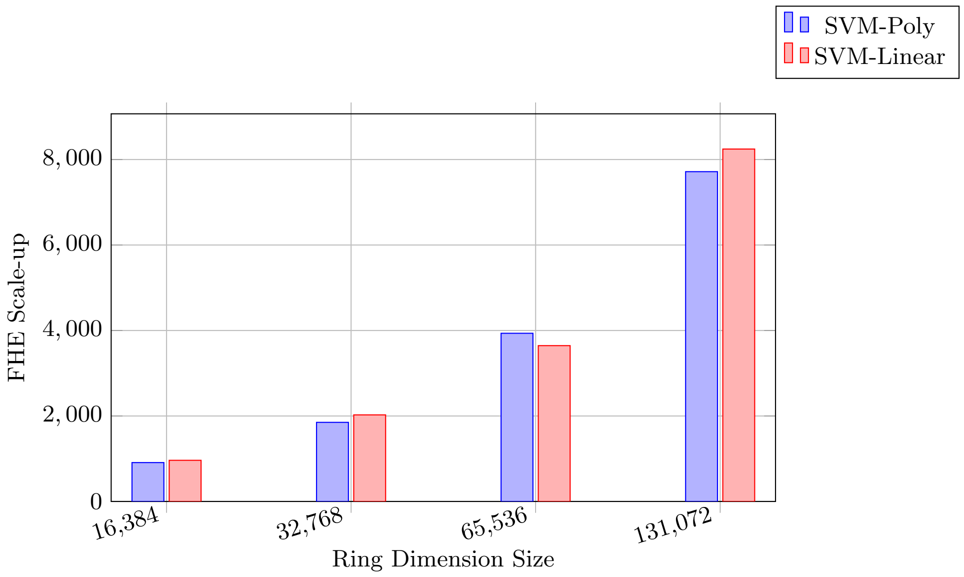

Fig. 9. Homomorphic Scale-up vs SVM-Linear and SVM-Poly Ring Dimension

Impact of Multiplication Depth on FHE Scale-up Table 7 presents the effect of increasing multiplication depth D on encrypted inference speed, as visualised in Figure 10.

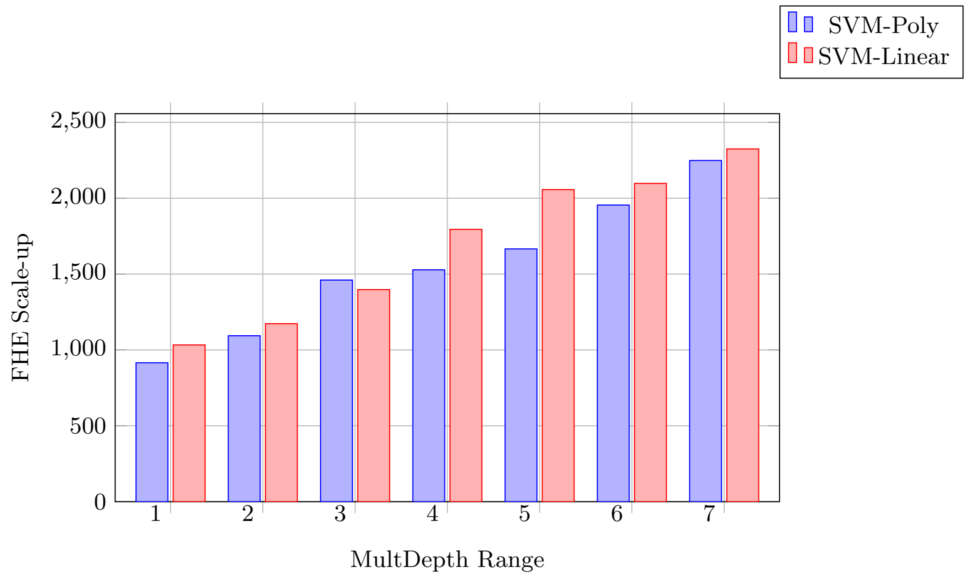

Fig. 10. Homomorphic Scale-up vs SVM-Linear and SVM-Poly MultDepth

Table 7: Homomorphic Scale-up for SVM-Linear and SVM-Poly with Varying Multiplication Depth

MD	**SS**	**FM**	**SL**	**BS**	**RD**	**SVM-Linear**	**SVM-Poly**
1	30	60	128	1024	16,384	1,032.8	915.2
2	30	60	128	1024	16,384	1,172.7	1,093.3
3	30	60	128	1024	16,384	1,397.2	1,460.5
4	30	60	128	1024	16,384	1,794.5	1,527.7
5	30	60	128	1024	16,384	2,056.4	1,665.7
6	30	60	128	1024	16,384	2,097.5	1,954.9
7	30	60	128	1024	16,384	2,324.5	2,248.4

The experimental results highlight key trade-offs in homomorphic encryption for SVM inference. Table 4 confirms that the encrypted SVM model maintains 96.7% accuracy, similar to the plaintext model, demonstrating that CKKS encryption does not affect classification performance. Table 5 shows that encrypted inference is approximately 1,000 times slower than plaintext inference, primarily due to polynomial evaluations under encryption. This slowdown arises from the computational complexity of homomorphic operations, particularly ciphertext multiplication and relinearisation [7]. Unlike plaintext arithmetic, where multiplication is a constant-time operation, homomorphic multiplication involves modular reductions, rescaling, and key-switching, leading to significant overhead [34].

Impact of Multiplication Depth Table 7 and Figure 10 reveal that increasing multiplication depth significantly raises execution time, with SVM-Poly scale-up increasing from 915.2s at depth 1 to 2248.4s at depth 7. The reason is that multiplication depth determines the number of sequential homomorphic multiplications that can be performed before bootstrapping is required [14]. As the depth increases:

- Noise Growth. Each multiplication amplifies noise, requiring frequent relinearisation and rescaling, which are computationally expensive [7].
-
- Exponentially Larger Ciphertexts. Higher-depth computations require larger ciphertext modulus values to maintain correctness, increasing memory and computation costs [35].
-
- Bootstrapping Overhead. If the noise exceeds the threshold, a bootstrapping step is needed, which further increases execution time [34,36].

These findings emphasise the need for optimisation strategies, including ciphertext packing, bootstrapping, and hardware acceleration, to improve the feasibility of encrypted machine learning in real-world applications.

Limitations and Future Work While the proposed approach demonstrates promising results, certain limitations must be addressed to enhance its practical applicability. The most significant challenge lies in the high computational cost of homomorphic encryption, which leads to substantial execution time overhead. This limitation poses a significant barrier to real-time applications, particularly in scenarios where rapid inference is required. Furthermore, the large memory footprint associated with ciphertext storage presents scalability concerns, especially for deployment on resource-constrained devices.

The increased computational burden can be attributed to the underlying complexity of homomorphic encryption operations, which involve polynomial arithmetic over large integer rings [7]. The reliance on number-theoretic transforms (NTTs) for polynomial multiplication introduces an inherent O(nlogn)

O(nlogn) computational cost, while the quadratic complexity of matrix-vector operations within SVM classification further compounds execution time [14]. Additionally, the trade-off between multiplication depth and accuracy, dictated by the hardness of the Ring Learning With Errors (RLWE) problem, influences both performance and security [27].

To mitigate these challenges, future research should explore hardware acceleration techniques, such as leveraging GPUs and FPGAs, to enhance computational efficiency [37]. Additionally, optimising encryption parameterssuch as ring dimension size and coefficient modulus selectioncan significantly reduce latency and memory consumption [7]. Exploring alternative cryptographic schemes, such as hybrid encryption approaches (Typically, it merges a fast symmetric encryption scheme with a secure asymmetric encryption scheme to balance efficiency and security), may further improve the feasibility of encrypted machine learning in real-world applications [38].

7 Conclusion

The usage of homomorphic encryption within machine learning provides great hope for privacy-aware learning. Unfortunately, it will come with an overhead of processing. This paper shows that using an extracted SVM model provides an excellent method of creating a model which can then be used to process data. In order to understand the key parameters which affect performance, the paper evaluates multiplication depth, scale size, first modulus size, security level, batch size, and ring dimension, along with two different SVM models, SVM-Poly and SVM-Linear. Overall, the results show that the two main parameters which affect performance are the ring dimension and the modulus size, and that SVM-Poly and SVM-Linear show similar performance levels.

8 Appendix A

In applying our SVM implementation, we can use sklearn to train the model Here:

```
import pandas as pd
import numpy as np
from sklearn.svm import SVC

# Load the data
X_train =pd.read_csv('data/credit_approval_train.csv')
X_test =pd.read_csv('data/credit_approval_test.csv')
y_train =pd.read_csv('data/credit_approval_target_train.csv')
y_test =pd.read_csv('data/credit_approval_target_test.csv')

# Model Training
print("---- Starting Models Training ----")
print("Starting SVM Linear")
svc_linear =SVC(kernel='linear')
svc_linear.fit(X_train, y_train.values.ravel())
print("SVM Linear Completed")

svc_poly =SVC(kernel='poly',degree=3,gamma=2)
svc_poly.fit(X_train, y_train.values.ravel())
print("SVM Poly Completed")

print("---- Model Training Completed! ----")

decision_function =svc_linear.decision_function(X_test)
ytestscore =decision_function[0]

decision_function_poly =svc_poly.decision_function(X_test)
ytestscore_poly =decision_function_poly[0]

# Saving Results
np.savetxt("models/weights.txt", svc_linear.coef_)
np.savetxt("models/intercept.txt", svc_linear.intercept_)
np.savetxt("data/ytestscore.txt", [ytestscore])
np.savetxt("models/dual_coef.txt", svc_poly.dual_coef_)
np.savetxt("models/support_vectors.txt", svc_poly.support_vectors_)
np.savetxt("models/intercept_poly.txt", svc_poly.intercept_)
np.savetxt("data/ytestscore_poly.txt", [ytestscore_poly])
```

This splits the input data into training and test data. The training data is then used to train the model with a linear and a polynomial SVM training model. It then outputs the model with a number of weights and intercept values. Next, we can run our homomorphic encryption method and take the training data (x), the weights, and the bias for processing [here]:

```
pt_x =cc.MakeCKKSPackedPlaintext(x)
pt_weights =cc.MakeCKKSPackedPlaintext(weights.tolist())
pt_bias =cc.MakeCKKSPackedPlaintext([intercept])
```

These values remain as plaintext values. We can then encrypt the training data with the public key [here]:

```
ct_x =cc.Encrypt(keys.publicKey, pt_x)
```

We then create an inner product with the cipher training data and the weights [here]:

```
ct_res =cc.EvalInnerProduct(ct_x, pt_weights,n)
```

An example of implementing a dot product is here. The output is then the multiplication (inner product) of the cipher values of the training data and the weights. Next, we can mask out the first value with:

```
mask =[0] *n
```

```
mask[0] =1
pt_mask =cc.MakeCKKSPackedPlaintext(mask)
ct_res =cc.EvalMult(ct_res, pt_mask)
```

Then we add the bias:

```
ct_res =cc.EvalAdd(ct_res, pt_bias)
```

Finally, we can decrypt the resultant value with the private key:

```
result =cc.Decrypt(ct_res, keys.secretKey)
```

9 Appendix B

The model training process was executed using the following command to train and save the model weights. The encrypted model files were subsequently used for inference:

```
python model_training.py
```

Upon execution, the following output was produced:

```
---- Starting Models Training ----
Starting SVM Linear
SVM Linear Completed
Starting SVM Poly
SVM Poly Completed
---- Model Training Completed! ----
All results saved successfully!
```

The dataset required for training is located in the data/ directory. However, to regenerate the dataset, the following command was executed:

```
python get_data.py
```

The script selected the following features:

```
Total number of features in dataset: 4
Selected: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

The original dataset contained 150 samples with 4 features:

```
Original data shape: (150, 4)
Features: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
```

Standardisation was applied, producing the following results:

```
Standardisation results:
sepal length (cm): mean=-0.000, std=1.003
sepal width (cm): mean=-0.000, std=1.003
petal length (cm): mean=-0.000, std=1.003
petal width (cm): mean=-0.000, std=1.003
```

The dataset was then split into training and testing sets:

```
Data processing completed successfully!

Saved files:
Training samples: 120
Testing samples: 30
Number of selected features: 4
```

These experiments provide insights into the trade-offs between security and computational efficiency in privacy-preserving machine learning. The execution of encrypted inference using

```
encrypted_svm_linear.py
```

produced the following output:

```
---- Testing OpenFHE Encryption ----
Original data: [1.5, 2.0, 3.5]
Decrypted values (real part): [1.4999997346263885, 1.9999997120806627, 3.499999428471009]
---- Running Encrypted Linear SVM ----
Avg Encrypted SVM Accuracy: 0.9667
Avg Non-Encrypted SVM Accuracy: 0.9667
Avg Encrypted Time: 0.2029 sec
Avg Non-Encrypted Time: 0.0002 sec
```

References

1. O. D. Team, “Openfhe: Open-source fully homomorphic encryption library,” GitHub Repository, 2023, https://github.com/openfheorg/openfhe-development.
2. R. L. Rivest, L. Adleman, M. L. Dertouzos *et al.*, “On data banks and privacy homomorphisms,” *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.
3. W. J. Buchanan, “Openfhe,” https://github.com/openfheorg/openfhe-development, OpenFHE, 2024, accessed: Feb 20, 2025. [Online]. Available: https://github.com/openfheorg/openfhe-development
4. C. Gentry, “A fully homomorphic encryption scheme,” 2009, crypto.stanford.edu/craig.
5. M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, “Fully homomorphic encryption over the integers,” in *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29.* Springer, 2010, pp. 24–43.

6. Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," *SIAM Journal on computing*, vol. 43, no. 2, pp. 831–871, 2014.
7. J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer, 2017, pp. 409–437.
8. W. J. Buchanan, "Homomorphic encryption (seal)," https://asecuritysite.com/seal, Asecuritysite.com, 2024, accessed: September 04, 2024. [Online]. Available: https://asecuritysite.com/seal
9. ——, "Homomorphic encryption with bfv using node.js," https://asecuritysite.com/seal/js_homomorphic, Asecuritysite.com, 2025, accessed: February 28, 2025. [Online]. Available: https://asecuritysite.com/seal/js_homomorphic
10. A. Wood, K. Najarian, and D. Kahrobaei, "Homomorphic encryption for machine learning in medicine and bioinformatics," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–35, 2020.
11. L. Ducas and D. Micciancio, "Fhew: bootstrapping homomorphic encryption in less than a second," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2015, pp. 617–640.
12. A. Al Badawi and Y. Polyakov, "Demystifying bootstrapping in fully homomorphic encryption," *Cryptology ePrint Archive*, 2023.
13. W. J. Buchanan, "Chebyshev approximations using openfhe and c++ (logarithm methods)," https://asecuritysite.com/openfhe/openfhe_18cpp, Asecuritysite.com, 2024, accessed: September 04, 2024. [Online]. Available: https://asecuritysite.com/openfhe/openfhe_18cpp
14. C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the forty-first annual ACM symposium on Theory of computing*, 2009, pp. 169–178.
15. M. Iezzi, "Practical privacy-preserving data science with homomorphic encryption: an overview," in *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2020, pp. 3979–3988.
16. W. J. Buchanan, "Logistic (sigmoid) function evaluation using openfhe and c++," https://asecuritysite.com/openfhe/openfhe_20cpp, Asecuritysite.com, 2024, accessed: September 06, 2024. [Online]. Available: https://asecuritysite.com/openfhe/openfhe_20cpp
17. R. Bender and U. Grouven, "Ordinal logistic regression in medical research," *Journal of the Royal College of physicians of London*, vol. 31, no. 5, p. 546, 1997.
18. W. J. Buchanan, "Ckks inner product using openfhe and c++," https://asecuritysite.com/openfhe/openfhe_13cpp, Asecuritysite.com, 2024, accessed: September 05, 2024. [Online]. Available: https://asecuritysite.com/openfhe/openfhe_13cpp
19. ——, "Matrix multiplication with homomomorphic encryption for bfv using openfhe and c++," https://asecuritysite.com/openfhe/openfhe_26cpp, Asecuritysite.com, 2024, accessed: September 06, 2024. [Online]. Available: https://asecuritysite.com/openfhe/openfhe_26cpp
20. M. Blatt, A. Gusev, Y. Polyakov, and S. Goldwasser, "Secure large-scale genome-wide association studies using homomorphic encryption," *Proceedings of the National Academy of Sciences*, vol. 117, no. 21, pp. 11 608–11 613, 2020.
21. O. Contributors, "Fully homomorphic encryption library," 2023, available at https://github.com/openfheorg/openfhe-development.
22. T. Nguyen, "Advancing privacy and accuracy with federated learning and homomorphic encryption," *Authorea Preprints*, 2023.

23. R. A. Fisher, “The use of multiple measurements in taxonomic problems,” *Annals of Eugenics*, vol. 7, no. 2, pp. 179–188, 1936.
24. U. M. L. Repository, “Iris dataset,” 2023, available at https://archive.ics.uci.edu/ml/datasets/iris.
25. C. Cortes and V. Vapnik, “Support-vector networks,” *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995.
26. A. Gron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd ed. O’Reilly Media, 2019.
27. Z. Brakerski, “Efficient fully homomorphic encryption from (standard) lwe,” *SIAM Journal on Computing*, vol. 43, no. 2, pp. 831–871, 2014.
28. J. H. Cheon, A. Kim, M. Kim, and Y. Song, “Numerical methods for homomorphic encryption,” *Cryptology ePrint Archive*, 2018.
29. M. Albrecht, R. Player, and S. Scott, “On the concrete hardness of learning with errors,” *Journal of Mathematical Cryptology*, vol. 12, no. 3, pp. 169–203, 2018.
30. A. Kim, Y. Song, and J. H. Cheon, “Batching methods for homomorphic encryption,” *Cryptology ePrint Archive*, 2018.
31. F. Pedregosa, G. Varoquaux, and A. e. a. Gramfort, “Scikit-learn: Machine learning in python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
32. J. P. Pinto, S. Kelur, and J. Shetty, “Iris flower species identification using machine learning approach,” in *2018 4th International Conference for Convergence in Technology (I2CT)*. IEEE, 2018, pp. 1–4.
33. F. Bourse, M. Minelli, M. Minihold, and P. Paillier, “Fast homomorphic evaluation of deep discretized neural networks,” in *Advances in Cryptology CRYPTO 2018*, ser. Lecture Notes in Computer Science, vol. 10992. Springer, 2018, pp. 483–512.
34. S. Halevi and V. Shoup, “Algorithms in helib,” *Advances in Cryptology CRYPTO 2014*, p. 554571, 2014.
35. M. S. Team, “Microsoft seal (simple encrypted arithmetic library),” 2022, available at https://www.microsoft.com/en-us/research/project/microsoft-seal/.
36. W. J. Buchanan and H. Ali, “Partial and fully homomorphic matching of ip addresses against blacklists for threat analysis,” *arXiv preprint arXiv:2502.16272*, 2025.
37. W. Dai, B. Deloingce, H. Chen, and K. Laine, “Accelerating fully homomorphic encryption using gpu,” *Proceedings of the 26th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, p. 8598, 2019.
38. Y. Mo, J. Liu, Y. Zhang, and K. Ren, “Efficient hybrid homomorphic encryption for encrypted machine learning,” *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, p. 16011614, 2022.