Efficient Logistic Regression on Large Encrypted Data

Kyoohyung Han[1], Seungwan Hong[1], Jung Hee Cheon[1], and Daejun Park[2]

[1] Seoul National University, Seoul, Republic of Korea
{satanigh, swanhong,jhcheon}@snu.ac.kr
[2] University of Illinois at Urbana-Champaign, Champaign, IL, USA
{dpark69}@illinois.edu

Abstract. Machine learning on encrypted data is a cryptographic method for analyzing private and/or sensitive data while keeping privacy. In the training phase, it takes as input an encrypted training data and outputs an encrypted model without using the decryption key. In the prediction phase, it uses the encrypted model to predict results on new encrypted data. In each phase, no decryption key is needed, and thus the privacy of data is guaranteed while the underlying encryption is secure. It has many applications in various areas such as finance, education, genomics, and medical field that have sensitive private data. While several studies have been reported on the prediction phase, few studies have been conducted on the *training* phase due to the inefficiency of homomorphic encryption (HE), leaving the machine learning training on encrypted data only as a long-term goal.

In this paper, we propose an efficient algorithm for logistic regression on encrypted data, and evaluate our algorithm on real financial data consisting of 422,108 samples over 200 features. Our experiment shows that an encrypted model with a sufficient Kolmogorov Smirnow statistic value can be obtained in ∼17 hours in a single machine. We also evaluate our algorithm on the public MNIST dataset, and it takes ∼2 hours to learn an encrypted model with 96.4% accuracy. Considering the inefficiency of HEs, our result is encouraging and demonstrates the practical feasibility of the logistic regression training on large encrypted data, for the first time to the best of our knowledge.

1 Introduction

Suppose multiple financial institutions want to predict credit scores of their customers. Although each institution could independently learn a prediction model using various machine learning techniques, they may be able to collectively learn a better model by considering all of their data together for training. However, it is risky in terms of data security to share financial data between institutions, being even illegal in many countries.

Homomorphic encryption (HE), an encryption scheme that allows arbitrary computations on encrypted data,[3] can be used to solve this dilemma. Using HE, multiple institutions can share their data in an encrypted form and run machine learning algorithms on the encrypted data without ever decrypting. This HE-based approach is flexible in that the training computation can be delegated to any party (or even an *untrusted* third party) without revealing the training data (other than their own). This flexibility is desirable, as other approaches require additional assumptions and conditions that may not be realizable in practice (see Section 1.4 for more details).

Despite many advantages, however, HE has not been used for computation-intensive tasks such as machine learning (especially on the training phase), having been thought to be impractical due to its large computation overhead. Indeed, basic operations (e.g., addition or multiplication) on ciphertexts are several (i.e., three to seven) orders of magnitude slower than the corresponding operations on plaintexts even in the state-of-the-art [GHS12, HS15, CKKS17, vDGHV10, BV14, BV11, Bra12, BGV12, LATV12, CLT14, DM15]. Moreover, some complex operations may cause additional overhead when they are reduced to a combination of basic operations.[4] For example, fractional number (e.g., fixed-point or floating-point) arithmetic operations on ciphertexts are quite expensive, as they involve bit-manipulation operations that are expressed as complex arithmetic circuits of a large depth.

In addition to the sheer amount of computation, the use of various complex operations, such as floating-point arithmetic and non-polynomial functions (e.g., sigmoid), makes it challenging to apply HE to machine learning algorithms. Indeed, HEs have been applied to machine learning algorithms only in non-realistic settings [GLN12, KSK$^+$18] where only small-size training datasets over a small number of features are considered; or, they have been applied only on the prediction phase [BLN14, BPTG15, GBDL$^+$16, LLH$^+$17, CdWM$^+$17, JVC18, BMMP17] where the amount of computation is much smaller than that of the training phase.

1.1 Our Results and Techniques

In this paper, we present an efficient algorithm for logistic regression on encrypted data, and demonstrate its practical feasibility against realistic size datasets, for the first time to the best of our knowledge. We evaluate our algorithm against a real, private financial dataset consisting of 422,108 samples over 200 features. Our implementation successfully learned a quality model in ∼17 hours on a single machine, where we tested it against a validation set of 844,217 samples and

[3] Precisely, it is a *fully* homomorphic encryption (FHE) that supports the *unlimited* number of operations on ciphertexts. However, throughout the paper, we will refer to it as simply HE as long as the precise meaning is clear in the context.

[4] Most of HE schemes support only basic operations like addition and multiplication as built-in, and require other operations to be represented in the form of a combination of the built-in operations.

obtained a sufficient Kolmogorov Smirnov statistic value of 50.84. The performance is “only” two to three orders of magnitude slower than that of plaintext learning, which is encouraging, considering the inherent computational overhead of HEs. We also executed our algorithm on the public MNIST dataset for more detailed evaluation, and it took ∼2 hours to learn an encrypted model with 96.4% accuracy. Below we describe the principal techniques used in our efficient logistic regression algorithm on a large encrypted dataset.

Approximate HE Our algorithm leverages the recent advances of (word-wise) *approximate* HE schemes and the *approximate* bootstrapping method to reduce the computational overhead. The approximate HE can quickly compute approximated results of complex operations, avoiding the bit-manipulation overhead. (Refer to Section 1.4 for comparison with other HE schemes.) Similarly, the approximate bootstrapping can efficiently bootstrap a ciphertext at the cost of additional approximation noise.

While both the approximate HE and the approximate bootstrapping can reduce the computational overheads, they have the disadvantage of introducing an additional noise for each computation step. Even if it is small, the noise may affect the overall machine learning performance (e.g., the convergence rate and accuracy), but it had not been clear how critical the small noise is. We empirically show that the additional noise is not significant to deteriorate the accuracy of a learned model and the convergence rate. Indeed, our finding is consistent with the results of low-precision training approaches in the literature [DSFRO17, ZLK^{+}16, GAGN15, CBD14] which have also empirically shown that small approximation (round-off) errors due to the low-precision are manageable.

HE-Optimized, Vectorized Logistic Regression Algorithm The approximate HE scheme we use also supports the packing method [CKKS17] which can further reduce the computation overhead. In the packed HEs, a single ciphertext represents an encryption of a vector of plaintexts, and ciphertext operations correspond to point-wise operations on plaintext vectors, so-called single instruction multiple data (SIMD) operations.

To maximize the benefits of the packed scheme, we vectorize our logistic regression algorithm to utilize the SIMD operations as much as possible. For example, the inner product operation is represented as a SIMD-multiplication followed by a sequence of rotations and SIMD-additions (Section 4.2). Moreover, we carefully tune the vectorized algorithm to minimize redundant computations caused by the use of the SIMD operations, reduce the depth of nested multiplications, and minimize the approximation noises by reordering operations (Section 4.3).

Parallelized Bootstrapping One of the most expensive operations of HEs is the bootstrapping operation (even with the approximate bootstrapping method). This operation needs to be periodically executed during the entire computation. In logistic regression, for example, it should be executed every few iterations, and dominates the overall training time. It is critical for performance to optimize the bootstrapping operation.

We design our algorithm to parallelize the bootstrapping operation. It splits a ciphertext into multiple smaller chunks and executes bootstrapping on each chunk in parallel, achieving a significant speedup of the overall performance. Moreover, we carefully design the packing of training data (see below) so that our algorithm continues to use the chunks without merging them in the next training iterations, which additionally saves time it takes to reconstruct a ciphertext from the chucks.

HE-Optimized, Efficient Partition of Training Data As mentioned above, we pack multiple plaintexts in a single ciphertext, and it is critical for performance how to pack (i.e., partition) the training dataset. The training data can be seen as an $n \times m$ matrix with n samples and m features. A naive encoding would pack each row (or column) into a ciphertext, resulting in a total of n (or m) ciphertexts. This encoding, however, is not efficient, since it either does not utilize the maximum capacity of the ciphertexts, or requires too much capacity, increasing the computation overhead drastically.

We design an efficient partition of training data in which a sub $n' \times m'$ matrix is packed into a single ciphertext, where the size of the matrix is set to the maximum capacity of each ciphertext, and m' is set to align with the aforementioned parallelization technique, avoiding an extra overhead of the ciphertext reconstruction (Section 4.2).

Approximating Non-Polynomial Functions As mentioned earlier, non-polynomial functions are computationally expensive in HEs. We mitigate this performance overhead issue by approximating them as polynomials. A sigmoid function, for example, is replaced by its polynomial approximation in our training algorithm. Note that, however, an approximation at a point such as Taylor expansion is not adequate for logistic regression (and machine learning in general) since the deviation could be too large at other points. Instead, we use an interval approximation whose difference on the interval is minimized in terms of least squares. Combined with a proper input normalization, the interval approximation has provided sufficient precision for logistic regression in our experiment.

1.2 Contributions

Our specific contributions and novelty in our algorithm, among the techniques described in the previous section, are as follows.

- We adopt, for the first time, the combination of *the approximate HE* and *the approximate bootstrapping* for the machine learning (training) on encrypted data. We demonstrate its practical feasibility by evaluating it on realistic size datasets, and empirically show that the approximation noise is not significant to deteriorate the overall learning performance. Note that adopting only the approximate HE *without* the approximate bootstrapping (as in [KSW+18, KSK$^+$18]) is not sufficient to achieve the level of scalability reported in this paper. The bootstrapping operation is essential for the scalability and its optimization is critical for the overall performance. Refer to Section 1.4 for more detailed comparison to related work.

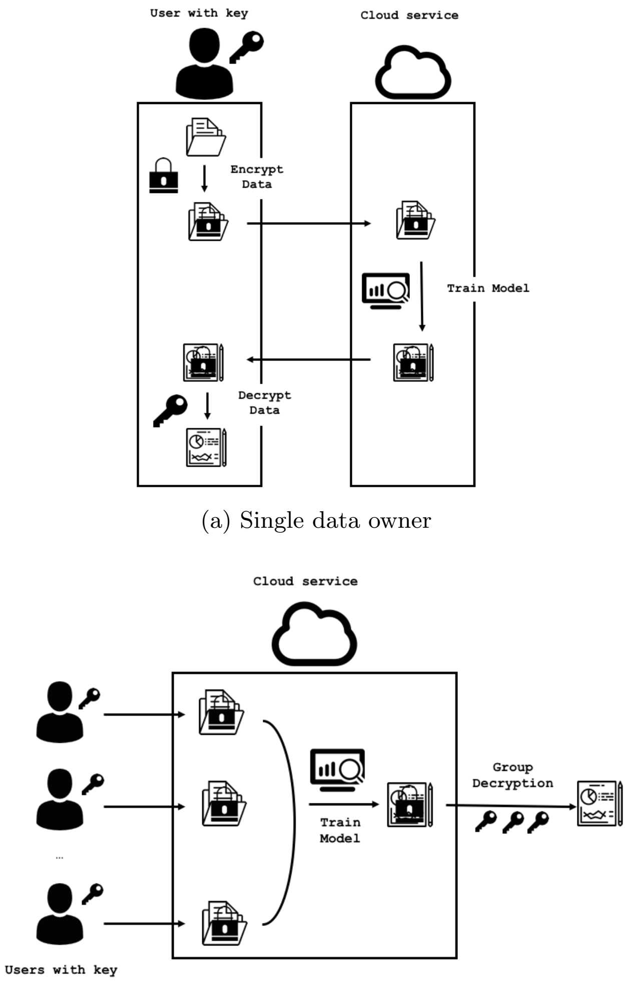
– We present novel optimization techniques, especially ones for the bootstrapping operation. We parallelize a bootstrapping operation by splitting a ciphertext, while we carefully design the partition of a training dataset to avoid reconstructing the split ciphertexts, which significantly reduces the parallelization overhead. We also fine-tune the evaluation order to minimize the accumulated approximation noises due to the approximate HE scheme. We admit, however, that the SIMD vectorization and the polynomial approximation are not novel.

1.3 Usage Model of Our Approach

There are several usage scenarios of our approach as illustrated in Figure 1. One typical scenario is in a private cloud machine learning (Figure 1-a), where a user uploads his private data after encrypting with HE, and later downloads an encrypted model from the cloud that has performed the machine learning training on the encrypted data. The private key of the HE scheme is owned only by the user, and the cloud possesses only the public parameters and evaluation keys. In this case, we can use a symmetric HE scheme that is more efficient than an asymmetric HE scheme.

Our approach can also be used in the multiple data owner setting (Figure 1-b). In that case, each data owner shares the public key of the HE scheme and uploads his data to the cloud after encrypting with HE. The cloud performs the same task as before and outputs an encrypted model. This model can be decrypted by the decryptor who owns the private key. Here the decryptor can be either a single entity or a group of entities that have their own share of the private key. In the latter case, an additional key sharing protocol is required for the public key to be jointly generated by the entities having a random share of the private key. We may use the threshold HE schemes [JRS17, CDN01] for that.

Note that, in the protocol, no information is revealed to each other except the learned model unless the underlying HE scheme is broken or its secret key is disclosed. This is the case even if the cloud is compromised. Hence it could be an ultimate solution for analyzing private or sensitive data while keeping privacy. In the financial area, for example, it can be used to construct a prediction model for the credit score from private financial data. In this case, each bank sends its encrypted data to a cloud, and the cloud performs a machine learning algorithm on the collected encrypted data, and sends back the encrypted prediction model to the contributed banks. The banks can collaboratively decrypt it, and share the model using a threshold or group decryption [JRS17, CDN01]. Another example comes from genomic and medical data. Genomic and medical data are very useful for analyzing and predicting diseases [KSK$^+$18, JWB$^+$17]. Due to its private and sensitive nature, however, the use of data is extremely limited



(a) Single data owner

(b) Multiple data owners

Fig. 1: Secure machine learning on the cloud via homomorphic encryption

especially for sharing data among hospitals. Our model can solve this problem by collecting genomic or medical data from hospitals after encryption and performing a machine learning on these data. The learned model can be decrypted by a trusted third party such as a government agency, and distributed to the participated hospitals. Refer to [ACC$^+$17] for more applications in other areas such as education and national security.

1.4 Related Work

There have been several studies on performing a machine learning without revealing private information. Here we consider two major types of approaches: HE-based approaches and the multi-party computation (MPC)-based approaches.

HE-Based Approaches Graepel *et al.* [GLN12] presented a homomorphic evaluation algorithm of two binary classifiers (i.e., linear means and Fisher's linear discriminant classifiers), and Kim *et al.* [KSW$^+$18, KSK$^+$18] proposed a homomorphic evaluation of logistic regression. However, they provided only a proof-of-concept evaluation, where small-scale training datasets (consisting of only dozens of samples and features) are considered. Moreover, it is not clear how scalable their approaches are as the size of datasets and the number of iterations increase. Indeed, their implementations require the multiplication depth (i.e., the number of iterations) to be bounded, meaning that their implementations are not scalable. Our algorithm, however, is scalable in the sense that it can admit an *arbitrary* number of iterations, and the time complexity is *linear* in terms of the number of iterations.

There also have been reported studies on homomorphic evaluation of the prediction phase of machine learning algorithms including neural networks [BLN14, BPTG15, GBDL$^+$16, LLH$^+$17, CdWM$^+$17, JVC18, BMMP17]. However, the prediction phase is much simpler than the training phase in terms of the amount of computation (especially in terms of the multiplication depth), and thus their techniques are hard to be applied to the training phase directly.

On the other hand, Aono *et al.* [AHPW16] presented a protocol for secure logistic regression using the additively homomorphic encryption. It approximates the cost function by a low-degree polynomial, and encrypts the training data in the form of the monomials of the polynomial. Then, the approximated cost function can be homomorphically evaluated by simply adding the encrypted monomials. This protocol, however, has the disadvantage that the number and/or the size of ciphertexts increase exponentially as the degree of the polynomial approximation increases.

MPC-Based Approaches Nikolaenko *et al.* [NWI$^+$13] proposed an MPC-based protocol for training linear regression model, which combines a linear homomorphic encryption and the Yao's garbled circuit construction [Yao86]. Mohassel and Zhang [MZ17] improved the protocol by using secure arithmetic operations on shared decimal numbers, and applied it to logistic regression and neural network training.

The MPC-based approaches, however, incur large communication overhead, where the communication overhead increases drastically as the number of participants increases. Moreover, they require all of the participants to be online during the entire training process, which adds another limitation in practice. To mitigate this problem, an approach using two delegating servers was proposed [MZ17], where multiple parties upload their data to two servers and delegate the training task to the servers using the two-party computation (2PC).

This approach, however, requires an additional assumption that two servers do not collude. Recall that our HE-based approach requires *no* assumption on the server, and can admit even a compromised server.

Other HE Schemes The bit-wise HE schemes [DM15, CGGI17] provide an efficient bootstrapping operation, and can admit a boolean circuit directly as a complex operation that involves bit-manipulation. However, their operations are inherently slow due to their large circuit depth. On the other hand, the word-wise HE schemes [BGV12, FV12, Bra12, CKKS17] provide more efficient operations since their circuit depth can be significantly reduced. However, they suffer from an expensive bootstrapping operation due to the large size of ciphertexts. Also, they do not provide the same level of efficiency for complex operations that involve bit-manipulation, as their circuit depth is still large in the form of an arithmetic circuit over words. The word-wise approximate HE scheme, adopted in our algorithm, can improve the efficiency of bit-manipulating complex operations at the cost of approximation noises. It is useful in applications where the small approximation noises in the intermediate computation steps are not critical for the final computation result, which is indeed the case for most of the machine learning algorithms.

2 Preliminaries

This section provides a brief background of logistic regression and homomorphic encryption.

2.1 Logistic Regression and Gradient Descent Method

Logistic regression is a machine learning algorithm to learn a model for classification. We focus on the binary classification throughout this paper for the simplicity of the presentation. In logistic regression, we consider the following model:

$$\log\left[\frac{\Pr(Y=0|X=\boldsymbol{x})}{\Pr(Y=1|X=\boldsymbol{x})}\right] = \langle \boldsymbol{w}, (1,\boldsymbol{x})\rangle$$

where:[5]

$$\Pr(Y=1|X=\boldsymbol{x}) = \frac{1}{1+e^{-\langle \boldsymbol{w},(1,\boldsymbol{x})\rangle}}$$

$$\Pr(Y=0|X=\boldsymbol{x}) = \frac{e^{-\langle \boldsymbol{w},(1,\boldsymbol{x})\rangle}}{1+e^{-\langle \boldsymbol{w},(1,\boldsymbol{x})\rangle}}$$

for an input vector X of d features, a class Y, a weight vector $\boldsymbol{w} \in \mathbb{R}^{d+1}$.

The goal of the logistic regression training, given m samples $\{(\boldsymbol{x}_i, y_i)\}_m$, is to find a weight vector $\boldsymbol{w}$ that minimizes the negative log likelihood function

[5] We write $\langle \cdot, \cdot \rangle$ for the inner product, and $(1, \boldsymbol{x})$ for a vector extended with 1 from $\boldsymbol{x}$.

$\ell(\boldsymbol{w}) = -\frac{1}{m} \cdot \log L(\boldsymbol{w})$, where:

$$L(\boldsymbol{w}) = \prod_{i=1}^{m} h_{\boldsymbol{w}}(\boldsymbol{x}_i)^{y_i} \cdot (1 - h_{\boldsymbol{w}}(\boldsymbol{x}_i))^{1-y_i}$$

with $h_{\boldsymbol{w}}(\boldsymbol{x}_i) = \sigma(\langle \boldsymbol{w}, (1, \boldsymbol{x}_i)\rangle)$ and $\sigma(x) = 1/(1 + e^{-x})$. Since $\ell(\boldsymbol{w})$ is convex, we can use the gradient descent method to find the vector $\boldsymbol{w}$ that minimizes $\ell(\boldsymbol{w})$. The gradient descent method for logistic regression is formulated as the following recurrence relation:

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \alpha \cdot \Delta_w \ell(\boldsymbol{w}_i)$$

for a learning rate α. The gradient of the log likelihood function is as follows:

$$\Delta_w \ell(\boldsymbol{w}) = -\frac{1}{m} \sum_{i=1}^{m} \sigma(-\langle \boldsymbol{z}_i, \boldsymbol{w}\rangle) \cdot \boldsymbol{z}_i$$

where $\boldsymbol{z}_i = y'_i \cdot (1, \boldsymbol{x}_i) \in \mathbb{R}^{d+1}$, and $y'_i = 2y_i - 1 \in \{-1, 1\}$.

2.2 Kolmogorov Smirnov Statistic

Kolmogorov Smirnov (KS) statistic is a statistic value that tells whether a given model can distinguish between occurrence and nonoccurrence of a certain event. Mathematically, the KS value is computed by the maximum difference between the cumulative percentage of occurrence and nonoccurrence of the event. The higher the KS value is, the better the model distinguishes the event. In general, the model is considered to distinguish the event well when its KS value is greater than 40. In logistic regression, the model is the weight vector $\boldsymbol{w}$, and the event is to determine $y = 1$. The KS statistic is a common measure to test the quality of the model in the financial area.

2.3 Fully Homomorphic Encryption

The concept of homomorphic encryption (HE) that allows computation on encrypted data was proposed by Rivest *et al.* [RAD78]. There have been several proposals that support a single operation. For example, ElGamal Encryption scheme allows the multiplication of ciphertexts, and Okamoto-Uchiyama scheme [OU98] and Pailler encryption scheme [Pai99] support the addition of ciphertexts without decryption. However, a homomorphic encryption scheme that supports both addition and multiplication had been a longstanding open problem. Supporting the two operations is important because an arbitrary computation function can be composed of addition and multiplication on $\mathbb{Z}_2$.

The first secure HE (based on the hardness assumption of a plausible number-theoretic problem) was proposed by Gentry [Gen09]. He first constructed a Somewhat Homomorphic Encryption (SHE) scheme that supports all kinds of operations without decryption but by a limited number. In his SHE scheme, after some number of operations, the plaintext may not be recovered from the corresponding ciphertext mainly because the noise in the encryption grows too large to alter the plaintext in the ciphertext. He also suggested a so-called *bootstrapping* procedure that converts a ciphertext with large noise into another ciphertext with the same plaintext but small noise to allow more operations. By this procedure, he proved that his SHE can be used to evaluate an arbitrary function on encrypted values with the bootstrapping technique without ever decrypting. Such schemes are called Fully Homomorphic Encryption (FHE) schemes.

Since his work, various kinds of FHE schemes have been suggested. Most of them are based on one of the following three hardness problems: Learning with Errors (LWE) problem [Reg09], Ring-LWE problem [LPR10] and Approximate GCD problem [vDGHV10]. Their message space is either an element of $\mathbb{Z}_p$ or a vector over $\mathbb{Z}_p$ for a positive integer p. When $p = 2$, it is advantageous in bit operations and bootstrapping, but rather slow for integer arithmetic. For example, one addition of two k-bit integers requires the evaluation of depth $O(k)$ circuit. On the other hand, if p is large, an integer arithmetic can be efficient until the result is smaller than p. However, the operation can be complicated and becomes inefficient after this point.

Recently, an approximate homomorphic encryption scheme is proposed by Cheon *et al.* [CKKS17] to solve this problem. The scheme, called HEAAN, supports efficient encrypted floating point operations by providing a rounding of plaintext (called *rescaling*) to discard insignificant figures from the plaintext as well as addition and multiplication. For example, when $p = \Omega(\log d)$ it can evaluate a d-power of a real number approximately with $\log d$ multiplications/rescalings of ciphertexts.

2.4 Polynomial Approximation

The classic ways of finding a polynomial approximation are Taylor expansion and Lagrange interpolation. These methods provide a precise approximation on a small range close to the point of interest, but their approximation error could drastically increase outside the small range.

The least squares fitting polynomial, on the other hand, provides a good approximation on a large range.

Definition 1 (Least Squares Fitting Polynomial). *For the given N points (y_i, x_i) and the degree d, the least squares fitting polynomial $p(x)$ is a polynomial of degree d that has the smallest square sum error, $\sum(y_i - p(x_i))^2$, among polynomials of degree $\leq d$.*

A method to construct the least squares fitting polynomial is as follows. Let $p(x) = a_0 + a_1 x + \cdots + a_d x^d$ and compute the partial derivatives of the square sum. Then the problem is reduced to solving the following system of equations:

$$\begin{bmatrix} N & \sum_{i=1}^N x_i & \cdots & \sum_{i=1}^N x_i^d \\ \sum_{i=1}^N x_i & \sum_{i=1}^N x_i^2 & \cdots & \sum_{i=1}^N x_i^{d+1} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^N x_i^d & \sum_{i=1}^N x_i^{d+1} & \cdots & \sum_{i=1}^N x_i^{2d} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_d \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^N y_i \\ \sum_{i=1}^N x_i y_i \\ \vdots \\ \sum_{i=1}^N x_i^d y_i \end{bmatrix}$$

There are various (numerical) methods that can be used to solve the above matrix equation.

3 Approximate Homomorphic Encryption

We briefly describe the approximate HE scheme, HEAAN, that we use in our homomorphic logistic regression algorithm.

3.1 Ring Learning with Errors Problem

Let us fix an integer N and a prime q. Let $\mathcal{R} = \mathbb{Z}[x]/(x^N + 1)$ and $\mathcal{R}_q = \mathcal{R}/q\mathcal{R}$. Let χ be an error distribution over $\mathcal{R}_q$. For a given polynomial $s \in \mathcal{R}_q$, we define the Ring Learning With Error (RLWE) distribution $\mathsf{RLWE}_{q,\chi,\boldsymbol{s}}$ by:

$$\mathsf{RLWE}_{q,\chi,\boldsymbol{s}} = \{(b, a) \in \mathcal{R}_q^2 : a \leftarrow \mathcal{U}(\mathcal{R}_q), e \leftarrow \chi, b = a \cdot s + e.\}$$

Here $\mathcal{U}(\mathcal{R}_q)$ is the uniform distribution over $\mathcal{R}_q$. The RLWE problem is to distinguish the uniform distribution in $\mathcal{R}_q^2$ with the RLWE distribution. This problem is widely used in various public key cryptographic schemes including homomorphic encryption. The security of the homomorphic encryption scheme we use also depends on the hardness of RLWE problem.

3.2 HEAAN Scheme

First, let us define some distributions used in the scheme. For a real $\sigma > 0$, $\mathsf{DG}(\sigma^2)$ is the distribution in $\mathbb{Z}^N$ that each component comes from the discrete Gaussian distribution of variance σ^2. For a positive integer h, $\mathsf{HWT}(h)$ is the set of vectors in $\{-1, 0, 1\}^N$ whose hamming weight is h. Finally, for a real $0 \leq \rho \leq 1$, $\mathsf{ZO}(\rho)$ is the distribution in $\{-1, 0, 1\}^N$ that each component is either 1 or -1 with probability $\rho/2$, respectively, and 0 with probability $1 - \rho$. Let $\mathcal{R} = \mathbb{Z}[x]/(x^N + 1)$ and $\mathcal{R}_Q = \mathbb{Z}_Q[x]/(x^N + 1)$ where Q and N are powers of two. A ciphertext is in the polynomial ring $\mathcal{R}_Q^2$ and a secret key is randomly selected from $\mathsf{HWT}(h)$. Note that Q and N are determined by both the depth of a target circuit and a security parameter λ.

HEAAN is a *leveled* HE scheme. A ciphertext is associated with a modulus, and the ciphertext modulus decreases for each homomorphic operation. Once the modulus goes below a lower bound, no more operation can be conducted on the associated ciphertext until it is bootstrapped (see Section 3.3 for the bootstrapping operation). HEAAN keeps track of this ciphertext modulus in a form of *level.* A ciphertext level ℓ denotes a ciphertext modulus of 2^ℓ. The maximum level L is $\log_2 Q$. If the same scaling factor $\Delta = 2^{\mathsf{pBits}}$ is used in the entire homomorphic computation, for example, $\log_2 Q$ should be set to be at least $d \cdot \mathsf{pBits}$, where d is the depth of the computation circuit of interest and pBits is the number of bits for precision. We also denote 2^ℓ by Q_ℓ.

KeyGen(λ, L) The key generation procedure is as follows:

- *Parameters*: Given the maximum level L, let $Q = 2^L$ and $P = 2^L$. Given the security parameter λ, we choose a power of two integer M, an integer h, and a real number $\sigma > 0$ for an RLWE problem that has λ-bit of security level.
- *Secret key*: Sample $s(x) \leftarrow \mathsf{HWT}(h)$ and let $\mathsf{sk} = (1, s(x))$ be the secret key.
- *Public key*: Sample $a(x) \leftarrow \mathcal{R}_Q$, and $e(x) \leftarrow \mathsf{DG}(\sigma^2)$. Let $\mathsf{pk} = (b(x), a(x)) \in \mathcal{R}_Q^2$ be the public key, where $b(x) = -a(x)s(x) + e(x) \pmod{Q}$.
- *Evaluation key*: Sample $a'(x) \leftarrow \mathcal{R}_{PQ}$ and $e'(x) \leftarrow \mathsf{DG}(\sigma^2)$. Let $\mathsf{evk} = (b'(x), a'(x)) \in \mathcal{R}_{PQ}^2$ be the (public) evaluation key, where $b'(x) = -a'(x)s(x) + P \cdot s^2(x) + e'(x) \pmod{PQ}$.

Encryption and Decryption To pack a vector of complex (or real) numbers of plaintexts, we convert such a vector into an element in the polynomial ring. For the simplicity of presentation, we describe the inverse of the conversion. By evaluating the non-conjugate roots to a polynomial with real coefficients, we can convert the polynomial into a vector of complex numbers. This mapping τ is an isomorphism and can be described as follows:

$$\begin{aligned} \tau : \mathbb{R}[x]/(x^N + 1) &\rightarrow \mathbb{C}^{N/2} \\ f(x) &\mapsto (f(\zeta_{2N}^{5^i}))_{0 \leq i < N/2} \end{aligned}$$

Here $\zeta_{2N} = \exp(2\pi i/2N) \in \mathbb{C}$ is a primitive $2N^{\text{th}}$ root of unity in complex field. This packing scheme enables the batch encryption that can achieve a better amortized performance using the single instruction multiple data (SIMD) operations. The encoding and decoding procedures are as follows.

- $\texttt{encode}(\boldsymbol{m} \in \mathbb{C}^{N/2}, \texttt{pBits})$: Compute $\tau^{-1}(\boldsymbol{m}) = m(x) \in \mathbb{R}[X]/(X^N + 1)$. Return an integer polynomial $m'(x) = \lfloor \Delta \cdot m(x) \rceil \in \mathcal{R}$ for $\Delta = 2^{\texttt{pBits}}$.
- $\texttt{decode}(m(x) \in \mathcal{R}, \texttt{pBits})$: Compute $m'(x) = m(x)/\Delta \in \mathbb{R}[X]/(X^N + 1)$ for $\Delta = 2^{\texttt{pBits}}$. Return a vector of complex numbers $\boldsymbol{m} = \tau(m'(x)) \in \mathbb{C}^{N/2}$.

The encryption and decryption procedures over the encoded plaintexts are as follows.

- $\texttt{encrypt}(\boldsymbol{m} \in \mathbb{C}^{N/2}, \texttt{pBits})$: Compute an integer polynomial $m(x) = \texttt{encode}(\boldsymbol{m}, \texttt{pBits})$. Sample $v(x) \leftarrow \mathsf{ZO}(0.5)$ and $e_0, e_1 \leftarrow \mathsf{DG}(\sigma^2)$. Return $(b(x), a(x)) = v(x) \cdot \mathsf{pk} + (m(x) + e_0(x), e_1(x)) \bmod Q$.
- $\texttt{decrypt}(c, \texttt{pBits})$: For the given level l ciphertext $c = (b(x), a(x))$, compute $m(x) = b(x) + a(x) \cdot s(x) \bmod Q_\ell$ for $Q_\ell = 2^\ell$. Return a vector $\boldsymbol{m} = \texttt{decode}(m(x), \texttt{pBits})$.

Homomorphic Operations Now we describe homomorphic operations on ciphertexts. First, we have addition and multiplication as follows.

- $\texttt{add}(c_1,\ c_2)$: Return $c_3 = (b_1(x) + b_2(x), a_1(x) + a_2(x)) \in \mathcal{R}^2_{Q_\ell}$ for the level ℓ ciphertexts $c_1 = (b_1(x), a_1(x))$ and $c_2 = (b_2(x), a_2(x))$. Note that $c3$ is an encryption of $\boldsymbol{m}_1 + \boldsymbol{m}_2$ when c_1 and c_2 are encryptions of $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$, respectively.
- $\texttt{mult}(c_1,\ c_2)$: Compute $(d_0, d_1, d_2) = (b_1(x) + b_2(x), a_1(x)b_2(x) + a_2(x)b_1(x), a_1(x)a_2(x)) \in \mathcal{R}^3_{Q_\ell}$ for the level ℓ ciphertexts $c_1 = (b_1(x),\ a_1(x))$ and $c_2 = (b_2(x), a_2(x))$. Return $c_3 = (d_0, d_1) + \lfloor P^{-1} \cdot d_2 \cdot \mathsf{evk} \rceil \in \mathcal{R}^2_{Q_\ell}$. Note that c_3 is an encryption of $\boldsymbol{m}_1 \circ \boldsymbol{m}_2$ with a scaling factor $\varDelta_1 \cdot \varDelta_2$ when c_1 and c_2 are encryptions of $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ with scaling factors $\varDelta_1$ and $\varDelta_2$, respectively. Here $\circ$ denotes the element-wise multiplication.

We also have addition and multiplication with a constant.

- $\texttt{cAdd}(c,\ \texttt{cnst},\ \texttt{pBits})$: Return $c' = (b(x) + \texttt{cnst} \cdot 2^{\texttt{pBits}}, a(x)) \mod Q_\ell$ for the level ℓ ciphertext $c = (b(x), a(x))$. Note that c' is an encryption of the vector $(m_1 + \texttt{cnst}, \cdots, m_{N/2} + \texttt{cnst})$ when c is the encryption of $\boldsymbol{m} = (m_1, \cdots, m_{N/2})$ with scaling factor $2^{\texttt{pBits}}$.
- $\texttt{cMult}(c,\ \texttt{cnst},\ \texttt{pBits})$: Return $c' = (\texttt{cnst} \cdot 2^{\texttt{pBits}} \cdot b(x), \texttt{cnst} \cdot 2^{\texttt{pBits}} \cdot a(x)) \mod Q_\ell$ for the level ℓ ciphertext $c = (b(x), a(x))$. Note that c' is an encryption of $\texttt{cnst} \cdot \boldsymbol{m}$ when c is the encryption of $\boldsymbol{m}$ with scaling factor $2^{\texttt{pBits}}$.
- $\texttt{cMultByVec}(c,\ \boldsymbol{m}',\ \texttt{pBits})$: Encode $\boldsymbol{m}'$ by $m'(x) = \text{encode}(\boldsymbol{m}', \texttt{pBits})$ and return $c' = (m'(x) \cdot b(x), m'(x) \cdot a(x)) \mod Q_\ell$ for the level ℓ ciphertext $c = (b(x), a(x))$. Note that c' is an encryption of $\boldsymbol{m} \circ \boldsymbol{m}'$ when c is the encryption of $\boldsymbol{m}$ with scaling factor $2^{\texttt{pBits}}$.

The rescaling operation, an important operation of HEAAN, is given below. The `rescale` operation is necessary to control the scaling factor of a ciphertext. The scaling factor increases for each homomorphic multiplication, growing exponentially as the number of multiplications increases. A ciphertext becomes no longer valid once the associated scaling factor becomes too large. The `modDown` operation has a role in matching the level of two ciphertexts, by reducing the level of one with a higher level.

- $\texttt{rescale}(c,\ \texttt{bits})$: Return $c' = (\lfloor b(x) \cdot 2^{-\texttt{bits}} \rceil, \lfloor a(x) \cdot 2^{-\texttt{bits}} \rceil) \in \mathcal{R}_{Q_{\ell-\texttt{bits}}}$ for the given level ℓ ciphertext $c = (b(x), a(x))$. Note that c' is an encryption of $\boldsymbol{m}$ with a new scaling factor $\varDelta/2^{\texttt{bits}}$ when c is an encryption of $\boldsymbol{m}$ with a scaling factor $\varDelta$.
- $\texttt{modDownBy}(c,\ \texttt{bits})$: Return $c' = c$ with level $\ell - \texttt{bits}$ for the given level ℓ ciphertext c. Note that c' is an encryption of the same message of c and the level of c' only differs from c.
- $\texttt{modDownTo}(c,\ c')$: Return $c'' = c$ with level ℓ' for the given level ℓ ciphertext c and level ℓ' ciphertext c'. Note that c'' is an encryption of the same message of c and the level of c'.

The mapping τ used in the packed encoding scheme has a nice property that comes from the Galois mapping. Let $g(f(x)) = f(x^5) \in \mathbb{R}[x]/(x^N + 1)$. Then, $\tau(g(f(x))) = (f(\zeta_{2N}^{5^{i+1}}))_{0 \le i < N/2}$. Therefore, applying g to a polynomial ring element corresponds to shifting of the complex vector in $\mathbb{C}^{N/2}$. This property is used in homomorphic left- and right-rotation operations.

- `RotKeyGen`(sk, i): Sample $a'' \leftarrow \mathcal{R}_{PQ}$ and $e'' \leftarrow \mathsf{DG}(\sigma^2)$ Let $\mathsf{rotk} = (b''(x), a''(x)) \in \mathcal{R}^2_{PQ}$ be a public key for rotation, where $b''(x) = -a''(x)s(x) + e'(x) + P \cdot s(x^{5^i}) \mod PQ$.
- `leftRotate`$(c,\ i)$: Compute $(d_0, d_1) = (b(x^{5^i}), a(x^{5^i})) \in \mathcal{R}^2_{Q_\ell}$ for the given level ℓ ciphertext $c = (b(x), a(x))$. Return $c' = (d_0, 0) + \lfloor P^{-1} \cdot d_1 \cdot \mathsf{rotk}_i \rceil \in \mathcal{R}^2_{Q_\ell}$ Note that c' is an encryption of $\boldsymbol{m'}$ such that $m'_j = m_{[j+i]_{N/2}}$ when c is an encryption of $\boldsymbol{m}$.

The right-rotation operation can be simply obtained using the left-rotation, that is, $\mathsf{rightRorate}(c, i) = \mathsf{leftRotate}(c, -i)$. Refer [CKKS17] for more details.

3.3 Bootstrapping for HEAAN

Recall that the homomorphic multiplication affects both the scaling factor and the modulus of a ciphertext. While the scaling factor can be adjusted by the rescaling operation, the modulus can be reset only by a so-called bootstrapping operation that we will describe in this section. The bootstrapping operation is necessary for a computation circuit of large depth. Without the bootstrapping operation, the number of possible nested multiplications is limited, and thus it is infeasible to admit a circuit of large depth. Recently, a bootstrapping operation for HEAAN has been proposed [CHK$^+$18,Kim18]. Their scheme does not require a bootstrapping key, but only a number of rotation keys that are used in various linear transformations and their inverse transformations.

- `bootstrap`(c): Return a new ciphertext c' with a larger ciphertext modulus. The size of c' is less than the maximum ciphertext modulus. Note that c' does not have the maximum ciphertext modulus. The homomorphic evaluation of the decryption computation will reduce the ciphertext modulus.

To use the bootstrap() function, we need to set the maximum ciphertext modulus to be large enough to evaluate the decryption computation circuit. This bootstrapping process decreases the ciphertext modulus, so we had to consider it for setting parameters in our experiment.

4 Logistic Regression on Encrypted Data

In this section, we explain our algorithm for efficient logistic regression on encrypted data. We first present a baseline (plaintext) algorithm of the logistic regression training, designed to be friendly to homomorphic evaluation (Section 4.1). Then we explain how to optimize the baseline algorithm to be efficiently evaluated in HEs (Sections 4.2 and 4.3).

4.1 HE-friendly Logistic Regression Algorithm

We first explain our baseline algorithm of the logistic regression training, as shown in Algorithm 1, that we will further optimize in the next section. We design the baseline algorithm to be friendly to homomorphic evaluation by avoiding the use of certain types of computations that are expensive in HEs.

Mini-Batch Gradient Descent We adopt the mini-batch gradient descent method, where we set the mini-batch size according to the number of slots in a packed ciphertext. We do not consider the stochastic gradient descent method since it does not utilize the maximum capacity of the packed ciphertext. Also, we do not consider the full-batch gradient descent method since it requires too many and/or large ciphertexts for each iteration when the training dataset is large.

Nesterov Accelerated Gradient Optimizer We adopt Nesterov accelerated gradient (NAG) as the gradient descent optimization method. We choose NAG among the various optimization methods, since it provides decent optimization performance without using the division operation that is expensive in HEs. The NAG can be formulated as follows:

$$\begin{aligned} w_{i+1} &= v_i - \gamma \cdot \varDelta_w \ell(v_i) \\ v_{i+1} &= (1-\eta) \cdot w_{i+1} + \eta \cdot w_i \end{aligned}$$

where w_i and v_i are two weight vectors to be updated for each iteration i, $\varDelta_w \ell(v_i)$ is the gradient of the log likelihood function (as given in Section 2.1), and γ and η are parameters.

Polynomial Approximation of Activation Function An essential step of the logistic regression training is to apply an activation function, e.g., the sigmoid function $\sigma(x) = 1/(1+e^{-x})$. Since non-polynomials are very expensive to evaluate in HEs, we consider its (low-degree) polynomial approximation σ' as an alternative in our algorithm. We use the least squares fitting method to approximate the sigmoid function. The least squares fitting polynomial provides a sufficient approximation within the given interval. Figure 2, for example, plots the original sigmoid function, its least squares fitting polynomial (of degree 3) within the interval $[-8, 8]$, and its Taylor expansion (of degree 3) at the point $x = 0$. Note that the Taylor polynomial provides an accurate approximation only around the given point, while the least squares fitting polynomial provides a good approximation in a wider range.

Baseline Algorithm The Algorithm 1 shows the resulting baseline algorithm. Note that each sample (row) $\boldsymbol{z}_i$ of the training data Z_i is structured by $\boldsymbol{z}_i = y_i' \cdot (1, \boldsymbol{x}_i) \in \mathbb{R}^f$, where $y_i' = 2y_i - 1 \in \{-1, 1\}$, and $\boldsymbol{x}_i$ and y_i are the original input samples and its class output, respectively (as described in Section 2.1).

Algorithm 1 HE-friendly logistic regression algorithm

```
**Input:** Mini-batches of training data $\{Z_i\}$ where $Z_i \in \mathbb{R}^{m \times f}$ (i.e., the mini-batch
    size is $m$), parameters $\gamma$ and $\eta$, the number of iterations $K$, and a polynomial
    approximation of sigmoid $\sigma'$
**Output:** Weight vectors $\boldsymbol{w}, \boldsymbol{v} \in \mathbb{R}^f$
 1: Initialize weight vector: $\boldsymbol{w}, \boldsymbol{v} \leftarrow \mathbf{0}$
 2: **for** $k$ in $[1..K]$ **do**
 3:     Select a mini-batch $Z_i$ (in order, or at random)
 4:     $\boldsymbol{a} = Z_i \cdot \boldsymbol{v}$
 5:     **for** $j$ in $[1..m]$ **do**
 6:         $b_j = \sigma'(a_j)$
 7:     **end for**
 8:     $\boldsymbol{\Delta} = \sum_{j=0}^{m-1} b_j \cdot Z_i[j]$
 9:     $\boldsymbol{w}^+ = \boldsymbol{v} - \gamma \cdot \boldsymbol{\Delta}$
10:     $\boldsymbol{v}^+ = (1 - \eta) \cdot \boldsymbol{w}^+ + \eta \cdot \boldsymbol{w}$
11:     $\boldsymbol{w} = \boldsymbol{w}^+, \ \boldsymbol{v} = \boldsymbol{v}^+$
12: **end for**
```

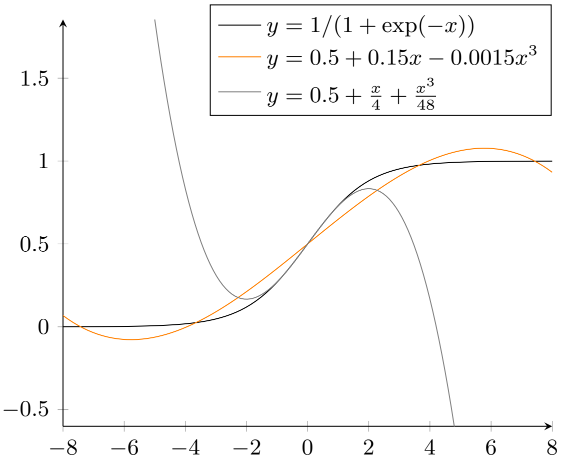


Fig. 2: Sigmoid (the first) and its two approximations using the least squares fitting method (the second) and the Taylor expansion (the third).

4.2 HE-Optimized Logistic Regression Algorithm

Now we optimize the baseline algorithm (Algorithm 1) to be efficiently evaluated in HEs against large encrypted data. Specifically, we optimize the body of the main iteration loop (lines 3–11 of Algorithm 1). Conceptually, the optimization consists of two parts: vectorization using homomorphic SIMD operations, and fine-tuning the evaluation order. In this section, we explain the first part, which will result in the vectorized body of the main iteration loop as shown in Algorithm 4. We will explain the second part in the next section.

Let us first define some notations. For two matrices A and B, we write $A + B$ and $A \circ B$ to denote the addition and the element-wise multiplication (i.e., Hadamard product) of A and B, respectively. Also, we write $A^{\circ k}$ to denote the element-wise exponentiation, i.e., $A^{\circ k} = \{a_{i,j}^k\}$ for $A = \{a_{i,j}\}$.

Partition and Encryption of Training Data Assume that the training data $\{x_{i,j}\}$ consists of n samples over $f-1$ features, throughout this section. This data can be seen as an $n \times f$ matrix Z including the target $\{y_i\}$ as follows:

$$Z = \begin{bmatrix} z[0][0], & z[0][1], & \cdots, & z[0][f-1] \\ z[1][0], & z[1][1], & \cdots, & z[1][f-1] \\ & & \vdots & \\ z[n-1][0], & z[n-1][1], & \cdots, & z[n-1][f-1] \end{bmatrix}$$

where $z[i][0] = y_i$ and $z[i][j+1] = y_i \cdot x_{i,j}$ for $0 \le i < n$ and $0 \le j < f-1$.[6]

We divide Z into multiple $m \times g$ sub-matrices $Z_{i,j}$ (for $0 \le i < n/m$ and $0 \le j < f/g$) as follows:

$$Z_{i,j} = \begin{bmatrix} z[mi][gj], & \cdots, & z[mi][gj+(g-1)] \\ z[mi+1][gj], & \cdots, & z[mi+1][gj+(g-1)] \\ & \vdots & \\ z[mi+(m-1)][gj], & \cdots, & z[mi+(m-1)][gj+(g-1)] \end{bmatrix}$$

$Z_{i,j}$ is supposed to be packed into a single ciphertext, and thus we set m and g in a way that utilizes the maximum ciphertext slots, $N/2$, that is, $m \times g = N/2$. Also, we set g to the same size of the partition of a weight vector for the bootstrapping parallelization, which in turn decides m, the size of a mini-batch block.

To encrypt $Z_{i,j}$ in a single ciphertext, we first represent it in a vector $\boldsymbol{p}_{i,j}$:

$$\boldsymbol{p}_{i,j}[k] = Z_{i,j}[\lfloor k/g \rfloor][k \bmod g] \quad (0 \le k < g \cdot m)$$

and encrypt $\boldsymbol{p}_{i,j}$ using the scheme described in Section 3:

$$\mathtt{encZ}[i][j] = \mathsf{encrypt}(\boldsymbol{p}_{i,j}; \Delta_z)$$

Note that we have nf/mg ciphertexts to encrypt the whole training data.

Partition and Encryption of Weight Vectors We have two weight vectors, $\boldsymbol{w}$ and $\boldsymbol{v}$, of size f in our logistic regression algorithm due to the NAG optimization (as shown in Section 4.1). We divide each of them into multiple sub-vectors, $\boldsymbol{w}_i$ and $\boldsymbol{v}_i$, for the purpose of the bootstrapping parallelization. Then we construct matrices, W_i and V_i, each of which consists of m duplicates of each of sub-vectors, $\boldsymbol{w}_i$ and $\boldsymbol{v}_i$, as follows:

$$W_i = \begin{bmatrix} w[gi], \ w[gi+1], \ \cdots, \ w[gi+(g-1)] \\ w[gi], \ w[gi+1], \ \cdots, \ w[gi+(g-1)] \\ \vdots \\ w[gi], \ w[gi+1], \ \cdots, \ w[gi+(g-1)] \end{bmatrix}$$

[6] We have $y_i \cdot x_{i,j}$ instead of $x_{i,j}$ for a simpler representation of the gradient descent method, as described in Section 2.1. This representation also has an advantage for computing a gradient $\Delta_w \ell(\boldsymbol{v}_i)$ over ciphertexts.

$$V_i = \begin{bmatrix} v[gi], \, v[gi+1], \, \cdots, \, v[gi+(g-1)] \\ v[gi], \, v[gi+1], \, \cdots, \, v[gi+(g-1)] \\ \vdots \\ v[gi], \, v[gi+1], \, \cdots, \, v[gi+(g-1)] \end{bmatrix}$$

We write $\texttt{encW}[i]$ and $\texttt{encV}[i]$ to denote encryptions of these matrices. We initialize them to be an encryption of a zero vector.

Homomorphic Evaluation of Inner Product One of the essential operations of logistic regression is the inner product. If we have m samples over g features, then for each iteration, we have to compute m inner products on vectors of size g, where each inner product requires g^2 multiplication and $g-1$ addition operations, that is, $m \cdot (g^2 \cdot \texttt{mult} + g \cdot \texttt{add})$ operations in total. Now we will show an optimized, batch inner product method using SIMD-addition, SIMD-multiplication, and rotation operations, which requires only *two* SIMD-multiplication operations and $2 \log g$ rotation-and-SIMD-addition operations to compute the m inner products, that is, $2 \cdot \texttt{SIMDmult} + 2 \log g \cdot (\texttt{rot} + \texttt{SIMDadd})$ in total. This batch method is extremely efficient in the packed HEs where SIMD operations provide high throughput at no additional cost compared to non-SIMD operations.

The batch inner product method is as follows. Suppose we want to compute $Z \cdot \boldsymbol{v}$ where $Z \in \mathbb{R}^{m \times g}$ and $\boldsymbol{v} \in \mathbb{R}^g$. Assume that g is a power of two.[7] First, we construct a matrix V that consists of m duplicate row-vectors of $\boldsymbol{v}$ as described in Section 4.2. Then, we can compute the Hadamard product, $Z \circ V$, by conducting a *single* SIMD-multiplication as follows:

$$Z \circ V = \begin{bmatrix} Z[1][1] \cdot v[1], & Z[1][2] \cdot v[2], & \cdots, & Z[1][g] \cdot v[g] \\ Z[2][1] \cdot v[1], & Z[2][2] \cdot v[2], & \cdots, & Z[2][g] \cdot v[g] \\ \vdots & \vdots & \ddots & \vdots \\ Z[m][1] \cdot v[1], & Z[m][2] \cdot v[2], & \cdots, & Z[m][g] \cdot v[g] \end{bmatrix}$$

Now, we need to compute the summation of the columns, which becomes the inner product result. We can compute the summation by repeating the rotation-and-addition operations $\log g$ times as follows. Let $\mathsf{Lrot}_i(A)$ be a matrix obtained by rotating each element of A to the left by i. Then, recursively evaluating the following recurrence relation starting from $A^{(0)} = A$ will give us $A^{(g)}$, *in* $\log g$ *steps*, whose first column is the summation of the columns of A:

$$A^{(2^{k+1})} = A^{(2^k)} + \mathsf{Lrot}_{2^k}(A^{(2^k)}) = \begin{bmatrix} \Sigma_{i=1}^{2^{k+1}} Z[1][i] \cdot v[i] & \cdots & - \\ \Sigma_{i=1}^{2^{k+1}} Z[2][i] \cdot v[i] & \cdots & - \\ \vdots & \ddots & \vdots \\ \Sigma_{i=1}^{2^{k+1}} Z[m][i] \cdot v[i] & \cdots & - \end{bmatrix}$$

Note that the other columns except the first are garbage, denoted by $-$, in the above. We can clean up the garbage columns by multiplying the zero vectors, and then duplicate the first column by applying the rotation-and-addition method. See Algorithm 3 for the complete details.

[7] Otherwise, we can pad zero columns in the end to make it a power of two.

Algorithm 2 SumRowVec: summation of row-vectors

```
Input: Matrix $A$ with size $f \times g$ for a power of two $f$
Output: Matrix $R$ with size $f \times g$
$R := A$
for $0 \leq i < \log_2 f$ do
    $R = \mathsf{Lrot}_{g \cdot 2^i}(R) + R$
end for
return $R$
```

Algorithm 3 SumColVec: summation of column-vectors

```
Input: Matrix $A$ with size $f \times g$ for a power of two $g$
Output: Matrix $R$ with size $f \times g$
$R := A$
for $0 \leq i < \log_2 g$ do
    $R = \mathsf{Lrot}_{2^i}(R) + R$
end for
$D = \{D_{i,j}\}$, where $D_{i,j} = 1$ if $j = 0$ and 0 otherwise.
$R = R \circ D$
for $0 \leq i < \log_2 g$ do
    $R := \mathsf{Rrot}_{2^i}(R) + R$
end for
return $R$
```

Note that we can compute the summation of row-vectors in a similar way, as shown in Algorithm 2. Below we illustrate the results of two procedures, SumRowVec and SumColVec:

$$\mathsf{SumRowVec}(A) = \begin{bmatrix} \sum_i a[i][1], & \cdots, & \sum_i a[i][g] \\ \sum_i a[i][1], & \cdots, & \sum_i a[i][g] \\ \vdots, & \ddots, & \vdots \\ \sum_i a[i][1], & \cdots, & \sum_i a[i][g] \end{bmatrix}$$

$$\mathsf{SumColVec}(A) = \begin{bmatrix} \sum_j a[1][j], & \cdots, & \sum_j a[1][j] \\ \sum_j a[2][j], & \cdots, & \sum_j a[2][j] \\ \vdots, & \ddots, & \vdots \\ \sum_j a[f][j], & \cdots, & \sum_j a[f][j] \end{bmatrix}$$

for $A = \{a_{i,j}\} \in \mathbb{R}^{f \times g}$.

Vectorized Algorithm Algorithm 4 shows the resulting vectorized body of the main iteration loop using the approaches described so far in this section. At line 6, we use the least squares fitting polynomial approximation of sigmoid, $y = 0.5 + 0.15x - 0.0015x^3$ (depicted in Figure 2). The bold symbols and numbers denote $m \times g$ matrices that consist of duplicates of corresponding elements. Note that the approximated sigmoid function is evaluated only once per iteration even with the partitioned weight vectors. Also, note that the two loops of iterating over the partitioned weight vectors can be run in parallel.

Algorithm 4 Vectorized body of the iteration loop

```
Input: Matrices $Z_j$, $W_j$, and $V_j$ for $0 \le j < f/g$
Output: Matrices $W_j^+$ and $V_j^+$ for $0 \le j < f/g$
 1: for $0 \le j < f/g$ do
 2:     $M_j = Z_j \circ V_j$
 3:     $M_j = \mathsf{SumColVec}(M_j)$
 4: end for
 5: $M = \sum_{j=0}^{f/g} M_j$
 6: $S = \mathbf{0.5} + \mathbf{0.15} \circ M - \mathbf{0.0015} \circ M^{\circ 3}$
 7: for $0 \le j < f/g$ do
 8:     $S_j = S \circ Z_j$
 9:     $\Delta_j = \mathsf{SumRowVec}(S_j)$
10:     $W_j^+ = V_j - \boldsymbol{\gamma} \circ \Delta_j$
11:     $V_j^+ = (\mathbf{1} - \boldsymbol{\eta}) \circ W_j^+ + \boldsymbol{\eta} \circ W_j$
12: end for
13: return $W_j^+$ and $V_j^+$ for $0 \le j < f/g$
```

4.3 Further Optimization

Now we explain the further optimization made on the top of Algorithm 4 by fine-tuning the evaluation order to minimize both the depth and the noise of multiplications. Our final HE-optimized algorithm is given in Algorithm 5.[8]

Minimizing Multiplication Depth In homomorphic evaluation, minimizing the depth of nested multiplications is critical to optimize the performance. The larger the multiplication depth, the larger the ciphertext modulus and/or the more often the bootstrapping operation needs to be executed. A large ciphertext modulus significantly increases the computation overhead, and the bootstrapping operation is very expensive. For example, when computing x^n, a naive method would require the nested multiplications of depth $n-1$, but an optimized method such as the square-and-multiply method would require only the multiplication depth of $\log n$.

We further optimize Algorithm 4 by minimizing the multiplication depth. A naive evaluation of Algorithm 4 requires the multiplication depth of 7. We reduce the depth to 5, by using the square-and-multiply method with further adjusting the evaluation order. This depth reduction allows us to reduce the size of the ciphertext modulus, improving the performance. Note that our depth minimization method will achieve a bigger depth reduction as a larger-degree polynomial is used in the sigmoid approximation (at line 6 in Algorithm 4).

Figure 3 illustrates our optimized evaluation of Algorithm 4 using the depth minimization method. It shows the optimized evaluation in the form of a circuit, where the inputs are given in the top of the figure, and the outputs are in the bottom. The circuit is layered by the multiplication depth, where each layer is

[8] The definitions of $\mathsf{encSumRowVec}$ and $\mathsf{encSumColVec}$ are provided in Appendix.

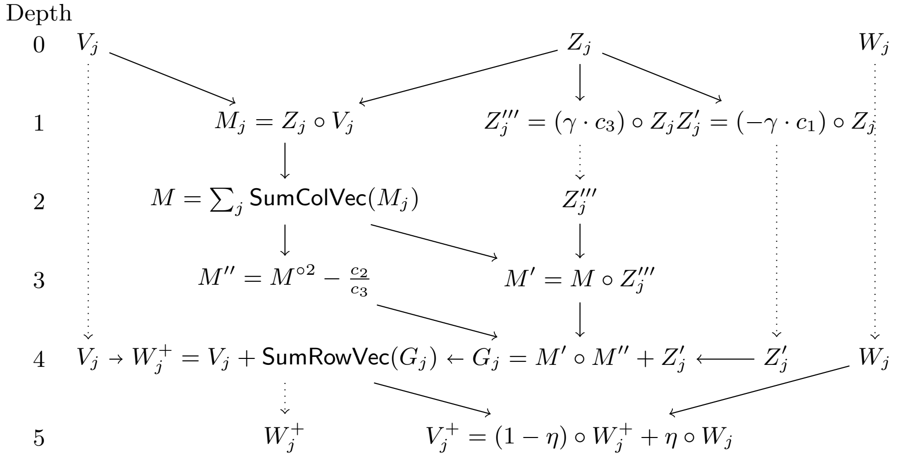

Depth
0
1
2
3
4
5
V_j
Z_j
W_j
$M_j = Z_j \circ V_j$
$Z_j''' = (\gamma \cdot c_3) \circ Z_j$
$Z_j' = (-\gamma \cdot c_1) \circ Z_j$
$M = \sum_j \mathsf{SumColVec}(M_j)$
Z_j'''
$M'' = M^{\circ 2} - \frac{c_2}{c_3}$
$M' = M \circ Z_j'''$
$V_j \rightarrow W_j^+ = V_j + \mathsf{SumRowVec}(G_j) \leftarrow G_j = M' \circ M'' + Z_j' \longleftarrow Z_j'$
W_j
W_j^+
$V_j^+ = (1-\eta) \circ W_j^+ + \eta \circ W_j$

Fig. 3: An optimized evaluation circuit of Algorithm 4 using the depth minimization method. The circuit is layered by the multiplication depth (in the left-hand side), where each layer consists of either normal multiplication (mult) or constant multiplication (cMult), with zero or more addition (add) operations. The solid arrow denotes the input wiring, and the dotted arrow denotes the value propagation. Since the circuit is layered by only the multiplication depth, the inputs of the addition operation are put in the same layer (e.g., as shown in the fourth layer). Algorithm 5 incorporates this optimized evaluation circuit.

labeled with its multiplication depth (in the left-hand side), i.e., the input layer is labeled with 0, and the output layer is labeled with 5. Given the inputs V_j, Z_j, and W_j,[9] the first layer computes $M_j = Z_j \circ V_j$ (corresponding to the line 2 in Algorithm 4), and $Z' = (-\gamma \cdot c_1) \circ Z_j$ and $Z''' = (\gamma \cdot c_3) \circ Z_j$ (corresponding to the partial computation of the lines 6, 8, and 10). The second layer computes $M = \Sigma_j(\mathsf{SumColVec}(M_j))$ (corresponding to the lines 3 and 5). The third layer computes $M' = M \circ Z'''$ and $M'' = M^{\circ 2} - \frac{c_2}{c_3}$. The fourth layer computes $G = M' \circ M'' + Z'$ and $W_j^+ = V_j + \mathsf{SumRowVec}(G)$. The fifth layer computes $V_j^+ = (1-\eta) \circ W_j^+ + \eta \circ W_j$. Note that $\mathsf{SumRowVec}(G)$ computed in the fourth layer effectively computes $-\gamma \circ \Delta_j$ (at line 10 in Algorithm 4).[10] Also note that the computation of $\mathsf{SumRowVec}(G)$ requires only the multiplication depth of 3, while a naive evaluation of $-\gamma \circ \Delta_j$ would require the multiplication depth of 5. In general, if we use a degree n polynomial approximation (at line 6 in

[9] Indeed, the whole evaluation circuit consists of duplicates of the presented circuit for each j being arranged side-by-side, which effectively parallelizes the loops in Algorithm 4.

[10] $\mathsf{SumRowVec}$ (G) $= \mathsf{SumRowVec}(M' \circ M'' + Z') = \mathsf{SumRowVec}((M \circ (\gamma \cdot c_3) \circ Z_j) \circ (M^{\circ 2} - \frac{c_2}{c_3}) + (-\gamma \cdot c_1) \circ Z_j) = \mathsf{SumRowVec}(-\gamma \cdot Z_j \circ (c_1 + c_2 \circ M - c_3 \circ M^{\circ 2})) = \mathsf{SumRowVec}(-\gamma \cdot Z_j \circ S) = \mathsf{SumRowVec}(-\gamma \cdot S_j) = -\gamma \circ \mathsf{SumRowVec}(S_j) = -\gamma \circ \Delta_j$

Algorithm 4), our depth minimization method will reduce the multiplication depth from $O(n)$ to $O(\log n)$.

Minimizing Approximation Noise Recall that the approximate HE used in our algorithm introduces an additional noise for each homomorphic operation. Even the homomorphic rotation and rescaling operations introduce the noise. We further optimize our algorithm to minimize the noise by reordering the evaluation order of homomorphic operations. For example, the rescaling operation has an effect of reducing the previously introduced noise. Reordering the rescaling operations, thus, can reduce the overall accumulated noise. Let us illustrate the approach. Suppose we want to multiply two ciphertexts $c_1 = \mathsf{Enc}(\boldsymbol{m_1})$ and $c_2 = \mathsf{Enc}(\boldsymbol{m_2})$, and rotate the multiplication result. Let $\boldsymbol{m}_3 = (\boldsymbol{m}_1 \circ \boldsymbol{m}_2)$. A naive way of computing that would have the following evaluation order:

$$
\begin{aligned}
c_3 &= \mathsf{Mult}(\mathsf{Enc}(\boldsymbol{m}_1), \mathsf{Enc}(\boldsymbol{m}_2)) = \mathsf{Enc}(\boldsymbol{m}_3 \cdot \mathit{\Delta} + \epsilon_1) \\
c_4 &= \mathsf{Rescale}(c_3, \mathit{\Delta}) = \mathsf{Enc}(\boldsymbol{m}_3 + \epsilon_1/\mathit{\Delta} + \epsilon_2) \\
c_5 &= \mathsf{Rotate}(c_4, i) = \mathsf{Enc}(\mathsf{Lrot}_i(\boldsymbol{m}_3) + \epsilon_1/\mathit{\Delta} + \epsilon_2 + \epsilon_3)
\end{aligned}
$$

where $\mathit{\Delta}$ is the scaling factor, and ϵ_i is the noise. However, we can reduce the final noise by adjusting the evaluation order, i.e., by swapping the rescaling operation and the rotation operation, as follows:

$$
\begin{aligned}
c_3 &= \mathsf{Mult}(\mathsf{Enc}(\boldsymbol{m}_1), \mathsf{Enc}(\boldsymbol{m}_2)) = \mathsf{Enc}(\boldsymbol{m}_3 \cdot \mathit{\Delta} + \epsilon_1) \\
c_4' &= \mathsf{Rotate}(c_3, i) = \mathsf{Enc}(\mathsf{Lrot}_i(\boldsymbol{m}_3) \cdot \mathit{\Delta} + \epsilon_1 + \epsilon_2) \\
c_5' &= \mathsf{Rescale}(c_4', \mathit{\Delta}) = \mathsf{Enc}(\mathsf{Lrot}_i(\boldsymbol{m}_3) + (\epsilon_1 + \epsilon_2)/\mathit{\Delta} + \epsilon_3)
\end{aligned}
$$

Note that the final noise is reduced from $\epsilon_1/\mathit{\Delta} + \epsilon_2 + \epsilon_3$ to $(\epsilon_1 + \epsilon_2)/\mathit{\Delta} + \epsilon_3$. Since $\epsilon_2 \ll \mathit{\Delta}$, this optimization effectively removes ϵ_2.

5 Evaluation

We evaluate our algorithm of logistic regression on encrypted data against both a real financial training dataset and the MNIST dataset. Our artifact is publicly available at [Ano18].

5.1 Logistic Regression on Encrypted Financial Dataset

We executed our algorithm on a private, real financial dataset to evaluate the efficiency and the scalability of our algorithm on a large dataset.

Training Dataset The *encrypted* dataset we consider to evaluate our logistic regression algorithm is the real consumer credit information maintained by a credit reporting agency.The dataset (for both training and validation), randomly sampled by the agency, consists of 1,266,325 individuals' credit information over

Algorithm 5 HE-optimized body of the iteration loop

```
Input: Ciphertexts $\mathsf{encZ}_j$, $\mathsf{encW}_j$, and $\mathsf{encV}_j$ for $0 \le j < f/g$, and parameters $\mathsf{wBits}$
    and $\mathsf{pBits}$
Output: Ciphertexts $\mathsf{encW}_j^+$ and $\mathsf{encV}_j^+$ for $0 \le j < f/g$
for $0 \le j < f/g$ do
    $\mathsf{encM}_j = \mathsf{rescale}(\mathsf{mult}(\mathsf{encZ}_j, \mathsf{encV}_j), \mathsf{wBits})$
    $\mathsf{encM}_j = \mathsf{encSumColVec}(\mathsf{encM}_j, \mathsf{pBits})$
end for
$\mathsf{encM} = \sum_{j=0}^{f/g} \mathsf{encM}_j$
$\mathsf{encM}'' = \mathsf{rescale}(\mathsf{mult}(\mathsf{encM}, \mathsf{encM}), \mathsf{wBits})$
$\mathsf{encM}'' = \mathsf{cAdd}(\mathsf{encM}'', -\mathbf{100}, \mathsf{wBits})$
for $0 \le j < f/g$ do
    $\mathsf{encZ}' = \mathsf{cMult}(\mathsf{encZ}_j, -\boldsymbol{\gamma} \circ \mathbf{0.5}, \mathsf{wBits})$
    $\mathsf{encZ}''' = \mathsf{cMult}(\mathsf{encZ}_j, \boldsymbol{\gamma} \circ \mathbf{0.0015}, \mathsf{wBits})$
    $\mathsf{encZ}''' = \mathsf{modDownTo}(\mathsf{encZ}''', \mathsf{encM})$
    $\mathsf{encM}' = \mathsf{rescale}(\mathsf{mult}(\mathsf{encM}, \mathsf{encZ}'''), \mathsf{wBits})$
    $\mathsf{encG} = \mathsf{rescale}(\mathsf{mult}(\mathsf{encM}', \mathsf{encM}''), \mathsf{wBits})$
    $\mathsf{encG} = \mathsf{add}(\mathsf{encG}, \mathsf{modDownTo}(\mathsf{encZ}', \mathsf{encG}))$
    $\mathsf{encG} = \mathsf{encSumRowVec}(\mathsf{encG})$
    $\mathsf{encW}_j^+ = \mathsf{add}(\mathsf{encG}, \mathsf{modDownTo}(\mathsf{encV}_j, \mathsf{encG}))$
    $\mathsf{encW}_j = \mathsf{modDownTo}(\mathsf{encW}_j, \mathsf{encW}_j^+)$
    $\mathsf{encW}_{j,1}^+ = \mathsf{cMult}(\mathsf{encW}_j^+, \mathbf{1} - \boldsymbol{\eta}, \mathsf{pBits})$
    $\mathsf{encW}_{j,2}^+ = \mathsf{cMult}(\mathsf{encW}_j, \boldsymbol{\eta}, \mathsf{pBits})$
    $\mathsf{encV}_j^+ = \mathsf{add}(\mathsf{encW}_{j,1}^+, \mathsf{encW}_{j,2}^+)$
    $\mathsf{encV}_j^+ = \mathsf{rescale}(\mathsf{encV}_j^+, \mathsf{pBits})$
    $\mathsf{encW}_j^+ = \mathsf{modDownTo}(\mathsf{encW}_j^+, \mathsf{encV}_j^+)$
end for
return $\mathsf{encV}_j^+$ and $\mathsf{encW}_j^+$ for $0 \le j < f/g$
```

200 features that are used for credit rating. Examples of the features are the loan information (such as the number of credit loans and personal mortgages), the credit card information (such as the average amount of credit card purchases and cash advances in the last three months), and the delinquency information (such as the days of credit card delinquency). The samples are labeled with a binary classification that refers to whether each individual's credit rating is below the threshold.

HE Scheme Parameters We use two scaling factors $\Delta = 2^{30}$ and $\Delta_c = 2^{15}$, where Δ is the regular scaling factor (for mult) and Δ_c is the constant scaling factor (for cMult) that is used for multiplying constant matrices and scalars such as $\boldsymbol{\eta}$ and $\boldsymbol{\gamma}$. We have the number of ciphertext slots $N/2 = 2^{15}$.

We set the initial ciphertext modulus Q for the weight vectors W and V as follows:

$$\log_2 Q = 5 + \mathtt{wBits} + \mathtt{I} \cdot (3 \cdot \mathtt{wBits} + 2 \cdot \mathtt{pBits})$$

where `wBits` $= \log_2 \varDelta$ and `pBits` $= \log_2 \varDelta_c$. The above formula is derived from the fact that each iteration reduces $(3 \cdot \texttt{wBits} + 2 \cdot \texttt{pBits})$-bits of the ciphertext modulus (see Section 4.3 and Figure 3 for more details). Here, `I` is the number of iterations per bootstrapping operation; that is, the bootstrapping operation is executed every `I` iterations. We have `I` = 5. Also, we set the largest ciphertext modulus Q' for the bootstrapping, according to the HEAAN scheme [CHK+18, Kim18], as follows: $\log_2 Q' = \log_2 Q + 24 + 14 \cdot (9 + \texttt{wBits})$.

Data			Performances	
Financial	No. Samples (training)	422,108	Accuracy	80%
	No. Samples (validation)	844,217	AUROC	0.8
	No. Features	200	K-S value	50.84
	No. Iterations	200	Public Key Size	≈ 2 GB
	Learning Rate	0.01	Encrypted Block Size	4.87 MB
	Block Size (mini-batch)	512	Running Time	1060 min

Table 1: Result of machine learning on encrypted data

Experimental Results We executed our logistic regression algorithm on the encrypted training set of 422,108 samples over 200 features. Having 200 iterations, it took 1,060 minutes to learn an *encrypted* model, i.e., ∼5 minutes per iteration on average, in a machine with IBM POWER8 (8 cores, 4.0GHz) and 256GB RAM. We sent the learned model to the data owner, and they decrypted and evaluated it on the validation set of 844,217 samples, having 80% accuracy and the KS value of 50.84. They confirmed that it provides a sufficient accuracy compared to their internal model learned using the plaintext dataset[11] and also our learned model gives appropriate weights on the important features (e.g., delinquency, loan, and credit card information) as expected.

Tabel 1 shows the detailed result of our experiment. We set the learning rate to be 0.01, and the mini-batch size to be 512. The ciphertext size of each mini-batch block is 4.87 MB, and thus the total size of the encrypted dataset is ∼4 GB = 4.87 MB × (422,108 / 512). The public key size is ∼2 GB.

5.2 Logistic Regression on Encrypted MNIST Dataset

We executed our logistic regression algorithm on the public MNIST dataset to provide a more detailed evaluation.

Training Dataset and Parameters We took the MNIST dataset [LCB99], and restructured it for the binary classification problem between 3 and 8. We

[11] According to their report, it took several minutes to learn a model on the plaintext using the same algorithm, and the model provides the KS value of 51.99.

compressed the original images of 28×28 pixels into 14×14 pixels, by compressing 2×2 pixels to their arithmetic mean. The restructured dataset consists of 11,982 samples of the training dataset and 1,984 samples of the validation dataset.

We use the same principle for setting the HE scheme parameters as shown in Section 5.1. We set $\varDelta = 2^{40}$, $\varDelta_c = 2^{15}$, and $\mathtt{I} = 3$. Also, we approximate the sigmoid function with the interval $[-16, 16]$ by the least squares fitting polynomial of degree 3, $y = 0.5 - 0.0843x + 0.0002x^3$.

Data			Performances	
MNIST	No. Samples (training)	11,982	Accuracy	96.4%
	No. Samples (validation)	1,984	AUROC	0.99
	No. Features	196	K-S value	N/A
	No. Iterations	32	Public Key Size	$\approx$ 1.5 GB
	Learning Rate	1.0	Encrypted Block Size	3.96 MB
	Block Size (mini-batch)	1024	Running Time	132 min

Table 2: Result of machine learning on encrypted data

Experimental Results We encrypted the MNIST dataset and executed our logistic regression algorithm. Table 2 shows the result. With 32 iterations, our logistic algorithm took 132 minutes to learn an encrypted model. The average time for each iteration is ~4 minutes, which is similar to that of the financial dataset, as expected. We decrypted the learned model and evaluated it on the validation dataset, obtaining 96.4% accuracy.[12]

Microbenchmarks We also executed our logistic regression algorithm on the plaintext dataset, and compared the result to that of the ciphertext learning. Recall that the approximate HE used in our algorithm introduces the approximation noise for each computation step, but it had not been clear how much the noise affects the overall training process. To evaluate the impact of the approximation noise on the overall learning performance (e.g., the convergence rate and accuracy), we measured the accuracy for each iteration for both plaintext and ciphertext training, and compared those results. Figure 4 shows the comparison result. It shows that the accuracy for each iteration in the ciphertext training is marginally different from that of the plaintext, especially in the early stage of the training process, but they eventually converged at the final step. This result implies that the additional noise introduced by the approximate HE evaluation is not significant to deteriorate the accuracy of a learned model and the training performance.

[12] The accuracy seems to be lower than the usual, but the difference is mainly due to the image compression, not because of the approximation noise. See Section 5.2 and Figure 4 for more details.

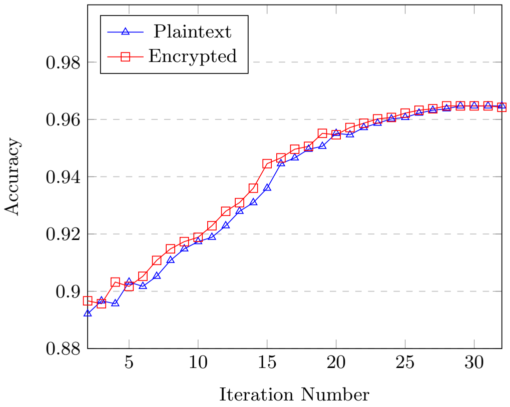


Fig. 4: Comparison between encrypted and plaintext training

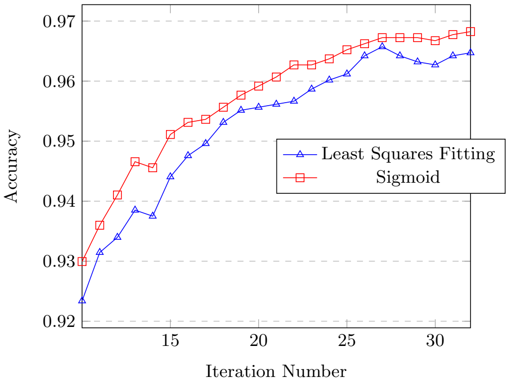


Fig. 5: Comparison between sigmoid and least squares fitting (of degree 3)

We also evaluate the effect of the precision of the polynomial approximation of sigmoid. We executed the same algorithm (on the plaintext) with three different sigmoid approximations: the original sigmoid (i.e., no approximation), the least squares fitting polynomial, and the Taylor expansion polynomial (depicted in Figure 2). Figure 5 and 6 show the comparison of accuracy between them. It shows that the approximation error of the least fitting polynomial is not significant, resulting in only the marginal difference of accuracy. However, the approximation error of the Taylor expansion polynomial is so large that it fails to learn a model; that is, the accuracy decreases as the number of iteration increases, and eventually it becomes 0 (i.e., an invalid model).

5.3 Discussion

It is not straightforward to provide the fair comparison of our performance with those of the related works, since the previous HE-based approaches are *not* ca-

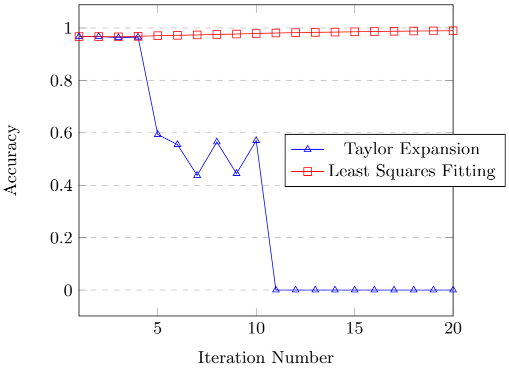


Fig. 6: Comparison between Taylor expansion between least squares fitting (both of degree 3)

pable of admitting such realistic size training datasets considered in this paper, and the MPC-based approaches do not support the same flexibility in the usage scenarios as ours. As a rough comparison, however, the recent MPC-based approach [MZ17] will take minutes[13] to learn a model on the MNIST dataset used in this paper, which is one or two orders of magnitude faster than ours. We note that, however, the MPC-based approach requires the additional assumption in the usage scenarios that either the number of participants is small, or the two servers do not collude.

Our algorithm requires the number of iterations to be provided in advance, which is inevitable due to the security of the underlying HE schemes. In our experiment on the financial data, the number was obtained by asking the data owner to provide a rough bound. We note that, however, one can use our algorithm in an interactive way that the data owners decrypt the learned model periodically (e.g., every 100 iterations), and decide whether to proceed further or not, depending on the quality of the model at the moment.

6 Conclusion and Further Work

In this paper, we presented an efficient logistic regression algorithm on large (fully) homomorphically encrypted data, and evaluated it against both the private financial data and the public MNIST dataset. Our implementation successfully learned a quality model in about 17 and 2 hours, respectively, which demonstrates the practical feasibility of our algorithm on realistic size data. We believe that the techniques we developed here can be also readily used for homomorphically evaluating other machine learning algorithms such as neural networks, which we leave as a future work.

[13] The time is obtained by extrapolating their experimental result on the MNIST dataset.

References

[ACC+17] David Archer, Lily Chen, Jung Hee Cheon, Ran Gilad-Bachrach, Roger A. Hallman, Zhicong Huang, Xiaoqian Jiang, Ranjit Kumaresan, Bradley A. Malin, Heidi Sofia, Yongsoo Song, and Shuang Wang. Applications of homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, July 2017.

[AHPW16] Yoshinori Aono, Takuya Hayashi, Le Trieu Phong, and Lihua Wang. Scalable and secure logistic regression via homomorphic encryption. Cryptology ePrint Archive, Report 2016/111, 2016. https://eprint.iacr.org/2016/111.

[Ano18] Anonymous. http://anonymous.4open.science/repository/0b20aa40-6f0c-46bf-887e-933b6d4952fa/, 2018.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

[BLN14] Joppe W Bos, Kristin Lauter, and Michael Naehrig. Private predictive analysis on encrypted medical data. *Journal of biomedical informatics*, 50:234–243, 2014.

[BMMP17] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. Cryptology ePrint Archive, Report 2017/1114, 2017. https://eprint.iacr.org/2017/1114.

[BPTG15] Raphael Bost, Raluca Ada Popa, Stephen Tu, and Shafi Goldwasser. Machine learning classification over encrypted data. In *NDSS*, volume 4324, page 4325, 2015.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *CRYPTO*, pages 868–886, 2012.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology–CRYPTO 2011*, pages 505–524. Springer, 2011.

[BV14] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. *SIAM Journal on Computing*, 43(2):831–871, 2014.

[CBD14] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[CDN01] Ronald Cramer, Ivan Damgård, and Jesper B Nielsen. Multiparty computation from threshold homomorphic encryption. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 280–300. Springer, 2001.

[CdWM+17] Hervé Chabanne, Amaury de Wargny, Jonathan Milgram, Constance Morel, and Emmanuel Prouff. Privacy-preserving classification on deep neural network. *IACR Cryptology ePrint Archive*, 2017:35, 2017.

[CGGI17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Improving TFHE: faster packed homomorphic operations and efficient circuit bootstrapping. 2017. https://eprint.iacr.org/2017/430.

[CHK+18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In

Annual International Conference on the Theory and Applications of Cryptographic Techniques, pages 360–384. Springer, 2018.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.

[CLT14] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. Scale-invariant fully homomorphic encryption over the integers. In *Public-Key Cryptography–PKC 2014*, pages 311–328. Springer, 2014.

[DM15] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping homomorphic encryption in less than a second. In *Advances in Cryptology–EUROCRYPT 2015*, pages 617–640. Springer, 2015.

[DSFRO17] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 561–574. ACM, 2017.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. http://eprint.iacr.org/.

[GAGN15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[GBDL$^+$16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210, 2016.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO 2012*, pages 850–867. Springer, 2012.

[GLN12] Thore Graepel, Kristin Lauter, and Michael Naehrig. Ml confidential: Machine learning on encrypted data. In *International Conference on Information Security and Cryptology*, pages 1–21. Springer, 2012.

[HS15] Shai Halevi and Victor Shoup. Bootstrapping for HElib. In *Advances in Cryptology–EUROCRYPT 2015*, pages 641–670. Springer, 2015.

[JRS17] Aayush Jain, Peter MR Rasmussen, and Amit Sahai. Threshold fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2017:257, 2017.

[JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.

[JWB$^+$17] Karthik A Jagadeesh, David J Wu, Johannes A Birgmeier, Dan Boneh, and Gill Bejerano. Deriving genomic diagnoses without revealing patient genomes. *Science*, 357(6352):692–695, 2017.

[Kim18] Andrey Kim. HEAANBOOT. https://github.com/kimandrik/HEAANBOOT, 2018.

[KSK$^+$18] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. Logistic regression model training based on the approximate homomorphic encryption. 2018.

[KSW⁺18] Miran Kim, Yongsoo Song, Shuang Wang, Yuhou Xia, and Xiaoqian Jiang. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR medical informatics*, 6(2):e19, 2018.

[LATV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234. ACM, 2012.

[LCB99] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The MNIST Database of Handwritten Digits, 1999.

[LLH⁺17] Ping Li, Jin Li, Zhengan Huang, Chong-Zhi Gao, Wen-Bin Chen, and Kai Chen. Privacy-preserving outsourced classification in cloud computing. *Cluster Computing*, pages 1–10, 2017.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[MZ17] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 19–38. IEEE, 2017.

[NWI⁺13] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. Privacy-preserving ridge regression on hundreds of millions of records. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 334–348. IEEE, 2013.

[OU98] Tatsuaki Okamoto and Shigenori Uchiyama. A new public-key cryptosystem as secure as factoring. In *International conference on the theory and applications of cryptographic techniques*, pages 308–318. Springer, 1998.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.

[RAD78] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. On data banks and privacy homomorphisms. 1978.

[Reg09] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):34, 2009.

[vDGHV10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in cryptology–EUROCRYPT 2010*, pages 24–43. Springer, 2010.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

[ZLK⁺16] Hantian Zhang, Jerry Li, Kaan Kara, Dan Alistarh, Ji Liu, and Ce Zhang. The zipml framework for training models with end-to-end low precision: The cans, the cannots, and a little bit of deep learning. *arXiv preprint arXiv:1611.05402*, 2016.

A Appendix: Detailed Algorithms

We provide the detailed procedures of encSumRowVec and encSumColVec that are used in Algorithm 5.

Algorithm 6 encSumRowVec: summation of row-vectors in ciphertext

Input: A ciphertext c, an encryption of $f \times g$ matrix satisfying $f \cdot g = \frac{N}{2}$.
Output: A ciphertext c'

```
$c' := c$
for $0 \le i < \log_2 f$ do
    cLrot = leftRotate($c', g \cdot 2^i$)
    $c'$ = add($c'$, cLrot)
end for
return $c'$
```

Algorithm 7 encSumColVec: summation of column-vectors in ciphertext

Input: A ciphertext c, an encryption of $f \times g$ matrix satisfying $f \cdot g = \frac{N}{2}$. A scaling parameter pBits.
Output: A ciphertext c'

```
$c' := c$
for $0 \le i < \log_2 g$ do
    cLrot = leftRotate($c', 2^i$)
    $c'$ = add($c'$, cLrot)
end for
$\boldsymbol{d} := (d_1, \cdots, d_{N/2})$, where $d_j = 1$ if $g$ divides $j$ and 0 otherwise.
$c'$ = cMultByVec($c', \boldsymbol{d}$, pBits)
$c'$ = rescale($c'$, pBits)
for $0 \le i < \log_2 g$ do
    cLrot = leftRotate($c', \frac{N}{2} - 2^i$)
    $c'$ = add($c'$, cLrot)
end for
return $c'$
```