


Privacy-Preserving Logistic Regression Training with a Faster Gradient Variant

 John Chiang

john.chiang.smith@gmail.com

January 27, 2022

ABSTRACT

Logistic regression training on an encrypted dataset has been an attractive idea to security concerns for years. In this paper, we propose a faster gradient variant called Quadratic Gradient for logistic regression and implement it via a special homomorphic encryption scheme. The core of this gradient variant can be seen as an extension of the simplified fixed Hessian from Newton’s method, which extracts information from the Hessian matrix into the naive gradient, and thus can be used to enhance Nesterov’s accelerated gradient (NAG), Adagrad, etc. We evaluate various gradient *ascent* methods with this gradient variant on the gene dataset provided by the 2017 iDASH competition and the image dataset from the MNIST database. Experimental results show that the enhanced methods converge faster and sometimes even to a better convergence result. We also implement the gradient variant in full batch NAG and mini-batch NAG for training a logistic regression model on a large dataset in the encrypted domain. Equipped with this gradient variant, full batch NAG and mini-batch NAG are both faster than the original ones.

Keywords Quadratic Gradient · Homomorphic Encryption · Fixed Hessian · Logistic Regression · Privacy-Preserving

1 Introduction

1.1 Background

Logistic regression (LR) is a widely used classification technology especially in medical risk assessment due to its simplicity but powerful performance. Given a person’s healthcare data related to a certain disease, we can train an LR model capable of telling whether or not this person is likely to develop this disease. However, such personal health information is highly sensitive and private to individuals. The privacy concern, therefore, becomes a major obstacle for individuals to share their biomedical data. The most secure solution is to encrypt the data into ciphertexts first and then securely outsource the ciphertexts to the cloud, without allowing the cloud to access the data directly.

iDASH is an annual competition that aims to call for implementing interesting cryptographic schemes in a biological context. Since 2014, iDASH has included the theme of genomics and biomedical privacy. The third track of the 2017 iDASH competition and the second track of the 2018 iDASH competition were both to develop homomorphic-encryption-based solutions for building an LR model over encrypted data. Thanks to these two competitions, the performance of LR training based on homomorphic encryption (HE) has been significantly improved.

1.2 Related work

Several studies on logistic regression models are based on homomorphic encryption. Aono et al. [1] only used an additive HE scheme and left some of the challenging HE computations to a trusted client. Kim et al. [2] discussed the problem of performing LR training in an encrypted environment. They used the full batch gradient descent in the training process, and the least-squares method to get the approximation of the sigmoid function. They also used the CKKS scheme, which makes it possible to homomorphically approximate the sigmoid function. In the iDASH 2017 competition, Bonte and Vercauteren [3], Kim et al. [4], Chen et al. [5], and Crawford et al. [6] all investigated the same problem that Kim et al. [2] studied. In the iDASH competition of 2018, Kim et al. [7] and Blatt et al. [8] further worked on it for an efficient packing and semi-parallel algorithm.

The papers most relevant to this work are [3], [4], and [9]. Bonte and Vercauteren [3] developed a practical algorithm called the simplified fixed Hessian (SFH) method, yet which could not be used on the MNIST database due to the SFH matrix being singular in this case. Our study complements their work by generalizing SFH to be invertible in any case. We also adopt the ciphertext packing technique introduced by Kim et al. [4] for efficient homomorphic computation. Han et al. [9] and our work both adopt the HEAAN scheme and its approximate bootstrapping for LR training on encrypted data. They implemented the mini-batch NAG method; we put into practice the mini-batch NAG with our faster gradient variant.

1.3 Contributions

Our specific contributions in this paper are as follows:

1. We comprehensively analyze the current technologies that solve privacy-preserving logistic regression training and demonstrate that the SFH method proposed by Bonte and Vercauteren [3] converges faster than Nesterov’s accelerated gradient (NAG) used by Kim et al. [4].
2. We propose a new gradient variant, Quadratic Gradient, which can unite the first-order gradient (descent) method and the second-order Newton’s method as one. We implement it in various gradient methods, and the resulting methods show a better performance in the convergence rate (sometimes even a better convergence result) than the corresponding original gradient methods.
3. We adopt this faster gradient variant to enhance NAG and its mini-batch version in LR training based on fully homomorphic encryption. Our experimental results show that, like in the plaintext context, the enhanced methods implemented in ciphertext context are faster than the original ones, and almost as accurate as the ones in plaintext context.

2 Preliminaries

2.1 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE) is a type of cryptographic scheme that can be used to compute an arbitrary number of additions and multiplications directly on the encrypted data. It was not until the year 2009 that Gentry constructed the first FHE scheme via a bootstrapping operation [10]. The efficiency of homomorphic encryption schemes has been improved significantly in these years. However, it is still not sufficient for practical applications. FHE schemes themselves are computationally time-consuming; the choice of dataset encoding matters likewise to the efficiency. In addition to these two limits, how to manage the magnitude of plaintext [11] also contributes to the slowdown. Cheon et al. [12] proposed a method to construct a HE scheme with a rescaling procedure, which could eliminate this technical bottleneck effectively. We adopt their open-source implementation HEAAN while implementing our homomorphic LR algorithms.

HEAAN has implemented an approximate bootstrapping operation to support fully homomorphic encryption. Cheon et al. [13] resorted to two mathematical facts: $\sin x \approx x$ around the points $x = 2 \cdot k\pi$ for some k and the periodicity of the *sine* function, together to calculate the modulus operation in the decryption algorithm for a ciphertext. They also iteratively approximated the values of $\cos 2\theta$ and $\sin 2\theta$ from the values of $\cos \theta$ and $\sin \theta$ in virtue of double-angle formulas: $\cos(2\theta) = \cos^2 \theta - \sin^2 \theta$ and $\sin(2\theta) = 2 \cdot \cos \theta \cdot \sin \theta$. In this way they could evaluate the decryption circuit homomorphically so as to refresh low-level ciphertexts with small modulus.

It is inevitable to pack a vector of multiple plaintexts and encrypt it into a single ciphertext for yielding a better amortized time of homomorphic computation. HEAAN supports a parallel technique (aka SIMD) to pack multiple numbers in a single polynomial by virtue of Chinese Remainder Theorem, and provides rotation operation on plaintext slots. The two operations play an important role in various computations of the data encrypted in the ciphertexts.

The underlying HE scheme in HEAAN is well described in [4, 2, 9] and the related foundation theory of abstract algebra can be found in [14]. A simple description of main functions provided in HEAAN is as follows:

- *KeyGen*(1^λ): Given the preset security parameter λ and the default largest ciphertext modulus in HEAAN, the key generation procedure can determine the needed parameters of the HE scheme to set the secret key **sk**, the public key **pk**, the public rotation keys (also used for bootstrapping), and the public evaluation key **evk** that is used for relinearisation after a multiplication between ciphertexts.
- *Encrypt*(**pk**, **m**): The encryption procedure converts a complex-number vector of plaintexts **m** into a ciphertext **ct** with the help of public key **pk**.

- *Decrypt*(sk, ct): For a given ciphertext ct , the decryption procedure deciphers the ct using the secret key sk and returns a vector of complex numbers, which is the message \mathbf{m} in ct .
- *Add*(ct_1, ct_2): The homomorphic addition operation on two ciphertexts ct_1 and ct_2 returns a ciphertext $\text{ct}_{add} = \text{Encrypt}(\text{pk}, \mathbf{m}_1 \oplus \mathbf{m}_2)$, which is an encryption of the resultant vector of the element-wise addition of two messages \mathbf{m}_1 and \mathbf{m}_2 , separately carried by ct_1 and ct_2 .
- *Mult*(ct_1, ct_2): For two ciphertexts ct_1 and ct_2 that are encryptions of \mathbf{m}_1 and \mathbf{m}_2 respectively, the homomorphic multiplication operation returns a ciphertext $\text{ct}_{mult} = \text{Encrypt}(\text{pk}, \mathbf{m}_1 \otimes \mathbf{m}_2)$, where $\mathbf{m}_1 \otimes \mathbf{m}_2$ is the result of an element-wise multiplication between \mathbf{m}_1 and \mathbf{m}_2 .
- *Rescale*(ct, bits): The rescaling operation is an important operation first implemented by HEAAN, which is necessary to homomorphically calculate a complicated univariate polynomial for a floating-point number. After each homomorphic multiplication, the *Rescale* operation can reduce the magnitude of a plaintext to an appropriate level with a new scaling factor 2^{bits} smaller than the original, but preserve the same message of ct .
- *leftRotate*(ct, r): The left-rotation operation returns a ciphertext ct' encrypting the left-shifted plaintext vector of ct by r positions.
- *rightRotate*(ct, r): The right-rotation operation is just the opposing action of the left-rotation operation, and can be easily obtained using the left-rotation, namely, $\text{rightRotate}(\text{ct}, r) = \text{leftRotate}(\text{ct}, N_s - r)$ with the number N_s of slots.
- *Bootstrap*(ct): The bootstrapping operation return a new ciphertext ct' encrypting the same message but with a larger ciphertext modulus for more possible nested multiplications.

2.2 Database Encoding Method

Kim et al. [4] devised an efficient and promising encoding method for the given database, which could utilize parallel operations of ciphertexts as much as possible and almost made full use of the computation and storage resources. They packed a data matrix consisting of the training data combined with training-label information into a single ciphertext in a row-by-row manner. Unlike them [4], we encrypt the training data X and the training labels Y separately into two different ciphertexts in order to calculate our gradient variant at a cost of redundant storage of training label Y , retaining training-data information for future use. X and Y are encrypted into ciphertexts as follows:

$$\text{ct}_X = \text{Enc} \begin{bmatrix} 1 & x_{11} & \dots & x_{1d} \\ 1 & x_{21} & \dots & x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \dots & x_{nd} \end{bmatrix}, \text{ct}_Y = \text{Enc} \begin{bmatrix} y_1 & y_1 & \dots & y_1 \\ y_2 & y_2 & \dots & y_2 \\ \vdots & \vdots & \ddots & \vdots \\ y_n & y_n & \dots & y_n \end{bmatrix}, \text{ct}_Z = \text{Enc} \begin{bmatrix} y_1 & y_1 x_{11} & \dots & y_1 x_{1d} \\ y_2 & y_2 x_{21} & \dots & y_2 x_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ y_n & y_n x_{n1} & \dots & y_n x_{nd} \end{bmatrix},$$

where ct_Z is the ciphertext Kim et al. [4] produced on the client side. One homomorphic-encryption element-wise multiplication operation between ct_X and ct_Y generates ct_Z .

We adopt their database-encoding method for LR: we first partition the two-dimensional training dataset into multiple fixed-size matrices that still retain the data structure with complete sample information. The size of such a data matrix needs to be set in advance so that each of the matrices can be able to be packed into a single ciphertext in a row-by-row manner. Zeros can be padded into the last matrix in case the size of training dataset cannot be divisible by this preset size of matrix. We then pack such a whole matrix into a single ciphertext one by one and perform operations on each of these ciphertexts in parallel. Using this encoding scheme, we can perform HE operations on any element or any part of the plaintext matrix encrypted in the ciphertext, with the help of only three HE operations - rotation, addition, and multiplication.

Suppose that a ciphertext encrypting such a data matrix consists of n samples with $(1 + d)$ covariates, then the structures of the two-dimensional matrix Z and one-dimensional ciphertext CT_Z are as follows, respectively:

$$Z = \begin{bmatrix} z_{10} & z_{11} & \dots & z_{1d} \\ z_{20} & z_{21} & \dots & z_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n0} & z_{n1} & \dots & z_{nd} \end{bmatrix},$$

and

$$CT_Z = (z_{10}, \dots, z_{1d}, z_{20}, \dots, z_{2d}, \dots, z_{n0}, \dots, z_{nd}).$$

We can extract any information we need from this ciphertext CT_Z by multiplying it with a specially-designed ciphertext encrypting some constant vector. For instance, if we want the second row of this matrix Z alone, we can just produce a ciphertext CT_F which encrypts a constant vector F like:

$$F = (0, \dots, 0, 1, \dots, 1, \dots, 0, \dots, 0),$$

and use an element-wise multiplication between CT_F and CT_W , to get the desired data.

We can also aggregate the values of z_{ij} in the same row or the same column. For example, in the calculation of the gradient, we need to calculate the sum of each row in the matrix Z . One simple way is to repeatedly perform the left shifting operation first and then accumulate these intermediate matrixes. We can get the desired result Z_s after repeating these two operations for d times, which can be formulated as:

$$\begin{aligned} Z_s &= Z \oplus Z^{(1)} \oplus \dots \oplus Z^{(d)} \\ &= \begin{bmatrix} z_{10} & z_{11} & \dots & z_{1d} \\ z_{20} & z_{21} & \dots & z_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n0} & z_{n1} & \dots & z_{nd} \end{bmatrix} \oplus \begin{bmatrix} z_{11} & z_{12} & \dots & z_{20} \\ z_{21} & z_{22} & \dots & z_{30} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n1} & z_{n2} & \dots & z_{10} \end{bmatrix} \oplus \dots \oplus \begin{bmatrix} z_{1d} & z_{20} & z_{21} & \dots \\ z_{2d} & z_{30} & z_{31} & \dots \\ \vdots & \vdots & \ddots & \dots \\ z_{nd} & z_{10} & z_{11} & \dots \end{bmatrix} \\ &= \begin{bmatrix} \sum_{j=0}^d z_{1j} & \star & \dots & \star \\ \sum_{j=0}^d z_{2j} & \star & \dots & \star \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{j=0}^d z_{nj} & \star & \dots & \star \end{bmatrix}, \end{aligned}$$

where “ \oplus ” is the element-wise addition and $Z^{(i)}$ denotes the resultant matrix after Z being left rotated by i positions. The output matrix Z_s is composed of the sum of each row in the first column and some “garbage” values in the remaining part of this matrix, denoted by “ \star ”. An efficient way, whose computational cost can be reduced down to $\Theta(\log_2 d)$, is widely adopted in practical programming including ours. Refer [4] for a detailed description of this technique.

We can filter out the meaningless values “ \star ” in Z_s by the operations described above: we design a constant vector F_c consisting of ones in the first column and zeros in the rest columns, and multiply Z_s by F_c , leaving the purified plaintext matrix held by the resulting ciphertext Z_p within its first column.

$$\begin{aligned} Z_p &= F_c \otimes Z_s \\ &= \begin{bmatrix} 1 & 0 & \dots & 0 \\ 1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \dots & 0 \end{bmatrix} \otimes \begin{bmatrix} \sum_{j=0}^d z_{1j} & \star & \dots & \star \\ \sum_{j=0}^d z_{2j} & \star & \dots & \star \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{j=0}^d z_{nj} & \star & \dots & \star \end{bmatrix} \\ &= \begin{bmatrix} \sum_{j=0}^d z_{1j} & 0 & \dots & 0 \\ \sum_{j=0}^d z_{2j} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{j=0}^d z_{nj} & 0 & \dots & 0 \end{bmatrix}. \end{aligned}$$

We can fill up the entire matrix Z_p with its first column: we sum up each row in the matrix Z but with the help of *right rotate* and addition operations:

$$\begin{aligned}
Z_q &= Z_p \oplus Z_p^{[1]} \oplus \dots \oplus Z_p^{[d]} \\
&= \begin{bmatrix} \sum_{j=0}^d z_{1j} & 0 & \dots & 0 \\ \sum_{j=0}^d z_{2j} & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{j=0}^d z_{nj} & 0 & \dots & 0 \end{bmatrix} \oplus \begin{bmatrix} 0 & \sum_{j=0}^d z_{1j} & \dots & 0 \\ 0 & \sum_{j=0}^d z_{2j} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \sum_{j=0}^d z_{nj} & \dots & 0 \end{bmatrix} \oplus \dots \oplus \begin{bmatrix} 0 & 0 & \dots & \sum_{j=0}^d z_{1j} \\ 0 & 0 & \dots & \sum_{j=0}^d z_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sum_{j=0}^d z_{nj} \end{bmatrix} \\
&= \begin{bmatrix} \sum_{j=0}^d z_{1j} & \sum_{j=0}^d z_{1j} & \dots & \sum_{j=0}^d z_{1j} \\ \sum_{j=0}^d z_{2j} & \sum_{j=0}^d z_{2j} & \dots & \sum_{j=0}^d z_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{j=0}^d z_{nj} & \sum_{j=0}^d z_{nj} & \dots & \sum_{j=0}^d z_{nj} \end{bmatrix},
\end{aligned}$$

where “ \oplus ” is still the element-wise addition but $Z^{[i]}$ denotes the resultant matrix after Z being right rotated by i positions.

In a similar way, we can aggregate the values of z_{ij} in the same column of the plaintext matrix. In [9], a procedure named “SumColVec” is to compute the summation of the columns of a matrix, with some other small but useful functions to manipulate the ciphertexts. By dint of these basic operations discussed above, more complex calculations such as computing the gradients in logistic regression models are feasible.

2.3 Logistic Regression

Logistic regression is widely used in binary classification tasks to infer whether a binary-valued variable belongs to a certain class or not. Therefore it is natural to model the distribution of the binary-valued variable into a Bernoulli distribution, which is a probability distribution yielding the probabilities of occurrence of each outcome. LR can be generalized from linear regression [15], which assumes that the response used for regression is a linear function of input $\mathbf{x} \in \mathbb{R}^{(1+d)}$, by mapping the whole real line ($\beta^T \mathbf{x}$) to $(0, 1)$ via the sigmoid function $\sigma(z) = 1/(1 + \exp(-z))$. Thus logistic regression can be formulated as follows:

$$\begin{aligned}
\Pr(y = +1 | \mathbf{x}, \beta) &= \sigma(\beta^T \mathbf{x}) = \frac{1}{1 + e^{-\beta^T \mathbf{x}}}, \\
\Pr(y = -1 | \mathbf{x}, \beta) &= 1 - \sigma(\beta^T \mathbf{x}) = \frac{1}{1 + e^{+\beta^T \mathbf{x}}},
\end{aligned}$$

where the vector $\beta \in \mathbb{R}^{(1+d)}$ is the main parameter of LR, $y \in \{\pm 1\}$ the class label, and the vector $\mathbf{x} = (1, x_1, \dots, x_d) \in \mathbb{R}^{(1+d)}$ the covariate. LR sets a threshold (usually 0.5) and compare its output with this threshold to decide the resulting class label. The logistic regression problem can be transformed into an optimization problem that seeks a parameter β to maximize $L(\beta) = \prod_{i=1}^n \Pr(y_i | \mathbf{x}_i, \beta)$. For convenience in the calculation, log-likelihood function is used as the preferred substitute for generating the LR model. Logistic regression aims to find the parameter vector β to maximize the log-likelihood function $l(\beta)$:

$$l(\beta) = \ln L(\beta) = - \sum_{i=1}^n \ln(1 + e^{-y_i \beta^T \mathbf{x}_i}),$$

where n is the number of examples in the training dataset. LR does not have a closed form of MLE, and thus two main methods are adopted to estimate the parameters of an LR model: (a) gradient descent via the gradient, and (b) Newton’s method by the Hessian matrix. The gradient and Hessian of the log-likelihood function $l(\beta)$ are given by, respectively:

$$\begin{aligned}
\nabla_{\beta} l(\beta) &= \sum_i (1 - \sigma(y_i \beta^T \mathbf{x}_i)) y_i \mathbf{x}_i, \\
\nabla_{\beta}^2 l(\beta) &= \sum_i (y_i \mathbf{x}_i) (\sigma(y_i \beta^T \mathbf{x}_i) - 1) \sigma(y_i \beta^T \mathbf{x}_i) (y_i \mathbf{x}_i) \\
&= X^T S X,
\end{aligned}$$

where S is a diagonal matrix with entries $S_{ii} = (\sigma(y_i \beta^T \mathbf{x}_i) - 1) \sigma(y_i \beta^T \mathbf{x}_i)$.

The log-likelihood function $l(\beta)$ of LR has at most a unique global maximum [16], where its gradient is zero. Newton's method is a second-order technique to numerically find the roots of a real-valued differentiable function, and thus can be used to solve the β in $\nabla_{\beta}l(\beta) = 0$ for LR.

3 Technical Details

It is quite time-consuming to compute the Hessian and its inverse in Newton's method for each iteration. One way to limit this downside is to replace the varying Hessian with a fixed matrix \bar{H} . This novel technique is called the fixed Hessian Newton's method. Böhning and Lindsay [17] have shown that the convergence of Newton's method is guaranteed as long as $\bar{H} \leq \nabla_{\beta}^2 l(\beta)$, where \bar{H} is a symmetric negative-definite matrix independent of β , and " \leq " denotes the Loewner ordering in the sense that the difference $\nabla_{\beta}^2 l(\beta) - \bar{H}$ is non-negative definite. With such a fixed Hessian matrix \bar{H} , the iteration for Newton's method can be simplified to:

$$\beta_{t+1} = \beta_t - \bar{H}^{-1} \nabla_{\beta} l(\beta).$$

Böhning and Lindsay also suggest the fixed matrix $\bar{H} = -\frac{1}{4}X^T X$ is a good lower bound for the Hessian of the log-likelihood function $l(\beta)$ in LR. The $-\frac{1}{4}$ in \bar{H} comes from the inequality: $-\frac{1}{4} \leq S_{kk}$, where S_{kk} is the element of S in $\nabla_{\beta}^2 l(\beta) = X^T S X$.

3.1 the Simplified Fixed Hessian method

Bonte and Vercauteren [3] simplify this lower bound \bar{H} further due to the need for inverting the fixed Hessian in the encrypted domain. They replace the matrix \bar{H} with a diagonal matrix B , whose diagonal elements are simply the sums of each row in \bar{H} , and suggest a specific order of calculation to get B more efficiently. How to invert the simplified fixed Hessian with their calculation order is described in Algorithm 1. We adopt the square brackets " $[]$ " to denote the index of a vector or matrix element in Algorithm 1 and all the algorithms below. For example, for a vector $v \in \mathbb{R}^{(n)}$ and a matrix $M \in \mathbb{R}^{m \times n}$, $v[i]$ means the i -th element of vector v and $M[i][j]$ the j -th element in the i -th row of M , where i and j both start from 1.

Algorithm 1 InvertSFH: inverse of the simplified fixed Hessian

Input: training dataset $X \in \mathbb{R}^{n \times (1+d)}$;

Output: inverse of the simplified fixed Hessian, $\bar{B} \in \mathbb{R}^{(1+d) \times (1+d)}$;

```

1: Set  $rowsum \leftarrow \mathbf{0}$   $\triangleright rowsum \in \mathbb{R}^n$ 
2: for  $i := 1$  to  $n$  do
3:   for  $j := 1$  to  $1 + d$  do
4:      $rowsum[i] \leftarrow rowsum[i] + X[i][j]$ 
5:   end for
6: end for
7: Set  $\tilde{B} \leftarrow \mathbf{0}$   $\triangleright \tilde{B} \in \mathbb{R}^{(1+d) \times (1+d)}$ 
8: Set  $\bar{B} \leftarrow \mathbf{0}$   $\triangleright \bar{B} \in \mathbb{R}^{(1+d) \times (1+d)}$ 
9: for  $j := 1$  to  $1 + d$  do
10:   Set  $temp \leftarrow 0$ 
11:   for  $i := 1$  to  $n$  do
12:      $temp \leftarrow temp + rowsum[i] \cdot X[i][j]$ 
13:   end for
14:    $\tilde{B}[j][j] \leftarrow \epsilon + |-\frac{1}{4} \cdot temp|$ 
15:    $\bar{B}[j][j] \leftarrow 1/\tilde{B}[j][j]$ 
16: end for
17: return  $\bar{B}$ 

```

Their new approximation B of the fixed Hessian is:

$$B = \begin{bmatrix} \sum_{i=0}^d \bar{h}_{0i} & 0 & \dots & 0 \\ 0 & \sum_{i=0}^d \bar{h}_{1i} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sum_{i=0}^d \bar{h}_{di} \end{bmatrix},$$

where \bar{h}_{ki} is the element of \bar{H} . This diagonal matrix B is in a very simple form and can be obtained from \bar{H} without much difficulty. Provided that a diagonal matrix is invertible, its inverse is still a diagonal matrix whose each diagonal element is the reciprocal of its corresponding item in the original matrix. The inverse of B can thus be approximated in the encrypted form by means of computing the inverse of every diagonal element of B via the iterative of Newton's method with an appropriate start value. Their simplified fixed Hessian Newton's method can be formulated as follows:

$$\begin{aligned}\beta_{t+1} &= \beta_t - B^{-1} \cdot \nabla_{\beta} l(\beta), \\ &= \beta_t - \begin{bmatrix} b_{00} & 0 & \dots & 0 \\ 0 & b_{11} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & b_{dd} \end{bmatrix} \cdot \begin{bmatrix} \nabla_0 \\ \nabla_1 \\ \vdots \\ \nabla_d \end{bmatrix}, \\ &= \beta_t - \begin{bmatrix} b_{00} \cdot \nabla_0 \\ b_{11} \cdot \nabla_1 \\ \vdots \\ b_{dd} \cdot \nabla_d \end{bmatrix},\end{aligned}$$

where b_{ii} is the reciprocal of $\sum_{i=0}^d \hat{h}_{0i}$, and ∇_i is the element of $\nabla_{\beta} l(\beta)$.

Consider a special situation: if b_{00}, \dots, b_{dd} all share the same value $-\eta$ with $\eta > 0$, the iterative formula of the SFH method can be given as:

$$\begin{aligned}\beta_{t+1} &= \beta_t - (-\eta) \cdot \begin{bmatrix} \nabla_0 \\ \nabla_1 \\ \vdots \\ \nabla_d \end{bmatrix}, \\ &= \beta_t + \eta \cdot \nabla_{\beta} l(\beta),\end{aligned}$$

which is the same as the formula of the naive gradient *ascent* method. With Gerschgorin's circle theorem [?], we can prove that $\alpha \cdot B$ also suffices to meet the convergence condition of the fixed Hessian method when $\alpha \geq 1$; It is ideal to set a larger learning rate as long as the algorithm converges, which can be guaranteed by the theory of fixed Hessian Newton's method. In other words, we would like a larger $|b_{ii}|$, namely a smaller $|B_{ii}|$, in which case it is when $\alpha = 1$, precisely what Bonte and Vercauteren suggest. From this angle, their simplified fixed Hessian is the optimal simplification of the fixed Hessian matrix. Experimental results show that when α is greater than 1, the speed of convergence begins to slow down; when α is set to less than 1, the algorithm starts to converge faster, and eventually ends in divergence. This phenomenon is also consistent with the setting of the learning rate for gradient (descent) methods. For some datasets, like the genetic dataset in the third track of 2017 iDASH competition, the actual convergence speed will be much faster than (close to twice as) the original SFH algorithm with the same convergence result when the inverse of α (the learning rate) is fixed to 2, even though there is no theoretical guarantee that the algorithm should converge.

As b_{ii} is a floating-point number with very small absolute value in the real-world applications, we treat $(1 + b_{ii}) \cdot \nabla_i$ or $b_{ii} \cdot \nabla_i$ as a correction for each corresponding element of the original gradient. For this reason, we consider $b_{ii} \cdot \nabla_i$ as a new enhanced gradient variant and assign a new learning rate to it. As long as we ensure that this new learning rate decreases from a positive number greater than 1 (such as 2) to 1 in a bounded number of iteration steps, the fixed Hessian Newton's method guarantees the algorithm with this new gradient variant will converge in the long run after the learning rate drops to 1.

In the proof of the simplified fixed Hessian matrix meeting the convergence condition, all entries in the symmetric matrix \bar{H} need to be non-positive. It is a necessary condition that the training dataset consists of all positive numbers, which can be ensured in the 2017 iDASH competition task owing to the positive genetic data. In other cases where the training dataset contains negative numbers, the simplified fixed Hessian method cannot guarantee to work properly. For the fixed Hessian matrix B that Bonte and Vercauteren [3] give, its diagonal elements could be zeros, especially when the dataset is a high-dimensional sparse matrix, which causes this diagonal matrix B to be non-invertible. For the training dataset (and testing dataset) from the MNIST database, there are several columns consisting of all zeros in the database matrix, which leads the corresponding B_{ii} to be zero and thus the SFH method proposed by [3] would fail to work functionally. We complement their work by generalizing this simplified fixed Hessian to be invertible in any case, and propose a faster gradient variant, which we term *quadratic gradient*.

3.2 Quadratic Gradient

Suppose that a differentiable scalar-valued function $F(\mathbf{x})$ has its gradient \mathbf{g} and Hessian matrix H , with any matrix $\bar{H} \leq H$ in the Loewner ordering as follows:

$$\mathbf{g} = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_d \end{bmatrix}, \quad H = \begin{bmatrix} \nabla_{00}^2 & \nabla_{01}^2 & \cdots & \nabla_{0d}^2 \\ \nabla_{10}^2 & \nabla_{11}^2 & \cdots & \nabla_{1d}^2 \\ \vdots & \vdots & \ddots & \vdots \\ \nabla_{d0}^2 & \nabla_{d1}^2 & \cdots & \nabla_{dd}^2 \end{bmatrix}, \quad \bar{H} = \begin{bmatrix} \bar{h}_{00} & \bar{h}_{01} & \cdots & \bar{h}_{0d} \\ \bar{h}_{10} & \bar{h}_{11} & \cdots & \bar{h}_{1d} \\ \vdots & \vdots & \ddots & \vdots \\ \bar{h}_{d0} & \bar{h}_{d1} & \cdots & \bar{h}_{dd} \end{bmatrix},$$

where $\nabla_{ij}^2 = \nabla_{ji}^2 = \frac{\partial^2 F}{\partial x_i \partial x_j}$. We construct a new fixed Hessian matrix \tilde{B} as follows:

$$\tilde{B} = \begin{bmatrix} -\epsilon - \sum_{i=0}^d |\bar{h}_{0i}| & 0 & \cdots & 0 \\ 0 & -\epsilon - \sum_{i=0}^d |\bar{h}_{1i}| & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & -\epsilon - \sum_{i=0}^d |\bar{h}_{di}| \end{bmatrix},$$

where ϵ is a small positive number to avoid division by zero (usually set to $1e-8$).

As long as \tilde{B} satisfies the convergence condition of the above fixed Hessian method, $\tilde{B} \leq H$, we can use this approximation \tilde{B} of the Hessian as a lower bound. Since we already assume that $\bar{H} \leq H$, it will suffice to show that $\tilde{B} \leq \bar{H}$. We prove $\tilde{B} \leq \bar{H}$ in a similar way just as Bonte and Vercauteren [3] did.

Lemma 3.1. *Let $A \in \mathbb{R}^{n \times n}$ be a symmetric matrix, and let B be the diagonal matrix whose diagonal entries $B_{kk} = -\epsilon - \sum_i |A_{ki}|$ for $k = 1, \dots, n$, then $B \leq A$.*

Proof. By definition of the Loewner ordering, we have to prove the difference matrix $C = A - B$ is non-negative definite, which means that all the eigenvalues of C need to be non-negative. By construction of C we have that $C_{ij} = A_{ij} + \epsilon + \sum_{k=1}^n |A_{ik}|$ for $i = j$ and $C_{ij} = A_{ij}$ for $i \neq j$. By means of Gerschgorin's circle theorem, we can bound every eigenvalue λ of C in the sense that $|\lambda - C_{ii}| \leq \sum_{i \neq j} |C_{ij}|$ for some index $i \in \{1, 2, \dots, n\}$. We conclude that $\lambda \geq A_{ii} + \epsilon + |A_{ii}| \geq \epsilon > 0$ for all eigenvalues λ and thus that $B \leq A$. \square

Definition 3.2 (Quadratic Gradient). *Given such a \tilde{B} above, we define the quadratic gradient as $G = \tilde{B} \cdot \mathbf{g}$ with a new learning rate η , where \tilde{B} is a diagonal matrix with diagonal entries $\tilde{B}_{kk} = 1/|\tilde{B}_{kk}|$, and η should be always no less than 1 and decrease to 1 in a limited number of iteration steps. Note that G is still a vector like the gradient \mathbf{g} . To maximize or minimize the function $F(\mathbf{x})$, we can use the iterative formulas: $\mathbf{x}_{k+1} = \mathbf{x}_k + \eta \cdot G$ or $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \cdot G$, just like the naive gradient.*

We point out here that \bar{H} could be the Hessian H itself and \tilde{B} further optimized to: $\tilde{B}_{kk} = \bar{h}_{kk} + |\bar{h}_{kk}| + \epsilon - \sum_{i=0}^d |\bar{h}_{ki}|$. We don't adopt this form of \tilde{B} due to its difficult implementation in the encrypted domain. In our experiments, we use $\bar{H} = -\frac{1}{4}X^T X$ to construct our \tilde{B} , and design the learning rate function with respect to the t -th step for the SFH method, $f_\eta(t, \kappa) = 2 - \sigma(20(t-1)/(\kappa-1) - 10)$, where κ is the preset maximum number of iterations. The function $f_\eta(t, \kappa)$ decreases roughly from 2 to 1 as t gradually increases from 1 to κ . Our simplified fixed Hessian method with quadratic gradient is described in Algorithm 2.

3.3 Various Applications

Quadratic Gradient can be used to enhance NAG, Adagrad, and Stochastic gradient descent (SGD).

NAG is a different variant of the momentum method to give the momentum term much more prescience. The iterative formulas of the gradient *ascent* method for NAG are as follows:

$$V_{t+1} = \beta_t + \alpha_t \cdot \nabla J(\beta_t), \quad (3)$$

$$\beta_{t+1} = (1 - \gamma_t) \cdot V_{t+1} + \gamma_t \cdot V_t, \quad (4)$$

where V_{t+1} is the intermediate variable used for updating the final weight β_{t+1} , and $\gamma_t \in (0, 1)$ is a smoothing parameter of moving average to evaluate the gradient at an approximate future position [4]. NAG with quadratic gradient is to replace (3) with $V_{t+1} = \beta_t + (1 + \alpha_t) \cdot \tilde{B} \cdot \nabla J(\beta_t)$. Our enhanced NAG method with quadratic gradient is described in Algorithm 3.

Algorithm 2 Simplified fixed Hessian with quadratic gradient

Input: training dataset $X \in \mathbb{R}^{n \times (1+d)}$; training label $Y \in \mathbb{R}^{n \times 1}$; and the preset number of iterations, κ ;

Output: the parameter vector $\beta \in \mathbb{R}^{(1+d)}$

```
1: Set  $\beta \leftarrow \mathbf{0}$  ▷ Initialize the weight vector
2: Set  $\bar{B} \leftarrow \text{InvertSFH}(X)$  ▷  $\bar{B} \in \mathbb{R}^{(1+d) \times (1+d)}$ 
   ▷ The Iterative Procedure of Gradient Descent Method
3: Set  $count \leftarrow 0$ 
4: while  $count < \kappa$  do
5:   Set  $Z \leftarrow \mathbf{0}$  ▷  $Z \in \mathbb{R}^n$  is the inputs for sigmoid function
6:   for  $i := 1$  to  $n$  do
7:     for  $j := 0$  to  $d$  do
8:        $Z[i] \leftarrow Z[i] + Y[i] \times \beta[j] \times X[i][j]$ 
9:     end for
10:  end for
   ▷ To compute the value of the sigmoid function for each input  $Z_i$ 
11:  Set  $\sigma \leftarrow \mathbf{0}$  ▷  $\sigma \in \mathbb{R}^n$  is to store the outputs of the sigmoid function
12:  for  $i := 1$  to  $n$  do
13:     $\sigma[i] \leftarrow 1/(1 + \exp(-Z[i]))$ 
14:  end for
   ▷ To calculate the gradient  $g \in \mathbb{R}^{(1+d)}$ 
15:  Set  $g \leftarrow \mathbf{0}$ 
16:  for  $j := 0$  to  $d$  do
17:    for  $i := 1$  to  $n$  do
18:       $g[j] \leftarrow g[j] + (1 - \sigma[i]) \times Y[i] \times X[i][j]$ 
19:    end for
20:  end for
   ▷ To calculate the quadratic gradient  $G \in \mathbb{R}^{(1+d)}$ 
21:  Set  $G \leftarrow \mathbf{0}$ 
22:  for  $j := 0$  to  $d$  do
23:     $G[j] \leftarrow \bar{B}[j][j] \times g[j]$ 
24:  end for
   ▷ To update the weight vector  $\beta$  with the new learning rate  $\eta \in [1, 2]$ 
25:  Set  $\eta \leftarrow f_\eta(1 + count, \kappa)$ 
26:  for  $j := 0$  to  $d$  do
27:     $\beta[j] \leftarrow \beta[j] + \eta \times G[j]$ 
28:  end for
29: end while
30: return  $\beta$ 
```

Adagrad is a gradient-based algorithm suitable for dealing with sparse data. The updated operations of Adagrad and its quadratic-gradient version, for every parameter β_i at each iteration step t , are as follow, respectively:

$$\beta_i^{(t+1)} = \beta_i^{(t)} - \eta / \sqrt{\epsilon + \sum_{k=1}^t g_i^{(k)} \cdot g_i^{(k)}},$$
$$\beta_i^{(t+1)} = \beta_i^{(t)} - (1 + \eta) / \sqrt{\epsilon + \sum_{k=1}^t G_i^{(k)} \cdot G_i^{(k)}}.$$

Stochastic gradient descent (SGD) with quadratic gradient can also be obtained by replacing the gradient g with the quadratic gradient G and increasing the learning rate by 1.

Adagrad's method is not a practical solution for homomorphic LR due to its frequent inversion operations. SGD method does not make full use of the maximum capacity and parallel operations of the packed ciphertexts. It seems plausible that NAG is the best choice for privacy-preserving LR training.

Algorithm 3 Nesterov's accelerated gradient with quadratic gradient

Input: training dataset $X \in \mathbb{R}^{n \times (1+d)}$; training label $Y \in \mathbb{R}^{n \times 1}$; and the preset number of iterations, κ ;

Output: the parameter vector $V \in \mathbb{R}^{(1+d)}$

```

1: Set  $\bar{B} \leftarrow \text{InvertSFH}(X)$   $\bar{B} \in \mathbb{R}^{(1+d) \times (1+d)}$ 
2: Set  $V \leftarrow \mathbf{0}$   $\triangleright$  Initialize the weight vector  $\mathbf{v} \in \mathbb{R}^{(1+d)}$ 
3: Set  $W \leftarrow \mathbf{0}$   $\triangleright$  Initialize the vector  $\mathbf{w} \in \mathbb{R}^{(1+d)}$ 
4: Set  $\alpha_0 \leftarrow 0.01$ 
5: Set  $\alpha_1 \leftarrow 0.5 \times (1 + \sqrt{1 + 4 \times \alpha_0^2})$ 
    $\triangleright$  The Iterative Procedure of Gradient Descent Method
6: for  $count := 1$  to  $\kappa$  do
7:   Set  $Z \leftarrow \mathbf{0}$   $\triangleright Z \in \mathbb{R}^n$  is the inputs for sigmoid function
8:   for  $i := 1$  to  $n$  do
9:     for  $j := 0$  to  $d$  do
10:       $Z[i] \leftarrow Z[i] + Y[i] \times V[j] \times X[i][j]$ 
11:    end for
12:  end for
    $\triangleright$  To compute the value of the sigmoid function for each input  $Z_i$ 
13:  Set  $\sigma \leftarrow \mathbf{0}$   $\triangleright \sigma \in \mathbb{R}^n$  is to store the outputs of the sigmoid function
14:  for  $i := 1$  to  $n$  do
15:     $\sigma[i] \leftarrow 1 / (1 + \exp(-Z[i]))$ 
16:  end for
    $\triangleright$  To calculate the gradient  $\mathbf{g} \in \mathbb{R}^{(1+d)}$ 
17:  Set  $\mathbf{g} \leftarrow \mathbf{0}$ 
18:  for  $j := 0$  to  $d$  do
19:    for  $i := 1$  to  $n$  do
20:       $\mathbf{g}[j] \leftarrow \mathbf{g}[j] + (1 - \sigma[i]) \times Y[i] \times X[i][j]$ 
21:    end for
22:  end for
    $\triangleright$  To calculate the quadratic gradient  $G \in \mathbb{R}^{(1+d)}$ 
23:  Set  $G \leftarrow \mathbf{0}$ 
24:  for  $j := 0$  to  $d$  do
25:     $G[j] \leftarrow \bar{B}[j][j] \times \mathbf{g}[j]$ 
26:  end for
    $\triangleright$  To update the weight vector  $V$ 
27:  Set  $\eta \leftarrow (1 - \alpha_0) / \alpha_1$ 
28:  Set  $\gamma \leftarrow 1 / (n \times count)$ 
    $\triangleright n$  is the size of training data
29:  for  $j := 0$  to  $d$  do
30:     $w_{temp} \leftarrow V[j] + (1 + \gamma) \times G[j]$ 
31:     $V[j] \leftarrow (1 - \eta) \times w_{temp} + \eta \times W[j]$ 
32:     $W[j] \leftarrow w_{temp}$ 
33:  end for
34:   $\alpha_0 \leftarrow \alpha_1$ 
35:   $\alpha_1 \leftarrow 0.5 \times (1 + \sqrt{1 + 4 \times \alpha_0^2})$ 
36: end for
37: return  $V$ 

```

For a large training dataset, the full batch gradient descent method using all the data requires too many ciphertexts and HE operations in every iteration, which is neither efficient nor necessary. In this situation, mini-batch gradient (descent) method should be considered. Our enhanced mini-batch NAG method is described in Algorithm 4.

Algorithm 4 Enhanced mini-batch NAG

Input: training-data mini-batches $\{X_i | X_i \in \mathbb{R}^{n \times (1+d)}, i \in [1, m]\}$ where n is the mini-batch size and m the number of mini-batches. Namely, it will take m iterations (passes) to complete one epoch; training-label mini-batches, $\{Y_i | Y_i \in \mathbb{R}^n, i \in [1, m]\}$, for each corresponding training-data mini-batches X_i ; the number of epochs, N_e ; and the preset number of iterations, κ ;

Output: the parameter vector $V \in \mathbb{R}^{(1+d)}$

```

1: for  $i := 1$  to  $m$  do
2:   Set  $\bar{B}_i \leftarrow \text{InvertSFH}(X_i)$   $\triangleright \bar{B}_i \in \mathbb{R}^{(1+d) \times (1+d)}$ 
3: end for
4: Set  $V \leftarrow \mathbf{0}$   $\triangleright$  Initialize the weight vector  $\mathbf{v} \in \mathbb{R}^{(1+d)}$ 
5: Set  $W \leftarrow \mathbf{0}$   $\triangleright$  Initialize the vector  $\mathbf{w} \in \mathbb{R}^{(1+d)}$ 
6: Set  $\alpha_0 \leftarrow 0.01$ 
7: Set  $\alpha_1 \leftarrow 0.5 \times (1 + \sqrt{1 + 4 \times \alpha_0^2})$ 
8: for  $epoch := 0$  to  $N_e - 1$  do
9:   Shuffle the mini-batches  $\{X_i\}$ 
10:  for  $k := 1$  to  $m$  do  $\triangleright Z \in \mathbb{R}^n$  is the inputs for sigmoid function
11:    Set  $Z \leftarrow \mathbf{0}$ 
12:    for  $i := 1$  to  $n$  do
13:      for  $j := 0$  to  $d$  do
14:         $Z[i] \leftarrow Z[i] + Y_k[i] \times V[j] \times X_k[i][j]$   $\triangleright X_k[i][j]$ 
15:      end for
16:    end for
17:    Set  $\sigma \leftarrow \mathbf{0}$   $\triangleright \sigma \in \mathbb{R}^n$ 
18:    for  $i := 1$  to  $n$  do
19:       $\sigma[i] \leftarrow 1 / (1 + \exp(-Z[i]))$ 
20:    end for
21:    Set  $g \leftarrow \mathbf{0}$ 
22:    for  $j := 0$  to  $d$  do
23:      for  $i := 1$  to  $n$  do
24:         $g[j] \leftarrow g[j] + (1 - \sigma[i]) \times Y_k[i] \times X_k[i][j]$ 
25:      end for
26:    end for
27:    Set  $G \leftarrow \mathbf{0}$ 
28:    for  $j := 0$  to  $d$  do
29:       $G[j] \leftarrow \bar{B}[j][j] \times g[j]$ 
30:    end for
31:    Set  $\eta \leftarrow (1 - \alpha_0) / \alpha_1$ 
32:    Set  $\gamma \leftarrow 1 / (1 + epoch + k) \times 1/n$   $\triangleright n$  is the size of training data
33:    for  $j := 0$  to  $d$  do
34:       $w_{temp} \leftarrow V[j] + (1 + \gamma) \times G[j]$ 
35:       $V[j] \leftarrow (1 - \eta) \times w_{temp} + \eta \times W[j]$ 
36:       $W[j] \leftarrow w_{temp}$ 
37:    end for
38:    if  $epoch \times N_e + k \geq \kappa$  then
39:      return  $V$ 
40:    end if
41:     $\alpha_0 \leftarrow \alpha_1$ 
42:     $\alpha_1 \leftarrow 0.5 \times (1 + \sqrt{1 + 4 \times \alpha_0^2})$ 
43:  end for
44: end for
45: return  $V$ 

```

4 Privacy-preserving Logistic Regression Training

4.1 Polynomial Approximation of Activation Function

Privacy-preserving logistic regression training via homomorphic encryption technique faces a difficult dilemma that no homomorphic schemes are capable of directly calculating the sigmoid function used in the LR model. A common solution is to replace the sigmoid function with a polynomial approximation, which can be calculated by any homomorphic encryption scheme. There exist several approaches to find a polynomial approximation, such as Chebyshev approximation, minimax approximation, the Remez algorithm, and the least-squares method. Our scheme is based on the last one, the least-squares method, which is also used by Kim et al. [4]. A function named “polyfit(·)” both in the numerical computing software Octave and the Python package Numpy is to fit the polynomial in a least-squares sense. To approximate the sigmoid function over the domain $[-8, 8]$ by a polynomial of degree 7, we only need 3 lines of Octave code: `x = [-8:0.1:+8]; y = 1./(1+exp(-x)); polyfit(x,y,7)` and can obtain the polynomial approximation of the sigmoid function: $g_7(x) = 0.5 + 0.21644 \cdot x + 5.6689e - 17 \cdot x^2 - 0.0081162 \cdot x^3 - 3.3037e - 18 \cdot x^4 + 0.00016276 \cdot x^5 + 4.4880e - 20 \cdot x^6 - 0.0000011612 \cdot x^7$. It is impossible to adopt the polynomial approximation to better fit an activation function over the whole real axis or even a moderate large interval in the encrypted domain. A promising way to crack this problem, probably the ultimate way for fitting the sigmoid function, is to limit the input of the sigmoid function to a narrow range. For an input z of the sigmoid function $\sigma(\cdot)$ with $|z| > 8$, $\sigma(z) \approx \sigma(z/|z| \cdot 8)$. We can just replace $\sigma(z)$ with $\sigma(\max(-8, \min(z, 8)))$ by means of two ciphertext comparison operations [18, 19] and then calculate it by using a low degree polynomial fitting the $\sigma(\cdot)$ just over the domain $[-8, 8]$. This solution is not practical due to the inefficiency of ciphertext comparison operations for now, which is partly why we don’t adopt this method.

4.2 Applying the Quadratic Gradient

The difficulty in applying the quadratic gradient to privacy-preserving LR training is to invert the diagonal matrix \tilde{B} . Our gradient variant extends the SFH matrix and thus can still be applied in the field of homomorphic encryption. We use Newton’s method again as Bonte and Vercauteren [3] did to approximate the inverse of each diagonal element $|\tilde{B}_{ii}|$, which need a good start value x_0 for the method to converge. Using Newton’s method to approximate $x = \frac{1}{|\tilde{B}_{ii}|}$, We get the equation: $\frac{1}{|\tilde{B}_{ii}|} - x = 0$, and the iterative formula: $x_{k+1} = x_k \cdot (2 - |\tilde{B}_{ii}| \cdot x_k)$, which can be further transformed to:

$$x_{k+1} - 1/|\tilde{B}_{ii}| = 1/|\tilde{B}_{ii}| \cdot (1 - |\tilde{B}_{ii}| \cdot x_k)^{2^{k+1}}, \quad (5)$$

where x_0 is the start value and k is the number of iteration steps. As long as $0 < x_0 < 2/|\tilde{B}_{ii}|$, the convergence of the Newton’s method in this case can be guaranteed. Since \tilde{B}_{ii} is derived from $\tilde{H} = -\frac{1}{4}X^T X$, the first column of X consists of ones, and the rest columns of X have been normalized into the range $[0, 1]$, we can easily calculate out that $|\tilde{B}_{00}|$ is the biggest among the $|\tilde{B}_{ii}|$ s and safely set $x_0 = 2/|\tilde{B}_{00}| - \epsilon$. We perform nine iterations of the method to get a pretty decent approximation of the inverse.

4.3 The Whole Pipeline

Our approach can be deployed in several practical scenarios, which could be the model in [9] or [2]. Without paying attention to the usage model, we assume that a third-party authority, which even could be the client itself, is needed to build a homomorphic encryption system. This third party issues secret key, public key, encryption algorithm, and decryption algorithm to the client. Another duty of this third party is to deploy the main HE system including all public keys to the cloud, as well as the privacy-preserving LR training algorithm. For the sake of simplicity, we adopt the assumptions that the entire dataset can be encrypted into a single ciphertext, and that the training data have been normalized into the range $(0, 1)$ and the training labels have been transferred to $\{-1, +1\}$, if not. Based on these assumptions, the whole pipeline of our enhanced gradient algorithms for privacy-preserving logistic regression training is fully illustrated in Figure 1, which consists of the following ten steps:

Step 1. The client need to prepare only four data matrices: the matrix \mathbf{x}_0 consisting of all $(x_0 = 2/|\tilde{B}_{00}| - \epsilon)$ that is used as the start value for Newton’s method to invert the matrix B ; the matrix X that is the training data inherently of two-dimensional structure; the matrix Y consisting of copies of the corresponding train label y_i in each row; the matrix W each row of which is filled with the initial (random) weight vector with every positive element close to zero. We recommend adopting the zero vector as the initial weight vector rather than the random vector. We adopt the zero vector as the weight vector in our programming.

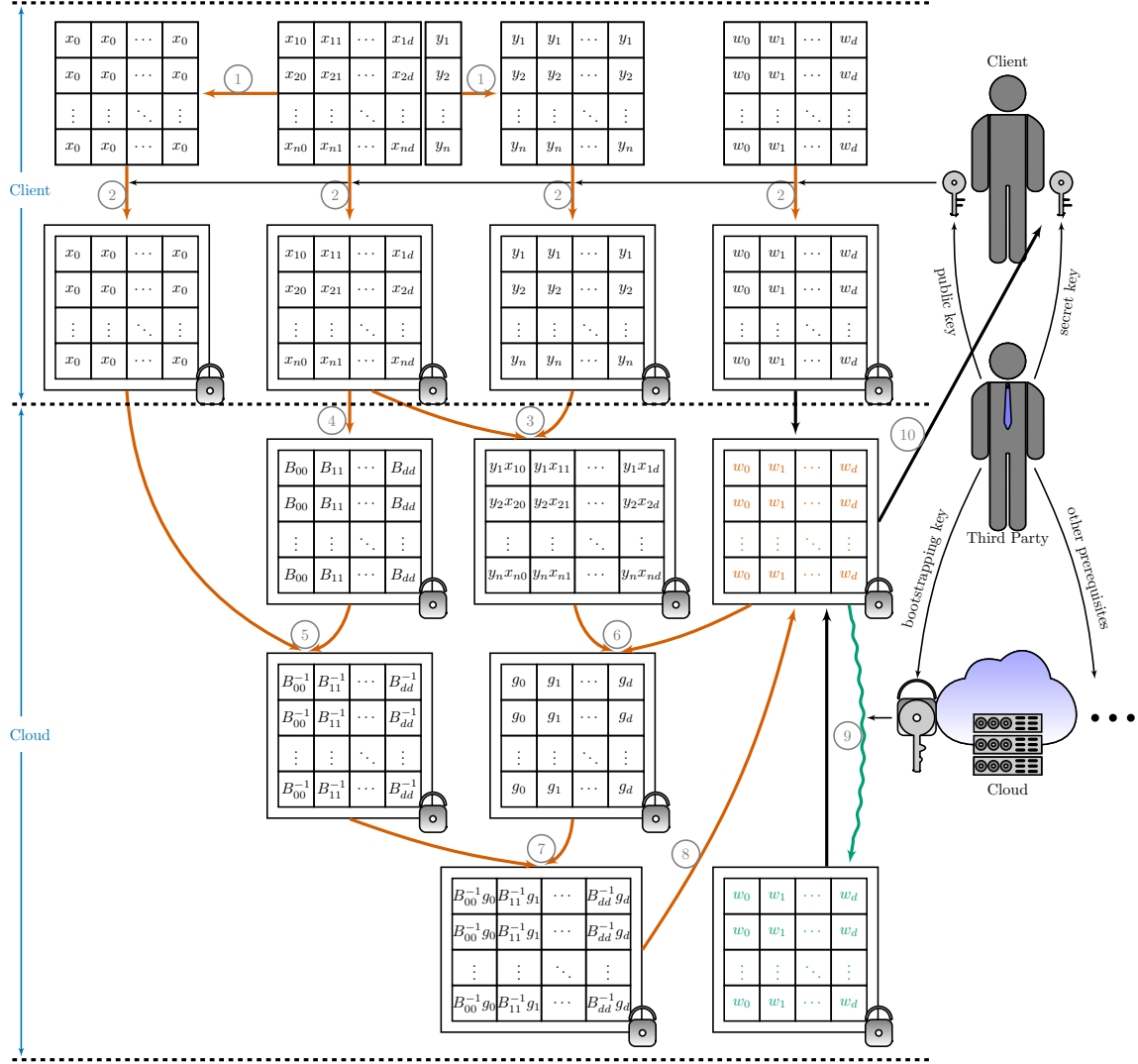


Figure 1: the whole pipeline of logistic regression training via homomorphic encryption

Step 2. The client then encrypts the four matrices, x_0, X, Y, W , into ciphertexts, $ct_{x_0}, ct_X, ct_Y, ct_W$, respectively, using the public key. After enciphering these four data matrices, the client sends the resulting ciphertexts to the public cloud.

So far, the client has finished its job and only needs to wait for the cloud to send back the ciphertexts encrypting the weight vector of the well-trained LR model.

Step 3. After receiving the ciphertexts uploaded by the client, the (public) cloud begins its work: at first, the cloud performs a SIMD multiplication between ct_X and ct_Y to get the ciphertext ct_Z .

Step 4. The cloud follows the specific calculating order in Algorithm 1 to produce the ciphertext ct_B encrypting a matrix consisting of all the diagonal elements of B in its corresponding each row, from the ciphertext ct_X via the basic operations introduced in the subsection 2.2.

In the rest of the training process, the cloud no longer needs the ciphertext ct_X and ct_Y .

Step 5. The cloud carries out operations on both the received ciphertexts ct_{x_0} and the resulting ciphertext ct_B , so as to perform the iterative formula (5) for nine times, to get the ciphertext $ct_{B^{-1}}$ encrypting the approximation of the inverse of B .

Step 6. The cloud calculates out the naive gradient ciphertext ct_g by homomorphic encryption operations on the ciphertext ct_Z and the weight ciphertext ct_W . The detailed calculation process composed of seven steps can be found in [4].

Step 7. The cloud produces the quadratic gradient by one SIMD multiplication between the ciphertext $ct_{B^{-1}}$ and ct_g , resulting in ciphertext ct_G .

Step 8. The cloud uses different learning-rate setting strategy for different algorithms, to update the weight ciphertext ct_W using the quadratic-gradient ciphertext ct_G .

Step 9. At the end of each iteration, the cloud checks out whether or not the remainder modulus of the weight ciphertext ct_W is large enough to complete the next round of updating the ciphertext ct_W . If not, the cloud then would bootstrap only these weight ciphertexts using rotation keys, rendering the weight ciphertexts with a large modulus.

Step 10. The cloud sends the weight ciphertext to the client if the algorithm reaches the preset maximum number of iterations; otherwise, the training process goes to *Step 6.* and continues running.

Now, the client could decipher the received ciphertext using the secret key and has the LR model for its only use.

5 Experimental results

5.1 Datasets

Two real-world datasets are used in our experiments, one in genomics and the other in image processing. The third task in the iDASH competition of 2017 provides the genomic dataset, which consists of 1579 records. Each record in it has 103 binary genotypes and a binary phenotype indicating if the patient has cancer. The image data is provided by the MNIST database, which contains a training dataset consisting of 60000 images and a testing dataset that has 10000 images. Each original image in the database is a 28×28 pixel bounding box and each pixel is expressed with a 256-level gray code. We pick out all the images containing handwritten digits ‘3’ and ‘8’, and compress them from 28×28 features into $14 \times 14 = 196$ features. The restructured training dataset consists of 11982 samples and the new generation of validation dataset has 1984 samples.

5.2 Experiments on plaintexts

We first evaluate the performance of various algorithms in the plaintext (naive) environment. All the experiments under this circumstance were done on the same desktop computer with an Intel Pentium CPU G640 at 2.80 GHz and 4 GB RAM. Maximum likelihood estimation (MLE) and area under the curve (AUC) are selected as the main indicators of how well the algorithm works. We evaluate four algorithms, SFH, NAG, and their quadratic-gradient versions (denoted as SFH + G, NAG + G) on the gene data, using 5-fold cross validation. For the MNIST database, we randomly select 1579 images from the training dataset as training data but the whole testing dataset as validation data. We repeat this selection process five times and evaluate these four algorithms on average, for better estimating the algorithm performance. Figure 2 shows the comparison of MLE and AUC in the training phase and testing phase respectively, among the four classifiers. The result of this experiment demonstrates that the quadratic-gradient versions outperform the originals significantly and SFH outspeeds NAG. We also compare the performance of Adagrad and SGD versus their quadratic-gradient versions (denoted as Adagrad + G, SGD + G) on the whole MNIST test data, especially for comparing the convergence rate in the training phases. We adopt the specific algorithm [20] to implement SGD. Figure 3 shows the comparison of MLE between Adagrad, SGD and their quadratic-gradient versions, and the fact that their quadratic-gradient versions perform better than the originals by a large margin. In all the Python experiments, the time to calculate the \bar{B} in quadratic gradient G before running the iterations, and the time to run each iteration for various algorithms are negligible (few seconds).

From the experimental result, we conclude: (a) the enhanced gradient methods have better performance than the original ones in not only the convergence rate but (sometimes) also the convergence result, without compromising the efficiency; (b) NAG with quadratic gradient is probably the best candidate for privacy-preserving LR training, compared to all other methods.

As we mentioned before, a mini-batch gradient (descent) method is necessary for a large dataset. We compare the performance (MLE and AUC) between the mini-batch NAG method of naive gradient and that of quadratic gradient. Figure 4 shows the comparison results on the MNIST test dataset that the quadratic gradient accelerates the algorithm in the training phase, even for the mini-batch gradient methods.

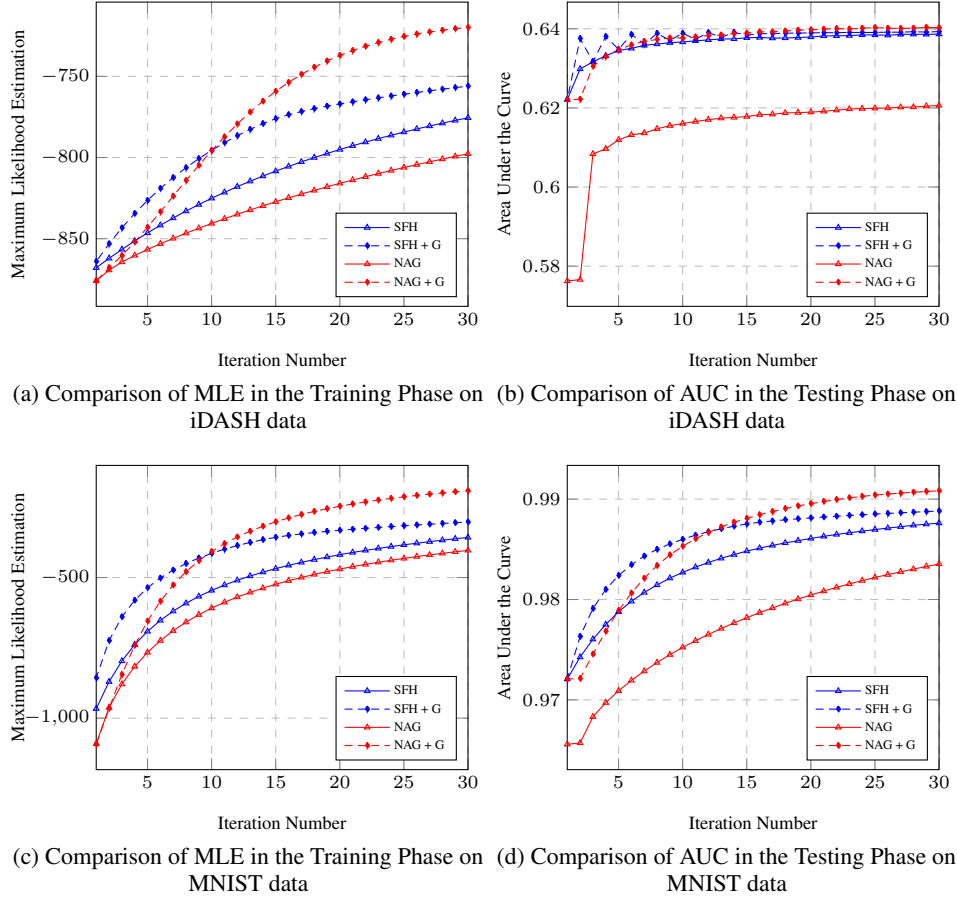


Figure 2: Results in the Plaintext (naive) Environment for four Methods

5.3 Experiments on ciphertexts

We observe that the input range of the sigmoid function tends to get wider and wider as the iterative process proceeds. In Figure 5, it shows the input ranges of enhanced full batch NAG and enhanced mini-batch NAG are getting larger along with the increment of iterations of the training process on the whole MNIST training dataset, which helps us decide to use various polynomial approximations the interval of which should be how large at different stages of the training procedure.

For the quadratic gradient method of full batch NAG, we use the approximate polynomials $g_3(x)$ in the first 5 iterations, $g_5(x)$ in the next 5 iterations, and $\bar{g}_7(x)$ in the last 20 iterations as follows:

$$\begin{cases} g_3(x) &= 0.5 + 0.15012 \cdot x - 0.00159300781 \cdot x^3, & \text{for } |x| < 8 \\ g_5(x) &= 0.5 + 0.19131 \cdot x - 0.0045963 \cdot x^3 + 0.0000412332 \cdot x^5, & \text{for } |x| < 8 \\ g_7(x) &= 0.5 + 0.14444 \cdot x - 0.0024198 \cdot x^3 + 0.00002172914 \cdot x^5 - 0.00000006949 \cdot x^7, & \text{for } |x| < 8 \\ \bar{g}_7(x) &= 0.5 + 0.10844 \cdot x - 0.0010239 \cdot x^3 + 0.00000518229 \cdot x^5 - 0.00000000934 \cdot x^7, & \text{for } |x| < 16 \end{cases}$$

where $g_3(x)$, $g_5(x)$, and $g_7(x)$ are all the approximate polynomials used by Kim et al. [4]. We reshape and stretch $g_7(x)$ along the X-axis to cover the domain $[-16, 16]$, resulting in $\bar{g}_7(x) = g_7(8/16 \cdot x)$.

For the quadratic gradient method of mini-batch NAG, we use the approximate polynomials $e_1(x)$ in the first epoch, $e_2(x)$ in the second epoch, and $\bar{e}_3(x)$ in the last epoch as follows:

$$\begin{cases} e_1(x) &= 0.5 + 0.23514 \cdot x - 0.012329 \cdot x^3 + 0.00039345 \cdot x^5 - 0.0000047292 \cdot x^7, & \text{for } |x| < 6 \\ f_1(x) &= 0.5 + 0.15512 \cdot x - 0.0028088 \cdot x^3 + 0.000026158 \cdot x^5 - 0.000000085324 \cdot x^7, & \text{for } |x| < 9 \\ e_2(x) &= 0.5 + 0.14465 \cdot x - 0.0024308 \cdot x^3 + 0.00002189 \cdot x^5 - 0.00000007022 \cdot x^7, & \text{for } |x| < 12 \\ e_3(x) &= 0.5 + 0.11634 \cdot x - 0.0011850 \cdot x^3 + 0.0000062074 \cdot x^5 - 0.000000011389 \cdot x^7, & \text{for } |x| < 16 \end{cases}$$

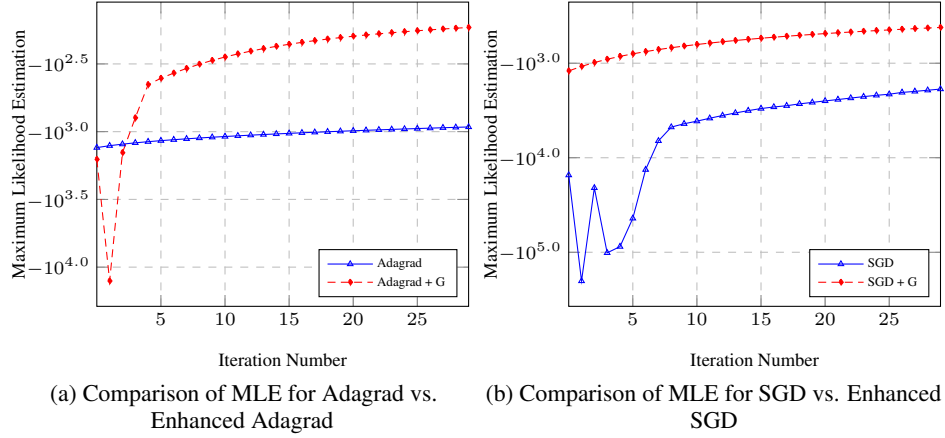


Figure 3: Results in the Training Phase for Adagrad, SGD vs. their Enhanced Methods

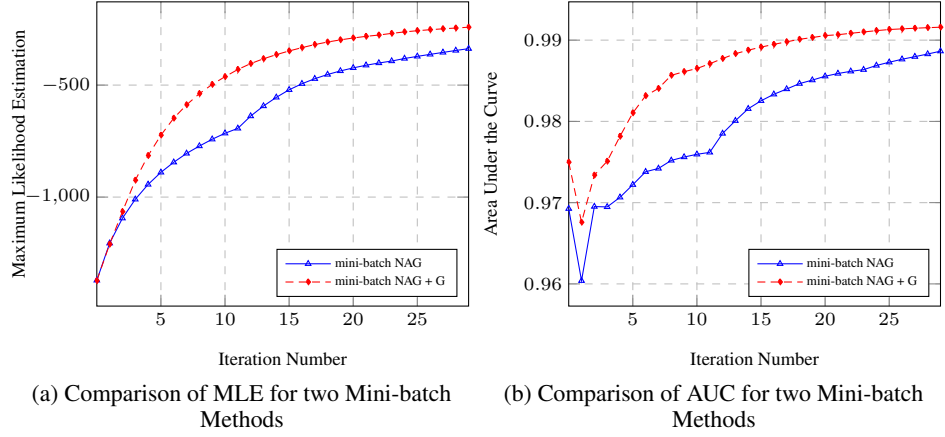
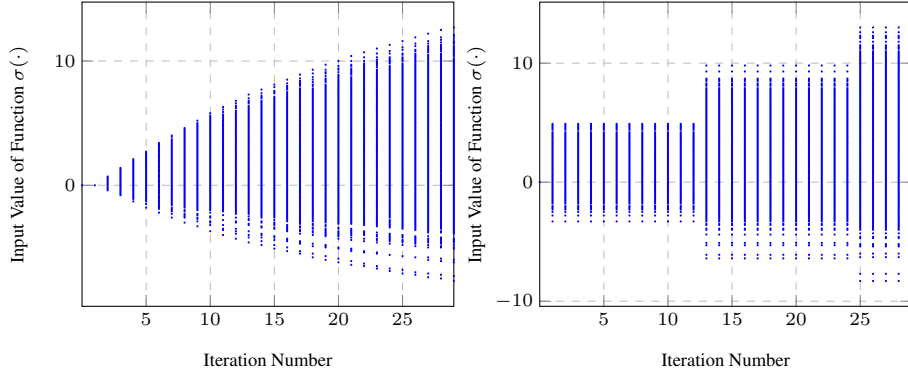


Figure 4: Results in the Training Phase: Mini-batch NAG vs. Enhanced Mini-batch NAG

where $e_1(x)$, $f_1(x)$ are both computed by the least-squares method using Octave, fitting the domain $[-6, +6]$, $[-9, +9]$ respectively and $e_2(x)$, $e_3(x)$ are obtained from the mapping: $e_2(x) = f_1(9/12 \cdot x)$, $e_3(x) = f_1(9/16 \cdot x)$, in order to get two polynomials better fitting the domains $[-12, +12]$ and $[-16, +16]$.

We implement the quadratic-gradient versions of full batch NAG and mini-batch NAG based on FHE, using the HEAAN library that implements a `rescaling` operation to support approximate arithmetic and approximate bootstrapping. The datasets we used and our C++ source codes are publicly available at <https://github.com/petitioner/HE.LR>.

The parameters of HEAAN library we set are: $\log N = 16$, $\log Q = 1200$, $\log p = 30$, $\text{slots} = 32768$, which ensure the security level $\lambda = 80$. Refer [4] for the details of these and other needed parameters. All the experiments on the ciphertexts were carried out on a server with an Intel Xeon CPU E5-2650 v4 at 2.20 GHz and 512 GB RAM. We run the two quadratic-gradient *ascent* methods (full batch NAG and mini-batch NAG) both on the first four-fifths of the gene dataset and the whole MNIST training dataset. The last fifth part of the gene dataset and the whole MNIST testing dataset are used to respectively check the generalization ability of the resulting LR models. Table 1 gives a summary of all the experimental results. The enhanced full batch NAG and the enhanced mini-batch NAG share a similar performance, except that the running time for each iteration is different on the datasets of different sizes. The too abundant data information does not augment the performance of the full batch NAG algorithm much, but instead increases the calculation time for each iteration. The number of iterations for various experiments is determined mainly based on the result of MLE on validation data.



(a) Input Range for Enhanced Full batch NAG (b) Input Range for Enhanced Mini-batch NAG

Figure 5: Input Range in the Training Phase

Table 1: Implementation Results for two Enhanced Classifiers

Method	(Full batch) NAG + G		(Mini-batch) NAG + G	
Dataset	MNIST	iDASH	MNIST	iDASH
No. Samples (training)	11982	1264	11982	1264
No. Samples (validation)	1984	315	1984	315
No. Features	196	103	196	103
No. Iterations	27	19	27	19
Block Size (mini-batch)	-	-	1024	1024
Encryption Time	71s	5s	60s	6s
Running Time	27.95h	117.88min	3.52h	14.49min
MLE	-237.271	-196.279	-233.19	-195.187
AUC	0.990	0.67	0.991	0.69

6 Conclusion

In this paper, we proposed a faster gradient variant called `quadratic gradient`, and implemented the quadratic-gradient versions of full batch NAG and mini-batch NAG in the encrypted domain to train the logistic regression model. The enhanced mini-batch NAG is a better method for privacy-preserving logistic regression training on a small or large dataset, compared with the enhanced full batch one.

The quadratic gradient presented in this work can be constructed from the Hessian matrix directly, and thus somehow integrates the second-order Newton’s method and the first-order gradient (descent) method together. Some gradient algorithms equipped with quadratic gradient show a better performance. There is a good chance that this quadratic gradient can boost other gradient methods like RMSprop, which is an open future work.

John Chiang, whose name was Li-Yue Sun (孫 黎月, 1988-11-19 -), would like to give special thanks to his mother Kui-Hua Chiang (姜 桂華, 1957-11-06 -) who is the main reason that he could weather the storms of life so far and to other family members that support him all the time.

Acknowledgments

The basic work of this paper, `quadratic gradient` and the enhanced full batch NAG, was nearly finished in September 2019. The initial version of this paper was written in April 2020, rejected by ICANN 2020.

The enhanced mini-batch NAG was introduced into this paper in September 2020 and later rejected by a special issue on the journal FGCS 2020.

We apologize for the ill-designed experiments due to the fact that too much research work has been compressed into this paper, which should be split into two papers. We also apologize for the English writing in this paper: Chiang just lost all the patience for this paper.

References

- [1] Yoshinori Aono, Takuya Hayashi, Le Trieu Phong, and Lihua Wang. Scalable and secure logistic regression via homomorphic encryption. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 142–144, 2016.
- [2] 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.
- [3] Charlotte Bonte and Frederik Vercauteren. Privacy-preserving logistic regression training. *BMC medical genomics*, 11(4):86, 2018.
- [4] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. Logistic regression model training based on the approximate homomorphic encryption. *BMC medical genomics*, 11(4):83, 2018.
- [5] Hao Chen, Ran Gilad-Bachrach, Kyoohyung Han, Zhicong Huang, Amir Jalali, Kim Laine, and Kristin Lauter. Logistic regression over encrypted data from fully homomorphic encryption.
- [6] Jack LH Crawford, Craig Gentry, Shai Halevi, Daniel Platt, and Victor Shoup. Doing real work with fhe: the case of logistic regression. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 1–12, 2018.
- [7] Miran Kim, Yongsoo Song, Baiyu Li, and Daniele Micciancio. Semi-parallel logistic regression for gwas on encrypted data. *IACR Cryptology ePrint Archive*, 2019:294, 2019.
- [8] Marcelo Blatt, Alexander Gusev, Yuriy Polyakov, Kurt Rohloff, and Vinod Vaikuntanathan. Optimized homomorphic encryption solution for secure genome-wide association studies. *IACR Cryptology ePrint Archive*, 2019:223, 2019.
- [9] Kyoohyung Han, Seungwan Hong, Jung Hee Cheon, and Daejun Park. Logistic regression on homomorphic encrypted data at scale. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9466–9471, 2019.
- [10] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.
- [11] Angela Jäschke and Frederik Armknecht. Accelerating homomorphic computations on rational numbers. In *International Conference on Applied Cryptography and Network Security*, pages 405–423. Springer, 2016.
- [12] 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.
- [13] 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.
- [14] M. Artin. *Algebra*. Pearson Prentice Hall, 2011.
- [15] Kevin P Murphy. *Machine learning: a probabilistic perspective*. The MIT Press, Cambridge, MA, 2012.
- [16] Paul D. Allison. Convergence failures in logistic regression. 2008.
- [17] Dankmar Böhning and Bruce G Lindsay. Monotonicity of quadratic-approximation algorithms. *Annals of the Institute of Statistical Mathematics*, 40(4):641–663, 1988.
- [18] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. Efficient homomorphic comparison methods with optimal complexity. *Cryptology ePrint Archive*, Report 2019/1234, 2019. <https://eprint.iacr.org/2019/1234>.
- [19] Jung Hee Cheon, Dongwoo Kim, Duhyeong Kim, Hun Hee Lee, and Keewoo Lee. Numerical method for comparison on homomorphically encrypted numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 415–445. Springer, 2019.
- [20] Peter Harrington. *Machine learning in action*, 2012.