Improved privacy-preserving training using fixed-Hessian minimisation

Tabitha Ogilvie, Rachel Player, and Joe Rowell
Information Security Group, Royal Holloway, University of London

ABSTRACT

The fixed-Hessian minimisation method can be used to implement privacy-preserving machine learning training from homomorphic encryption. This is a relatively under-explored part of the literature, with the main prior work being that of Bonte and Vercauteren (BMC Medical Genomics, 2018), who proposed a simplified Hessian method for logistic regression training over the BFV homomorphic encryption scheme. Our main contribution is to revisit the fixed-Hessian approach for logistic regression training over the CKKS homomorphic encryption scheme. We improve on the prior work in several aspects, most notably showing how the native encoding in CKKS can be used to take advantage of SIMD operations. We implement our new fixed-Hessian approach in SEAL and compare it to more commonly-used minimisation methods, based on Gradient Descent and Nesterov's Accelerated Gradient Descent. We find that the fixed-Hessian approach exhibits fast run time and comparable accuracy to these alternative approaches. Moreover, it can be argued to be more practical in the privacy-preserving training context, as no step size parameter needs to be chosen.

As an additional contribution, we describe and implement three distinct training algorithms, based on Gradient Descent, Nesterov's Accelerated Gradient Descent, and a fixed-Hessian method respectively, to achieve privacy-preserving ridge regression training from homomorphic encryption. To the best of our knowledge, this is the first time homomorphic encryption has been used to implement ridge regression training on encrypted data.

CCS CONCEPTS

• **Security and privacy → Public key (asymmetric) techniques**;

KEYWORDS

homomorphic encryption; fixed Hessian; logistic regression; ridge regression; privacy-preserving training

1 INTRODUCTION

Cloud computing is becoming increasingly prevalent, with individuals, groups and companies outsourcing heavy computation to dedicated cloud computing services. At the same time, interest in machine learning techniques has grown, with many researchers and organisations keen to harness the predictive power of the data they already possess or have access to. In particular, many machine learning models are too complex to train on a general purpose machine, so outsourcing this computation to the cloud is a popular option. Moreover, it is common that the training data is sensitive or legally protected: for example, it could be the medical data of patients, the employment status of vulnerable members of society, or the financial records of a company. This motivates the problem of privacy-preserving machine learning training, which is the focus of this paper.

Specifically, we are interested in outsourcing the training of a machine learning model to the cloud, while keeping the training data confidential. This is a different context to that considered in many priors work on privacy-preserving training. For example, some prior works, such as [43, 46, 61], have sought to train models in a joint manner, or to train models on aggregated data, as in [12]. In our context, we do not want the cloud to learn from our data at all, but rather to perform the training as a service.

In our situation, we consider a client who possesses a quantity of data that they wish to use to build a machine learning model, but they lack the computational resources to handle this training locally. They would like to outsource the training to the cloud, but do not want the cloud to have access to their training data in the clear. Using homomorphic encryption [28] is a plausible solution for this scenario. The client encrypts their training data and transmits it to the cloud, who then homomorphically trains the model. The result is an encryption of the model, which is transmitted back to the client, who decrypts to obtain the model. We note that the security of homomorphic encryption depends on the absence of decryption oracles. For this reason, all the methods suggested in this paper would result in one-trip protocols: the client transmits encrypted data, the cloud returns an encrypted model.

Many works have shown that homomorphic encryption can be valuable at performing privacy-preserving inference on encrypted data [10, 15–18, 45, 66]. Privacy-preserving inference refers to the situation where the machine learning model has already been trained and is held on the cloud. The client then asks the cloud to use the model to make a prediction on a new datapoint, while keeping this datapoint confidential. In contrast, leveraging homomorphic encryption to train the model itself in a privacy-preserving way has been less widely studied.

Two common types of machine learning model are logistic regression and ridge regression. Logistic regression predicts the value of a dependent variable, e.g. malign or benign, based on the values of a set of independent variables. This technique is widely used in many disciplines, including genetics [60], healthcare [65], and economics [62]. Ridge regression is a form of linear regression that uses L2 regularisation to avoid overfitting. Its usage is ubiquitous in statistics, machine learning, and data mining [38, 44, 55, 63].

To train either a logistic regression or ridge regression model, we must minimise an appropriate cost function. To train these two types of machine learning model, in this paper, we consider the use of three possible minimisation methods: Gradient Descent, Nesterov's Accelerated Gradient Descent, and Newton-Raphson and Hessian-based methods. We mainly focus on the third of these three methods: that is, using a fixed-Hessian minimisation. The main prior work in this setting is that of Bonte and Vercauteren [14],

who used a fixed-Hessian method and the BFV [27] homomorphic encryption scheme to homomorphically train a logistic regression model.

1.1 Contributions

Our main contribution is to present a new implementation of privacy-preserving logistic regression training using fixed-Hessian minimisation. Our starting point is the work of Bonte and Vercauteren [14], which we improve upon in several aspects. First, we switch the underlying homomorphic encryption scheme from BFV to CKKS [22]. This enables us to use the native CKKS encoding to take advantage of SIMD operations while still being able to compute on non-integers. In contrast, [14] used w-NIBNAF encoding [13] to convert real numbers into sparse integer BFV plaintext polynomials and so were unable to achieve SIMD speedups. Next, we adopt a data preprocessing step similar to [40] and [41] to save layers during polynomial evaluations. In addition, instead of using the 1-degree Taylor approximation to the sigmoid function, we used the Chebyshev polynomial approximation used in [9]. We also adapted the approximation of the inverse function, using a linear approximation as the starting point and increasing the number of iterations from one to three. Finally, we increased the number of updates to the weights vector from one iteration to four.

We implement our new fixed-Hessian approach for logistic regression in Microsoft SEAL [59] version 3.5.1 and compare it to more commonly-used minimisation methods, based on Gradient Descent and Nesterov's Accelerated Gradient Descent. Our results are presented in Table 1, which shows that our fixed-Hessian approach achieves comparable accuracy and AUC to prior work, at a greater security level and with a faster runtime. In more detail, for a 5 fold CV average, applied to the Edinburgh dataset, our fixed-Hessian approach achieves 88.26% CV accuracy with a training time of 27 seconds per fold, while the Nesterov's Accelerated Gradient Descent approach of [40] achieves 88.90% with 42 seconds per fold. Moreover, we argue that the fixed-Hessian approach is more practical in the privacy-preserving training context, as no step size parameter needs to be chosen.

As an additional contribution, we describe and implement three separate privacy-preserving training algorithms for ridge regression. To the best of our knowledge, this is the first time homomorphic encryption has been used to implement ridge regression training on encrypted data. All the implementations reported on this work are publicly available on Github [52, 53].

1.2 Related work

The work of Bonte and Vercauteren [14] was a contribution to Track 3 of the iDASH 2017 competition[1], which concerned using homomorphic encryption to train a logistic regression model using a genomic data set of 1579 samples over 18 features. While [14] proposed a fixed-Hessian minimisation, the other finalists proposed solutions using Gradient Descent [19, 40]. The work [40] has since been updated [21, 35]. The iDASH 2017 Track 3 data set was also considered by Crawford *et al.* in [24]. Instead of using iterative methods, they opt for formulating the model parameters as the solution to a closed form approximation. Other works considering logistic regression training using homomorphic encryption include [4, 23, 30, 41, 57].

In Section 5 we consider the use of homomorphic encryption to enable a single client to completely outsource ridge regression training to a server. As far as we are aware, the use of homomorphic encryption for ridge regression training has not been considered in the literature. However, solutions based on multi-party computation have been proposed. For example, Nikolaenko *et al.* [49] considered many users contributing data to an evaluator who outputs a model in the clear. Other works considering privacy-preserving training using multiparty computation include [1, 29, 34, 37, 43, 46, 64].

1.3 Structure of the paper

We recall relevant background in Section 2. We describe our new fixed-Hessian minimisation approach for logistic regression in Section 3. We report on an implementation of our fixed-Hessian approach, and compare it with other minimisation methods for logistic regression, in Section 4. In Section 5, we describe and implement three different approaches to privacy-preserving training using ridge regression.

2 BACKGROUND

2.1 CKKS

The CKKS scheme [22] is a homomorphic encryption scheme that supports approximate arithmetic on floating point numbers. The key idea of CKKS is to interpret noise not as an error term but as part of the usual errors observed when computing with floating point numbers. We now recall the main features of the scheme that are relevant for our purposes, following the presentation of [40] and [9]. For additional details, including noise analysis, we refer the reader to [22].

CKKS has a native encoding function from the message space to the plaintext space, arising from the canonical embedding[2]. In more detail, the message space of CKKS is $\mathbb{C}^{N/2}$, and the encoding function is given by

$$\text{Encode}((m_1, ..., m_{N/2})) = \lfloor \triangle \cdot \phi^{-1}(m_1, m_2, ..., m_{N/2}) \rceil$$

where $\phi(\cdot)$ is the complex canonical embedding and $\triangle$ is the 'scale'; that is, the desired degree of precision. Like several other homomorphic encryption schemes, CKKS supports the encoding of multiple messages from the message space M into a single plaintext which enables "slotwise" operations on the messages.

The plaintext space of CKKS is $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$, where N is a power of 2. We employ the RNS variant [21] for efficient implementation, so that the initial ciphertext modulus is given by $Q_L = \prod_{i=1}^{L} p_i$ for distinct primes p_i, and lower level ciphertext moduli are given by $Q_l = \prod_{i=1}^{l} p_i, 0 < l \leq L$. Writing $\mathcal{R}_l = \mathcal{R}/Q_l\mathcal{R}$, we have that a level l ciphertext is a pair of polynomials (c_0, c_1) in the ciphertext space $\mathcal{R}_l^2$.

The CKKS scheme is comprised of the following algorithms, specified below: `KeyGen`, `SwitchKeyGen`, `EvalKeyGen`, `RotKeyGen`, `Enc`, `Dec`, `Add`, `AddPlain`, `Mult`, `MultPlain`, `Rescale`, `Rotate`.

[1] https://www.humangenomeprivacy.org/2017/

[2] For a comprehensive explanation of this embedding, and other algebraic background, we refer the reader to [20, 42].

- $\texttt{KeyGen}(1^\lambda)$: Sample a secret $s \leftarrow \chi_{enc}$, an error $e \leftarrow \chi_{err}$, and a random ring element $a \leftarrow \mathcal{R}_L$. Set the secret key as $\texttt{sk} \leftarrow (1, s)$ and the public key as $\texttt{pk} \leftarrow (b, a) \in \mathcal{R}_L^2$, where $b = -as + e \mod 2^L$. Output $(\texttt{sk}, \texttt{pk})$.
- $\texttt{SwitchKeyGen}(\texttt{sk}, s')$: Parse sk as $(1, s)$. For $s' \in \mathcal{R}$, sample a random $a' \leftarrow \mathcal{R}_{2L}$ and an error $e' \leftarrow \chi_{err}$. Set the switching key $\texttt{swk} \leftarrow (b', a') \in \mathcal{R}_{\in \mathcal{L}}{}^2$, where $b' = -a's' + e' + Q_L s \mod Q_{2L}$. Output swk.
- $\texttt{EvalKeyGen}(\texttt{sk})$: Output $\texttt{evk} \leftarrow \texttt{KSGen}(\texttt{sk}, s^2)$.
- $\texttt{RotKeyGen}(\texttt{sk}, \kappa)$: Output $\texttt{rk}^{(\kappa)} \leftarrow \texttt{KSGen}(\texttt{sk}, s^{(\kappa)})$.
- $\texttt{Enc}(\texttt{pk}, m)$: for a plaintext $m \in \mathcal{R}$, sample $v \leftarrow \chi_{enc}$ and $e_0, e_1 \leftarrow \chi_{err}$ and set $\texttt{ct} = v \cdot \texttt{pk} + (m + e_0, e_1) \mod Q_L$. Output ct.
- $\texttt{Dec}(\texttt{sk}, \texttt{ct})$: Parse ct as $(c_0, c_1) \in \mathcal{R}_l^2$. Set $m' = c_0 + c_1 \cdot s \mod Q_l$. Output m'.
- $\texttt{Add}(\texttt{ct}_1, \texttt{ct}_2)$: for two ciphertexts $\texttt{ct}_1, \texttt{ct}_2 \in \mathcal{R}_l$, output $\texttt{ct}_{\texttt{add}} = \texttt{ct}_1 + \texttt{ct}_2 \mod Q_l$
- $\texttt{AddPlain}(\texttt{ct}, c)$: for a ciphertext $\texttt{ct} = (b, a) \in \mathcal{R}_l^2$ and a plaintext $c \in \mathcal{R}$, output $\texttt{ct}_{\texttt{addp}} = (b + c, a) \mod Q_l$
- $\texttt{Mult}(\texttt{ct}_1, \texttt{ct}_2, \texttt{evk})$: for two ciphertexts $\texttt{ct}_1 = (b_1, a_1), \texttt{ct}_2 = (b_2, a_2) \in \mathcal{R}_l^2$, let $(d_0, d_1, d_2) = (b_1 b_2, a_1 b_2 + a_2 b_1, a_1 a_2) \mod Q_l$. Set $\texttt{ct}_{\texttt{mult}} = (d_0, d_1) + \lfloor b Q_L^{-1} \cdot d_2 \cdot \texttt{evk} \rceil \mod Q_l$. Output $\texttt{ct}_{\texttt{mult}}$.
- $\texttt{MultPlain}(\texttt{ct}, m)$: for a ciphertext $\texttt{ct} \in \mathcal{R}_l^2$ and a plaintext $m \in \mathcal{R}$, output $\texttt{ct}_{\texttt{multp}} \leftarrow m \cdot \texttt{ct} \mod Q_l$.
- $\texttt{Rescale}(\texttt{ct})$: for a ciphertext $\texttt{ct} \in \mathcal{R}_l$, output $\texttt{ct}_{\texttt{rs}} = \lfloor \texttt{ct}/p_l \rceil \mod Q_{l-1}$.
- $\texttt{Rotate}(\texttt{ct}, \texttt{rk}^{(\kappa)}, \kappa)$: for a ciphertext $\texttt{ct} = (b, a) \in \mathcal{R}_l$, output $\texttt{ct}_{\texttt{rot}} \leftarrow (b^{(\kappa)}, 0) + \lfloor Q_L^{-1} \cdot a^{(\kappa)} \cdot \texttt{rk}^{(\kappa)} \rceil \mod Q_l$.

When multiplying two ciphertexts that correspond respectively to messages m_1, m_2, each encoded with scale factor $\triangle$, the resulting ciphertext $\texttt{ct}_{\texttt{mult}}$ will correspond to the message $m_1 \cdot m_2$ with a scale factor of $\triangle^2$. We choose all of the intermediary primes, $p_2, ..., p_{L-1}$ to be approximately equal to $\triangle$. Thus, after a rescaling, $\texttt{ct}_{\texttt{rs}} = \texttt{Rescale}(\texttt{ct}_{\texttt{mult}})$ will correspond to the message $m_1 \cdot m_2$ with a scale factor of approximately $\triangle$.

We denote by $\texttt{ct}.m$ a ciphertext that corresponds to the message m. We sometimes make explicit the messages m_i in the 'slots' by writing $m = (m_1, m_2, ..., m_{N/2})$. In Algorithm 1 we recall a useful algorithm, AllSum, previously described in the literature [33, 41], which we will make extensive use of. Given a ciphertext $\texttt{ct}.m$, where $m = (m_1, m_2, ..., m_{N/2})$, AllSum generates a ciphertext $\texttt{ct}_{\texttt{all}}$ that corresponds to $\left(\sum_{i=1}^{N/2} m_i, \sum_{i=1}^{N/2} m_i, , ..., \sum_{i=1}^{N/2} m_i\right)$. This algorithm uses SIMD to generate the sum of $N/2$ complex numbers using $\log N/2$ additions.

Algorithm 1 AllSum($\texttt{ct}.m$)

Input: A ciphertext $\texttt{ct}.m$ corresponding to the message $(m_1, m_2, ..., m_{N/2})$

Output: a ciphertext $\texttt{ct}_{\texttt{all}}$ corresponding to the message $\left(\sum_{i=1}^{N/2} m_i, \sum_{i=1}^{N/2} m_i, ..., \sum_{i=1}^{N/2} m_i\right)$

$\texttt{ct}_{\texttt{all}} \leftarrow \texttt{ct}.m$
for $i = 0, \ldots, \log N/2 - 1$ **do**
  $\texttt{ct}_{\texttt{all}} \leftarrow \text{Add}\left(\texttt{ct}_{\texttt{all}}, \text{Rotate}(\texttt{ct}_{\texttt{all}}, 2^i)\right)$
end for

2.2 SEAL implementation of CKKS

All experiments in this paper were written in the Microsoft SEAL homomorphic encryption library [59], version 3.5.1. The SEAL library implements the RNS variants [5, 21] of both the BFV [27] and CKKS [22] homomorphic encryption schemes, having functions for encoding, encrypting, and all evaluation operations. In SEAL, the ciphertext modulus Q_L is specified via a set of bit lengths, $\{b_0, ..., b_k\}$. SEAL will then find primes $p_0, ..., p_k$ with $\log p_i \approx b_i$. These primes serve a variety of different functions, which specify how their bit lengths should be chosen. The prime p_0 is the decryption prime, so must be larger than the scale $\triangle$, giving $b_0 > \log \triangle$. The difference $(b_0 - \log \triangle)$ gives how many bits of precision we are guaranteed before the decimal point. The intermediary primes $p_1, ..., p_{k-1}$ are the rescaling primes, so should be set close to $\triangle$, giving $b_i = \log \triangle, 1 \leq i \leq k-1$. The final prime p_k is the so-called special prime [32] which is used during key switching, and SEAL requires these to be at least as big as all other primes in the ciphertext modulus, so that $b_k \geq b_0$.

The SEAL library follows the Homomorphic Encryption Standard [2] in order to ensure security of the parameters chosen. In particular, the underlying Learning with Errors (LWE) [56] instance is parameterised by the ring dimension N, the ciphertext modulus Q_L, a discrete Gaussian error distribution with standard deviation σ and a secret distribution S. We will always use an error distribution with $\sigma = 3.2$ and a uniform ternary secret distribution, which are are the default choices in SEAL. Following [2], we target 128-bit security and assume that the logarithm of the cost of lattice reduction with blocksize β in dimension d is $0.292\beta + 16.4 + \log(8d)$ [8]. In SEAL, as the product $Q_L = \prod_{i=0}^{k} p_i$ forms the initial ciphertext modulus, in order to achieve 128-bit security we must have $\sum_{i=0}^{k} b_i \leq \log Q_L$, where the value of $\log Q_L$ for a given N is as above. If a ciphertext modulus is specified by $\{b_0, ..., b_k\}$, the resulting parameters can handle circuits of depth $k - 1$.

2.3 Machine learning training models

2.3.1 Logistic regression. Logistic regression is a machine learning classification technique that predicts the value of a dependent variable, e.g. malign or benign, based on the values of a set of independent variables. In this work, we consider a binary logistic regression classification, where the value to be predicted takes one of two values, $y \in \{\pm 1\}$. Given model parameters $\beta = (\beta_0, \beta_1, ..., \beta_d)$, we calculate the probability that $y = 1$ given covariate values $(x_1, ..., x_d) \in \mathbb{R}^d$ as:

$$\mathbb{P}(y|\beta, \mathbf{x}) = \sigma(y\beta^{\mathrm{T}}\mathbf{x}) \,.$$

where we define $x_0 = 1$. To fit this model to our training data we need to maximise the likelihood estimator:

$$\prod_{i=1}^{n} \mathbb{P}(y_i|, \beta \mathbf{x}_i) = \prod_{i=1}^{n} \sigma(y\beta^{\mathrm{T}}\mathbf{x}) \,.$$

This is equivalent to minimising the following cost function:

$$J(\beta) = -\sum_{i=1}^{n} \log(1 + \exp(-y_i \beta^T \mathbf{x}_i)) \,. \tag{1}$$

For logistic regression we use two metrics to measure the quality of a derived model on a test set: AUC and accuracy. Accuracy corresponds to the proportion of the test set that the model correctly classifies against a threshold of 0.5. AUC stands for Area Under the receiver operating curve (ROC), and can be understood as the probability the derived model will rank a positive sample randomly chosen from the test set higher than a negative sample randomly chosen from the test set [47].

2.3.2 Ridge regression. Ridge regression is a form of linear regression that uses L2 regularisation to avoid overfitting. Given a set of n vectors with d features, $x_i \in \mathbb{R}^d$, and n output variables $y_i \in \mathbb{R}$, linear regression aims to find weights $\beta_0, \beta_1, ..., \beta_d \in \mathbb{R}$ such that

$$y_i \approx \beta_0 + \beta_1 x_{i1} + \beta_1 x_{i2} + ... + \beta_1 x_{id}$$

for each i. Augmenting each feature vector by setting $x_{i0} = 1$, we can rewrite this as $y_i \approx \beta^T x_i$. Ridge regression fits a linear model while penalising large values of β_j for $0 < j \leq d$ in order to stop the model from overfitting to the training data. The cost function for ridge regression is given by:

$$J(\beta) = \frac{1}{2}(\lambda \sum_{i=1}^{d} \beta_i^2 + \sum_{i=1}^{n} (y_i - \beta^T x_i)^2) \tag{2}$$

where the parameter λ is the regularisation parameter, reflecting how much we want to penalise large coefficients. This parameter is typically set via cross validation.

By differentiating Equation (2), we find that the j^{th} component of the gradient vector is given by

$$\frac{\partial J}{\partial \beta_j} = \begin{cases} \sum_{i=1}^{n} x_{ij}(\beta^T x_i - y_i) & j = 0 \\ \lambda \beta_j + \sum_{i=1}^{n} x_{ij}(\beta^T x_i - y_i) & j \neq 0 \end{cases}$$

Letting $\tilde{I}$ be the $(d+1) \times (d+1)$ diagonal matrix with 0 in the upper most entry and 1s otherwise, we can rewrite this as:

$$\nabla J(\beta) = ((X^T X + \lambda \tilde{I})\beta - X^T y) \tag{3}$$

where

$$y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} \quad \text{and} \quad X = \begin{pmatrix} x_{10} & x_{11} & \dots & x_{1d} \\ x_{20} & x_{21} & \dots & x_{2d} \\ \vdots & \ddots & & \\ x_{n0} & x_{n1} & \dots & x_{nd} \end{pmatrix}$$

so that ridge regression differs from logistic regression in that it has a closed form solution, given by:

$$\beta = (X^T X + \lambda \tilde{I})^{-1} X^T y. \tag{4}$$

The inverse of the matrix $X^T X + \lambda \tilde{I}$ can be found via Cholesky decomposition for $\lambda > 0$ with complexity $O(d^3)$ [49]. However, this technique is difficult to achieve homomorphically. Therefore, in this work, we will seek to minimise the cost function (2) directly. For numerical stability and comparability, we scale each covariate to the range $[0, 1]$, and centre the regressand via $y_i \mapsto y_i - \overline{y}$.

We can measure the predictive power of a ridge regression model using the coefficient of determination, or r^2, statistic:

$$r^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}} = 1 - \frac{\sum_{i=1}^{n}(y_i - \beta^T x_i)^2}{\sum_{i=1}^{n}(y_i - \overline{y})^2}.$$

A perfect model would have $r^2 = 1$, while a model that always predicts the mean $\overline{y}$ would have $r^2 = 0$. A model can receive a negative r^2 score, indicating that, on average, it is worse at predicting y_i than simply guessing the mean.

2.4 Optimisation techniques

In this section we recall the common approaches for homomorphically minimising a cost function, such as the logistic regression cost function given by Equation (1). These are Gradient Descent, Nesterov's Accelerated Gradient Descent, and Newton-Raphson and Hessian Based Methods.

2.4.1 Gradient Descent. Gradient Descent is perhaps the simplest minimisation technique, proceeding by taking successive steps "downhill". We use the observation that, at a given value of the parameters $\beta \in \mathbb{R}^{(d+1)}, d \geq 1$, we decrease the value of $J(\beta)$ fastest by taking a step in the direction of the negative of the gradient vector, $-\nabla J(\beta)$. Thus, we should perform updates of the form:

$$\beta^{(k+1)} = \beta^{(k)} - \alpha \nabla J(\beta) \tag{5}$$

where α is the step size or learning rate. Recall that, if $\beta = (\beta_0, ..., \beta_d)$ the coordinates of $\nabla J(\beta)$ are given by

$$\nabla J(\beta) = \left(\frac{\partial J}{\partial \beta_0}, \frac{\partial J}{\partial \beta_1}, ..., \frac{\partial J}{\partial \beta_d} \right).$$

If $\alpha > 0$ is small enough, (5) guarantees that $J(\beta^{(k+1)}) \leq J(\beta^{(k)})$. In other words, the value of our cost function shrinks every iteration. However, choosing α too small will result in very slow convergence. Typically the value of α will be chosen using line search [26, 31], or changed every iteration as a function of the previous values of β [6]. These techniques are difficult to achieve homomorphically.

Whenever J is convex and differentiable, and its gradient is Lipschitz continuous with constant L, we have that:

$$J(\beta^{(k)}) - J(\beta_{min}) \leq \frac{L}{2k} \left| \beta^{(0)} - \beta_{min} \right|_2^2,$$

where β_{min} is the global minimum, and we take a fixed step size $\alpha \leq 1/L$. Thus we can understand the convergence rate of Gradient Descent under this conditions to be $O(1/k)$, where k is the number of iterations. Equivalently, if we want our solution β to satisfy $J(\beta^k) - J(\beta_{min}) \leq \epsilon$, we need to complete $O(1/\epsilon)$ iterations. More details can be found in [50].

2.4.2 Nesterov's Accelerated Gradient Descent. Although Gradient Descent is guaranteed to converge for the right cost functions and the right choice of step size, it can exhibit a kind of "zigzagging" behaviour. To combat this, and hopefully achieve faster convergence, Nesterov's Accelerated Gradient Descent (NAD) maintains a momentum term. Intuitively, this can be understood as kicking a ball down a hill rather than taking fixed steps. Nesterov's Accelerated Gradient Descent maintains two terms: $\beta^{(k)}$, the model parameters, and $v^{(k)}$, the momentum term. Adopting the notation of [19], updates are then given by [48]:

$$\begin{cases} \beta^{(k+1)} & = v^{(k)} - \alpha_k \cdot \nabla J(v^{(k)}) \\ v^{(k+1)} & = (1 - \gamma_k) \cdot \beta^{(k+1)} + \gamma_k \cdot \beta^{(k)} \end{cases}$$

where $0 < \gamma_t < 1$ is a smoothing parameter.

We set $\gamma_t = 1 - t_k/t_{k+1}$, according to the Fast Iterative Shrinkage-Threshold Algorithm [7], where $t_1 = 1$ and $t_{k+1} = \frac{1+\sqrt{1+4t_k^2}}{2}$. For this choice of γ_t and a good choice of α_t, Nesterov's Accelerated Gradient Descent exhibits convergence at rate $O(1/k^2)$ in the number of iterations k.

2.4.3 Newton-Raphson and Hessian Based Methods. We proceed from the observation that the vector β that minimises the cost function will return a gradient vector of zero [14]. For example, for the logistic regression cost function of Equation (1), and for $j \in \{0, \ldots, d\}$, the j^{th} component of this gradient vector at a point $\beta \in \mathbb{R}^{(d+1)}$ is given by

$$\nabla l(\beta)_j = \frac{\partial J}{\partial \beta_j} = \sum_{i=1}^{n} (1 - \sigma(y_i \beta^{\mathrm{T}} \mathbf{x_i})) y_i x_{ij}\,, \tag{6}$$

where we define $x_{i0} = 1$. To find the root $\beta \in \mathbb{R}^{(d+1)}$ to $\nabla l(\beta) = \mathbf{0}$, we use the Newton-Raphson root finding method. Given the right kind of function and a suitable starting point, Newton-Raphson guarantees very fast convergence [11].

For our running example of Equation (6), in the multivariate equation $\nabla l(\beta) = \mathbf{0}$, these updates are given by

$$\beta^{(k+1)} = \beta^{(k)} - H(\beta^{(k)})^{-1} \nabla(\beta^{(k)}), \tag{7}$$

where the matrix $H(\beta) \in \mathbb{R}^{(d+1)\times(d+1)}$ is the Hessian matrix, given element-wise by

$$H(\beta)_{ij} = \frac{\partial^2 J}{\partial \beta_j \partial \beta_j} = -\sum_{k=1}^{n} (1 - \sigma(y_k \beta^{\mathrm{T}} \mathbf{x_k})) \sigma(y_k \beta^{\mathrm{T}} \mathbf{x_k}) (y_k)^2 x_{kj} x_{ki}\,.$$

3 IMPROVED FIXED-HESSIAN TRAINING FOR LOGISTIC REGRESSION

In this section we describe our improvements to the fixed-Hessian minimisation approach of [14], in the context of logistic regression training over CKKS. We first present an overview of our approach.

Recall from Section 2.4.3 that we are interested in evaluating the update (7) homomorphically. Doing so directly presents three difficulties. Firstly, we need to approximate the sigmoid function using a polynomial. Secondly, evaluating the value of the Hessian $H(\beta_k)$ at each iteration is extremely costly in terms of levels. Thirdly, performing a matrix inversion for an arbitrary matrix each iteration is very challenging. To address the first problem, we adopt the linear approximation to the sigmoid function given in [9], given by:

$$\sigma(x) \approx \frac{1}{2} + \frac{5}{32} x, \tag{8}$$

found by approximating the sigmoid function with Chebyshev polynomials.

For the second and third difficulties, we take a similar approach to [14], who proceed by replacing the full Hessian $H(\beta)$ with a fixed diagonal matrix $\tilde{H}$. The diagonal entries of this matrix $\tilde{H}$ are given by:

$$-\tilde{H}_{kk} = \frac{1}{4} \sum_{i=0}^{d} \sum_{j=1}^{n} x_{jk} x_{ji}\,. \tag{9}$$

We show how to homomorphically evaluate the entries of $\tilde{H}$ in Algorithm 2.

Substituting $H(\beta)$ with $\tilde{H}$ results in losing the guarantee of quadratic convergence Newton-Raphson normally provides. However, since $H(\beta) \geq \tilde{H}$, it is argued in [14] that, by using the results from [11], the resulting Newton-Raphson sequence is monotonically decreasing, exhibits guaranteed convergence, and still converges linearly, with rate $c < 1$.

Fixing the matrix $\tilde{H}$ means the updates (c.f. Equation (7)) that we require to evaluate are now given by:

$$\beta^{(k+1)} = \beta_k - \tilde{H}^{-1} \nabla(\beta^{(k)})\,. \tag{10}$$

Since we have replaced the Hessian with a diagonal matrix, to perform the matrix inversion $(-\tilde{H})^{-1}$ we simply need to find the values $\frac{1}{-\tilde{H}_{kk}}$ for $0 \leq k \leq d$. We show how to do so in Algorithm 3.

Note that, like the sigmoid function, this matrix inversion needs to be approximated with a polynomial evaluation. The authors of [14] use Newton-Raphson in one variable applied to the function $f(x) = \frac{1}{x} - a$, where a is the number they wish to invert. In [14], a fixed starting value and one iteration is used. In contrast, we increase the number of iterations, and use a linear approximation as the initial value, to minimise the relative error. Our approach is discussed in more detail in Section 3.3.

Putting everything together, having evaluated $(-\tilde{H})^{-1}$, we are able to evaluate an approximation of the circuit given by Equation (10). We show how to do so in Algorithm 4.

3.1 Encoding

An appropriate choice of encoding for the training data can enable us to reduce the number of levels required in Algorithm 4. We use a "feature by feature" encoding, similar to [41]. Specifically, we define $z_{ji} \coloneqq y_j x_{ji}$, and then construct $(d+1)$ messages

$$Z_i = \left(\frac{z_{1i}}{2}, \frac{z_{2i}}{2}, ..., \frac{z_{ni}}{2}\right) \in \mathbb{R}^n\,.$$

As long as $n \leq N/2$, these messages can be encoded into the plaintext space and then encrypted, giving $(d+1)$ ciphertexts $\mathtt{ct}.Z_i$ for $0 \leq i \leq d$ that encrypt the training data. The factor $1/2$ saves levels during training due to the choice of sigmoid approximation. For a higher degree approximation, a different constant factor should be used: for example, [22] uses a factor of $1/8$. A more compact encoding method exists [40], but it requires extra levels to compensate.

3.2 Homomorphic evaluation of $\tilde{H}$

We first show how to homomorphically evaluate the entries of $\tilde{H}$. We begin by rewriting Equation (9) as follows:

$$-\tilde{H}_{kk} = \sum_{i=0}^{d} \sum_{j=1}^{n} \frac{y_j x_{jk}}{2} \frac{y_j x_{ji}}{2} = \sum_{j=1}^{n} \frac{z_{jk}}{2} \sum_{i=0}^{d} \frac{z_{ji}}{2} \tag{11}$$

where the first equality exploits the fact that $y_i \in \{\pm 1\}$ and the second comes from swapping the order of summation and using the definition of z_{ij}. We can therefore evaluate the diagonal entries of $\tilde{H}$ using Algorithm 2, which consumes one level, and from line 5 can be evaluated in parallel. For ease of exposition, in Algorithm 2 we omit the relinearisation steps that follow all ciphertext-ciphertext multiplications. In our implementation, we parallelised the loop given by lines 5 - 9 as each of the ciphertexts can be operated on independently. We remark that the usage of SIMD in this Algorithm

Algorithm 2 Homomorphic evaluation of the entries of $-\tilde{H}$.

Input: Ciphertexts $\{\mathtt{ct}.Z_j\}_{j=0,\ldots,d}$
Output: Ciphertexts $\{\mathtt{ct}.\tilde{H}_j\}_{j=0,\ldots,d}$

1: $\mathtt{ct}_{\text{sum}} \leftarrow \mathtt{ct}.z_0$
2: **for** $j = 1, \ldots, d$ **do**
3: $\quad \mathtt{ct}_{\text{sum}} \leftarrow \text{Add}(\mathtt{ct}_{\text{sum}}, \mathtt{ct}.z_j)$
4: **end for** ▷ the kth entry of $\mathtt{ct}_{\text{sum}}$ corresponds to $\sum_{i=0}^{d} \frac{z_{ki}}{2}$
5: **for** $j = 0, \ldots, d$ **do**
6: $\quad \mathtt{ct}.\tilde{H}_j \leftarrow \text{Mult}(\mathtt{ct}_{\text{sum}}, \mathtt{ct}.z_j)$
7: $\quad \mathtt{ct}.\tilde{H}_j \leftarrow \text{RS}(\mathtt{ct}.\tilde{H}_j)$
8: $\quad$ ▷ the kth entry of $\mathtt{ct}.\tilde{H}_j$ corresponds to $\frac{z_{kj}}{2} \sum_{i=0}^{d} \frac{z_{ki}}{2}$
9: $\quad \mathtt{ct}.\tilde{H}_j \leftarrow \text{AllSum}(\mathtt{ct}.\tilde{H}_j)$
10: $\quad$ ▷ Every entry of $\mathtt{ct}.\tilde{H}_j$ corresponds to $-\tilde{H}_{jj}$ as defined in (11)
11: **end for**

enables the complexity of the algorithm to stay constant in the size of the dataset n so long as $n \leq N/2$, and beyond this range the complexity of the modified procedure would only increase linearly in $\lfloor \frac{2n}{N} \rfloor$.

3.3 Homomorphic evaluation of $-\tilde{H}^{-1}$

Having homomorphically evaluated the entries of the matrix $-\tilde{H}$, it is necessary to calculate their inverse. This is achieved in Algorithm 3, where we present a general procedure for finding $\frac{1}{d}$ for an arbitrary value d which can be bounded to an interval $[a, b]$. This Algorithm is fully parallelisable, and consumes $1 + 2\kappa$ levels, where κ is the number of iterations.

If the number we want to invert is d, we are seeking a root to the equation $f(x) = \frac{1}{x} - d$, which has Newton-Raphson updates of the form

$$x_{(k+1)} = 2x_{(k)} - dx_{(k)}^2 .$$

However, the speed at which this converges is very sensitive to the initial value. In everyday usage, practitioners have the advantage of being able to see the number they are trying to invert, and so can use lookup tables or approximate $\frac{1}{x}$ over a tight range to minimise the initial error and guarantee fast convergence [51]. Since we are working over encrypted data, we use a linear approximation, $T_1 + T_2 d$. We use the results from [58], who show that, in order to minimise the relative error in both the initial and final approximations over a range $[a, b]$, one should take:

$$T_1(a, b) = \frac{8(a + b)}{a^2 + 6ab + b^2}, \qquad T_2(a, b) = \frac{-8}{a^2 + 6ab + b^2}$$

Taking $a = 1$ and $b = X$ for X a value we will discuss later, we now have that the maximum relative error after κ iterations of Newton-Raphson is given by [58]:

$$R_{max}(\mu) = \left(\frac{(X - 1)^2}{X^2 + 6X + 1} \right)^{2^\kappa} .$$

To choose X, we need to have some idea about the maximum possible absolute value of each $-\tilde{H}_{kk}$. Examining (11), we see this cannot exceed $\frac{B(d+1)n}{4}$, where B is an upper bound on the absolute value of each feature. If the client is uncomfortable sending this value of $X = \frac{B(d+1)n}{4}$ in the clear, they could instead encrypt the values T_1 and T_2 and transmit them with the training data. In Algorithm 3 we assume the T_1 and T_2 are encoded in to the plaintexts $\mathtt{pt}.T_1$ and $\mathtt{pt}.T_2$ respectively. In our implementation, we parallelised the entirety of this process into $d + 1$ threads.

Algorithm 3 Invert($\mathtt{ct}.\tilde{H}_j, \mathtt{pt}.T_1, \mathtt{pt}.T_2, \kappa$): Homomorphic evaluation of $\frac{1}{-\tilde{H}_{kk}}$ for $k = 0, \ldots, d$

Input: Ciphertexts $\{\mathtt{ct}.\tilde{H}_j\}_{j=0,\ldots,d}$, plaintexts $\mathtt{pt}.T_1, \mathtt{pt}.T_2$, and a number of iterations κ
Output: Ciphertexts $\{\mathtt{ct}.\overline{H}_j\}_{j=0,\ldots,d}$ where $\overline{H}_j$ corresponds to $\frac{1}{\tilde{H}_{jj}}$

1: **for** $j = 0, \ldots, d$ **do**
2: $\quad \mathtt{ct}.\overline{H}_j \leftarrow \text{Mult}(\mathtt{pt}.T_2, \mathtt{ct}.\tilde{H}_j)$
3: $\quad \mathtt{ct}.\overline{H}_j \leftarrow \text{RS}(\mathtt{ct}.\overline{H}_j)$
4: $\quad \mathtt{ct}.\overline{H}_j \leftarrow \text{Add}(\mathtt{pt}.T_1, \mathtt{ct}.\overline{H}_j)$
5: $\quad$ ▷ We have (homomorphically) initialised $\overline{H}_j$ to $T_1 + T_2\tilde{H}_j$
6: $\quad$ **for** k=1,...,κ **do**
7: $\quad\quad \mathtt{ct}_{\text{temp}} \leftarrow \text{Add}(\mathtt{ct}.\overline{H}_j, \mathtt{ct}.\overline{H}_j)$
8: $\quad\quad \mathtt{ct}.\overline{H}_j \leftarrow \text{Mult}(\mathtt{ct}.\overline{H}_j, \mathtt{ct}.\overline{H}_j)$
9: $\quad\quad \mathtt{ct}.\overline{H}_j \leftarrow \text{RS}(\mathtt{ct}.\overline{H}_j)$
10: $\quad\quad \mathtt{ct}.\overline{H}_j \leftarrow \text{Mult}(\mathtt{ct}.\overline{H}_j, \mathtt{ct}.\tilde{H}_j)$
11: $\quad\quad \mathtt{ct}.\overline{H}_j \leftarrow \text{RS}(\mathtt{ct}.\overline{H}_j)$
12: $\quad\quad \mathtt{ct}.\overline{H}_j \leftarrow \text{Subtract}(\mathtt{ct}_{\text{temp}}, \mathtt{ct}.\overline{H}_j)$
13: $\quad$ **end for**
14: **end for**

3.4 Putting it all together

Having evaluated the entries of our fixed Hessian, we are able to evaluate Equation (10), the updates on the weights β. This is done in Algorithm 4, which consumes $1 + 3(\mu - 1)$ levels, where μ is the number of iterations.

Component wise, using Equation (6) and Equation (8), we must evaluate:

$$\beta^{(k+1)}(j) = \beta^{(k)}(j) + \overline{H}_j \left(\sum_{i=1}^{n} \frac{z_{ij}}{2} - \sum_{i=1}^{n} \frac{5z_{ij}}{8 \cdot 2} \frac{\beta^{(k)} \cdot \mathbf{z}_i}{2} \right) .$$

Note that $\sum_{i=1}^{n} \frac{z_{ij}}{2}$ can be precomputed as it does not depend on β_k, while only $\beta^T \mathbf{z}_i$ cannot be computed feature-by-feature, as the inner product requires all $d + 1$ features. In our implementation, we initialise all weights to zero, so the first iteration is given simply by $\beta^{(1)} = \overline{H}_j \sum_{i=1}^{n} \frac{z_{ij}}{2}$. In our implementation we parallelised the loop given by lines 11 - 18.

Algorithms 2, 3 and 4, employed consecutively, constitute the entire learning process. In total, this process consumes $2 + 2\kappa + 3(\mu - 1)$ levels, where κ corresponds to the number of iterations in Algorithm 3, and μ corresponds to the number of updates to the weight vector β as in Algorithm 4.

4 IMPLEMENTATION AND COMPARISON

In this section we report on an implementation of our improved fixed-Hessian minimisation method for logistic regression that was introduced in Section 3. We also present (in Table 1) experimental results that enable us to compare our approach to prior works [40,

Algorithm 4 Homomorphic evaluation of fixed-Hessian updates to the weights $\{\beta_j\}_{j=0,1...,d}$ for logistic regression

Input: Ciphertexts $\{\mathtt{ct}.\overline{H}_j\}_{j=0,...,d}$, $\{\mathtt{ct}.Z_j\}_{j=0,...,d}$, a plaintext $\mathtt{pt}.a_1$ which has $\frac{5}{8}$ in every slot, and an iteration number μ
Output: Ciphertexts $\{\mathtt{ct}.\beta_j\}_{j=0,1...,d}$

for $j = 0, 1..., d$ **do**
  $\mathtt{ct}_{\mathrm{allsum}_j} \leftarrow \mathrm{AllSum}(\mathtt{ct}.Z_j)$
  $\mathtt{ct}.\beta_j \leftarrow \mathrm{Mult}(\mathtt{ct}.\overline{H}_j, \mathtt{ct}_{\mathrm{allsum}_j})$
  $\mathtt{ct}.\beta_j \leftarrow \mathrm{RS}(\mathtt{ct}.\beta_j)$
end for
for $k = 2, ..., \mu$ **do**
  $\mathtt{ct}_{\mathrm{ip}} \leftarrow \mathrm{RS}(\mathrm{Mult}(\mathtt{ct}.\beta_0, \mathtt{ct}.Z_0))$
  for $j = 1, ..., d$ **do**
    $\mathtt{ct}_{\mathrm{ip}} \leftarrow \mathrm{Add}(\mathtt{ct}_{\mathrm{ip}}, \mathrm{RS}(\mathrm{Mult}(\mathtt{ct}.\beta_j, \mathtt{ct}.Z_j)))$
  end for
  for $j = 0, ..d$ **do**
    $\mathtt{ct}_{\mathrm{grad}} \leftarrow \mathrm{RS}(\mathrm{Mult}(\mathtt{pt}.a_1, \mathtt{ct}.Z_j))$
    $\mathtt{ct}_{\mathrm{grad}} \leftarrow \mathrm{RS}(\mathrm{Mult}(\mathtt{ct}_{\mathrm{ip}}, \mathtt{ct}_{\mathrm{grad}}))$
    $\mathtt{ct}_{\mathrm{grad}} \leftarrow \mathrm{AllSum}(\mathtt{ct}_{\mathrm{grad}})$
    $\mathtt{ct}_{\mathrm{grad}} \leftarrow \mathrm{Subtract}(\mathtt{ct}_{\mathrm{allsum}_j}, \mathtt{ct}_{\mathrm{grad}})$
    $\mathtt{ct}_{\mathrm{grad}} \leftarrow \mathrm{RS}(\mathrm{Mult}(\mathtt{ct}.\overline{H}_j, \mathtt{ct}_{\mathrm{grad}}))$
    $\mathtt{ct}.\beta_j \leftarrow \mathrm{Add}(\mathtt{ct}.\beta_j, \mathtt{ct}_{\mathrm{grad}})$
  end for
end for

41] that also implement homomorphic logistic regression training using (respectively) Gradient Descent minimisation and Nesterov's Accelerated Gradient Descent.

4.1 Experimental set up

We compare our fixed-Hessian minimisation (FH) for logistic regression to the Gradient Descent (GD) approach of [41] and the Nesterov's Accelerated Gradient Descent (NAD) approach of [40] for a 5 fold CV average, applied to the Edinburgh dataset. This dataset contains 1253 observations, each with one classification and 9 covariates [39].

For a fair comparison, we have altered the underlying Learning with Errors (LWE) parameter sets used by [41] and [40] in order to increase the targeted security level from 80-bit to 128-bit. In more detail, we always use the SEAL recommended parameter set $(N, \log Q_L)$ for ring dimension $N = 2^{15}$ to achieve an estimated 128-bit security according to the Homomorphic Encryption Standard [2]. The number of iterations μ, and the requirement to decrypt with a certain precision, imposes a minimum bit length of Q_L. We set μ so that $\log Q_L$ is as close as possible to the maximum value permitted by the security analysis, and then compose Q_L to permit the computation.

We choose to decrypt using 10 bits of precision, so that for Gradient Descent using a degree 3 sigmoid approximation as in [41] the bit length of $\log Q_L$ is given by $2\cdot(\log\triangle+10)+(4(\mu-1)+1)\log\triangle$, while for Nesterov's, using the same approximation, a lower bound on $\log Q_L$ is given by $2\cdot(\log\triangle+10)+(6(\mu-1)+1)\log\triangle$. For our fixed-Hessian method, a lower bound on $\log Q_L$ is given by $2\cdot(\log\triangle+10)+(2+2\kappa+3(\mu-1))\log\triangle$.

Descent	Enc	$\log\triangle$	$\deg g$	μ	Time	AUC	Acc. (%)
GD [41]	F	27	3	8	102.18s	0.89	82.53
	F	28	3	7	59.12s	0.93	83.65
	F	31	3	7	58.62s	0.95	80.46
NAD [40]	D	30	3	5	42.51s	0.96	88.90
	D	31	3	5	42.57s	0.94	88.74
FH (Ours)	F	39	1	5	45.70s	0.94	88.50
	F	40	1	4	27.18s	0.92	88.26
	F	45	1	4	30.98s	0.92	88.26

Table 1: Comparison of our fixed-Hessian minimisation (FH) for logistic regression to the Gradient Descent (GD) approach of [41] and the Nesterov's Accelerated Gradient Descent (NAD) approach of [40]. All rows correspond to parameter settings targetting 128-bit security. The column 'Enc' denotes the type of encoding, either feature-by-feature (F) or database (D). The column '$\deg g$' denotes the degree of the sigmoid approximation. The column 'μ' denotes the number of iterations. The column 'Time' gives the training time in seconds. The column 'Acc.' gives the accuracy. All computations were carried out across 10 cores on a computer with two "Intel Xeon E5-2690v4 @ 2.6GHz" and 256GB of RAM.

In the original implementations [40, 41], the parameter $\log\triangle$ was set to 28 bits, respectively 30 bits. At the higher security level of 128-bit, this permits 7, respectively 5, iterations. We report on implementation using these values of $\log\triangle$, but we also calculated that the same number of iterations can be performed at this security level, while increasing both the values of $\log\triangle$ to 31 bits. We also experimented by calculating the maximum values of $\log\triangle$ that would permit us to perform an additional iteration: these are 27 bits for Gradient Descent, and 26 bits for Nesterov. The 26-bit scale proved too low to complete the algorithm, while we observed the Gradient Descent algorithm was non-deterministic, so we report the averages over 5 runs. We retain the original encoding style: either feature-by-feature, as in this work, or a more compact database encoding, as in [40]; as well as the original degree $\deg g$ of the sigmoid approximation.

We note that our re-implementation of [40] uses one value of $\triangle$ throughout, while the authors of [40] suggest performing several of the plaintext multiplications at a lower precision in order to reduce the bits of ciphertext modulus consumed in these operations and so increase the number of updates to the parameters β. They suggest encoding certain plaintexts at 20-bit precision while encoding the data at 30-bit precision, and this modification would have enabled us to calculate an extra iteration while maintaining the same security level. Unfortunately, SEAL calculated that there weren't enough appropriate 20-bit primes to use in the RNS modulus, and so we were unable to implement this improvement.

For our own fixed-Hessian method we needed a higher precision of around $\log\triangle = 40$ as many of the numbers $\frac{1}{H_{kk}}$ were very close to zero. We set the number of iterations in Algorithm 3 as $\kappa = 3$, so that at 39-bit precision we could perform 5 iterations, while at 45-bit precision we can perform 4 iterations. The results of our comparison can be found in Table 1.

4.2 Analysis and discussion

When running logistic regression on the Edinburgh dataset in the clear, we observed a maximum possible CV accuracy of 91% and CV AUC of 0.96. Therefore, the results of Table 1 suggest that both the fixed-Hessian method and the Nesterov's Accelerated Gradient Descent method proposed in [40] offer a promising solution to outsourced logistic regression model training based on homomorphic computation.

However, Nesterov's Accelerated Gradient Descent requires the choice of a step size α_k, and, unlike the smoothing parameter γ_k, it is not clear how to choose this value. In [40] and [41] the authors use a harmonic learning rate of $\alpha_k = \frac{10}{1+k}$, and we adopted this approach for our re-implementation. Although this is more general than a fixed value, it still requires the selection of a numerator as in some cases the value of 10 will be much too large.

In practice, we might choose the step size α_k using a backwards line search: selecting a value, performing an update, and then calculating accuracy and either proceeding or taking a smaller step depending on whether the accuracy increases. We could alternatively use a grid search, which entails training various models with various steps sizes and then either selecting the model that shows the best CV performance or retraining with a new step size based on which selections performed best. Either of these methods could lead to security concerns associated with creating an interactive protocol where the server can in essence ascertain which ciphertexts correspond to "good" models and which do not, as well as increasing the computational cost. We will discuss the difficulties of choosing α further in the context of ridge regression in Section 5.

4.3 Extensions for larger ring dimension

Using a larger ring dimension $N = 2^{16}$ would enable us to increase the number of Gradient Descent iterations from 7 to 15, Nesterov's Accelerated Gradient Descent iterations from 5 to 10, and fixed-Hessian iterations from 4 to 10. A parameter set with $N = 2^{16}$ is not recommended in the Homomorphic Encryption Standard [2] and hence is not directly available in SEAL. Similar to the suggestion in [25], for $N = 2^{16}$, with an error distribution $\sigma = 3.2$, and a uniform ternary secret distribution, we can chose a ciphertext modulus of bitsize $\log Q_L = 1775$. The LWE estimator[3] of [3] estimates this parameter set to have 128-bit security, assuming (in line with [2]) that logarithm of the cost of lattice reduction with blocksize β in dimension d is $0.292\beta + 16.4 + \log(8d)$.

We were unable to increase the iteration number for Gradient Descent and Nesterov's Accelerated Gradient Descent, so we cannot present a full comparison analogous to Table 1 for the $N = 2^{16}$ parameter set. For both these minimisation methods, we found that accuracy and AUC decreased sharply after around 4 iterations, which we suspect is due to noise corruption from the higher degree sigmoid approximation coupled with a lower precision. Plaintext experimentation indicates that this would increase the Gradient Descent CV accuracy to 86% and the Nesterov's Accelerated Gradient Descent accuracy to 90%. For the fixed-Hessian method in this setting, we obtained a CV accuracy of 91% and CV AUC of 0.95: these results are close to the best possible.

[3] https://bitbucket.org/malb/lwe-estimator, commit fb7deba

In summary, our experiments show that the fixed-Hessian method can be preferred to Nesterov's Accelerated Gradient Descent for outsourcing logistic regression model training, as it offers very high accuracy and fast running times, while avoiding the requirement to find and set an appropriate step size α.

5 RIDGE REGRESSION TRAINING

In this section, we consider how to achieve privacy-preserving ridge regression training. Recall from Section 2.3.2 that we are interested in minimising the ridge regression cost function (Equation (2)) and could do so, for example, using Gradient Descent, Nesterov's Accelerated Gradient Descent, or a fixed-Hessian approach.

5.1 Gradient Descent and Nesterov's Accelerated Gradient Descent

We first outline how to evaluate a Gradient Descent approach to ridge regression training homomorphically, and then explain how to extend these methods to compute Nesterov's Accelerated Gradient Descent updates.

The Gradient Descent updates for ridge regression are given by

$$\beta_j \leftarrow \beta_j - \alpha \frac{\partial J}{\partial \beta_j} \tag{12}$$

where the right hand side is given by

$$(1 - \lambda\alpha \mathbb{1}_{\{j\neq 0\}})\beta_j + \underbrace{\alpha \sum_{i=1}^{n} y_i x_{ij}}_{Y_j} - \sum_{k=0}^{d} \beta_k \underbrace{\alpha \sum_{i=1}^{n} x_{ij} x_{ik}}_{M_{jk}}$$

and α is the Gradient Descent learning rate. We can fix a learning rate α, which enables us to save time and levels by precomputing the quantities $Y_j = \alpha \sum_{i=1}^{n} y_i x_{ij}, 0 \le j \le d$ and $M_{jk} = \alpha \sum_{i=1}^{n} x_{ij} x_{ik}, 0 \le j, k \le d$. The precomputation of Y and M_{jk} is given in Algorithms 5 and 6 respectively. Each algorithm consumes 2 levels and they can be run in parallel to each other. In our implementation, Algorithm 5 is entirely parallelised, whereas for Algorithm 6 we only calculate the inner loop given by lines 8 - 12 in parallel.

To encode the training data, we opt again for a "feature by feature" encoding, creating messages y and X_j where

$$y = (y_1, y_2, ..., y_n),$$
$$X_j = (x_{1j}, x_{2j}, ..., x_{nj})$$

Note that in practice, the vector X_0 consists entirely of 1s, so does not need to be transmitted. We also define two sets of plaintexts, C and I, which can be constructed by the server. The set I consists of $d+1$ plaintexts, each corresponding to a row of the $(d+1)\times(d+1)$ identity matrix, so that multiplying by $\mathtt{pt}.I_j$ corresponds to deleting all but the j^{th} slot in the underlying message. The set C is the same except instead of 1s on the diagonal, we have the fixed learning rate α.

Having calculated the ciphertexts $\mathtt{ct}.Y$ and $\{\mathtt{ct}.M_j\}_{j=0,...,d}$, we are able to perform Gradient Descent updates to the parameters β_j as follows. The weights are encoded as a single message $\beta = (\beta_0, ..., \beta_d)$. We compute the updated parameters one feature at a

Algorithm 5 Homomorphic evaluation of $Y = (Y_0, ..., Y_d)$

Input: Ciphertexts $\{\mathtt{ct}.X_j\}_{j=0,...,d}$ and $\mathtt{ct}.y$, plaintexts $\{\mathtt{pt}.C_j\}_{j=0,...,d}$
Output: A ciphertext $\mathtt{ct}.Y$ which corresponds to the message $(Y_0, ..., Y_d)$, with Y_j defined by $Y_j = \alpha \sum_{i=1}^{n} y_i x_{ij}$

$\mathtt{ct}.Y \leftarrow \mathtt{ct}.0$
for $j = 0, ..., d$ **do**
  $Z \leftarrow \text{Mult}(\mathtt{ct}.Y, \mathtt{ct}.X_j)$
  $Z \leftarrow \text{RS}(Z)$
  $Z \leftarrow \text{AllSum}(Z)$
  $Z \leftarrow \text{Mult}(\mathtt{pt}.C_j, Z)$
  $Z \leftarrow \text{RS}(Z)$
  $Y \leftarrow \text{Add}(\mathtt{ct}.Y, Z)$
end for

Algorithm 6 Homomorphic evaluation of M_j

Input: Ciphertexts $\{\mathtt{ct}.X_j\}_{j=0,...,d}$ and plaintexts $\{\mathtt{pt}.C_j\}_{j=0,...,d}$
Output: Ciphertexts $\{\mathtt{ct}.M_j\}_{j=0,...,d}$, where $\mathtt{ct}.M_j$ corresponds to the message $(M_{j0}, M_{j1}, ..., M_{jd})$, with M_{jk} defined by $M_{jk} = \alpha \sum_{i=1}^{n} x_{ik} x_{ij}$

for $j = 0, ..., d$ **do**
  $\mathtt{ct}.M_j \leftarrow \text{Mult}(\mathtt{ct}.X_j, \mathtt{ct}.X_j)$
  $\mathtt{ct}.M_j \leftarrow \text{RS}(\mathtt{ct}.M_j)$
  $\mathtt{ct}.M_j \leftarrow \text{AllSum}(\mathtt{ct}.M_j)$
  $\mathtt{ct}.M_j \leftarrow \text{Mult}(\mathtt{ct}.M_j, \mathtt{pt}.C_j)$
  $\mathtt{ct}.M_j \leftarrow \text{RS}(\mathtt{ct}.M_j)$
  for $k = 0, .., j-1$ **do**
    $Z \leftarrow \text{Mult}(\mathtt{ct}.X_j, \mathtt{ct}.X_k)$
    $Z \leftarrow \text{RS}(Z)$
    $Z \leftarrow \text{AllSum}(Z)$
    $\mathtt{ct}.M_j \leftarrow \text{Add}(\mathtt{ct}.M_j, \text{RS}(\text{Mult}(\mathtt{pt}.C_k, Z)))$
    $\mathtt{ct}.M_k \leftarrow \text{Add}(\mathtt{ct}.M_k, \text{RS}(\text{Mult}(\mathtt{pt}.C_j, Z)))$
  end for
end for

time, first multiplying $\mathtt{ct}.\beta$ by $\mathtt{ct}.M_j$ to obtain a ciphertext that corresponds to

$$(\alpha\beta_0 \sum_{i=1}^{n} x_{ij}x_{i0}, \alpha\beta_1 \sum_{i=1}^{n} x_{ij}x_{i1}, ..., \alpha\beta_d \sum_{i=1}^{n} x_{ij}x_{id}).$$

We then AllSum and multiply by I_j. Summing all the resulting ciphertexts and subtracting from $\mathtt{ct}.Y$, we calculate a ciphertext $\mathtt{ct}_{\text{grad}}$ that has its j^{th} entry

$$\alpha \sum_{i=1}^{n} y_i x_{ij} - \alpha \sum_{k=0}^{d} \beta_k \sum_{i=1}^{n} x_{ij}x_{ik}$$

exactly as is required in (12), so that we can update all parameters simultaneously. We initialise all weights to zero, so that the first iteration is simply $\beta = Y$. Algorithm 7 perform μ iterations of this process. Updating the parameters β consumes 2 levels per (full) iteration, so that the total number of levels consumed by Algorithm (7) is $2(\mu - 1)$, bringing the number of levels required to evaluate the entire process to $2 + 2(\mu - 1)$.

Algorithm 7 Homomorphic evaluation of μ iterations of Gradient Descent for ridge regression

Input: Ciphertexts $\mathtt{ct}.Y$, $\{\mathtt{ct}.M_j\}_{j=0,...,d}$, plaintexts $\{\mathtt{pt}.I_j\}_{j=0,...,d}$ and $\mathtt{pt}.\lambda$ which corresponds to the message $(0, \lambda, \lambda, ..., \lambda)$, and an iteration number μ
Output: A ciphertext $\mathtt{ct}.\beta$, corresponding to the values $(\beta_0, \beta_1, ..., \beta_d)$ after μ Gradient Descent updates.

$\mathtt{ct}.\beta \leftarrow \mathtt{ct}.Y$
for $i = 2, ..., \mu$ **do**
  $\mathtt{ct}_{\text{grad}} \leftarrow \mathtt{ct}.Y$
  for $j = 0, ..., d$ **do**
    $Z \leftarrow \text{RS}(\text{Mult}(\mathtt{ct}.\beta, \mathtt{ct}.M_j))$
    $Z \leftarrow \text{AllSum}(Z)$
    $Z \leftarrow \text{RS}(\text{Mult}(\mathtt{pt}.I_j, Z))$
    $\mathtt{ct}_{\text{grad}} \leftarrow \text{Subtract}(\mathtt{ct}_{\text{grad}}, Z)$
  end for
  $Z \leftarrow \text{RS}(\text{Mult}(\mathtt{ct}.\beta, \mathtt{pt}.\alpha))$
  $Z \leftarrow \text{RS}(\text{Mult}(Z, \mathtt{pt}.\lambda))$
  $\mathtt{ct}.\beta \leftarrow \text{Subtract}(\mathtt{ct}.\beta, Z)$
  $\mathtt{ct}.\beta \leftarrow \text{Subtract}(\mathtt{ct}.\beta, \mathtt{ct}_{\text{grad}})$
end for

If instead we wish to perform Nesterov's Accelerated Gradient Descent, we require an extra multiplication each iteration. We recall that Nesterov's Accelerated Gradient Descent maintains a momentum term $v^{(t)} \in \mathbb{R}^{(d+1)}$, and has updates given by [48]:

$$\beta^{(t+1)} = v^{(t)} - \alpha_t \nabla J(v^{(t)}) \tag{13}$$

$$v^{(t+1)} = (1 - \gamma_t)\beta^{(t+1)} + \gamma_t \beta^{(t)} \tag{14}$$

where $0 \leq \gamma_t \leq 1$ is a smoothing parameter. We observe that the update given by the first line is simply the Gradient Descent update applied to v in place of β, and so can be accomplished with Algorithm 7. For the second line of the update, we can use SIMD techniques to update every component of v simultaneously, giving:

$$\mathtt{ct}.v \leftarrow \text{RS}(\text{Mult}(\mathtt{pt}.(1-\gamma_t), \mathtt{ct}.\beta^{(t+1)})) + \text{RS}(\text{Mult}(\mathtt{pt}.\gamma_t, \mathtt{ct}.\beta^{(t)})) \tag{15}$$

This increases the number of levels required per iteration from 2 to 3. In our implementation we use the FISTA schedule for γ_t [7] which gives $\gamma_0 = 0$, so that we can initialise both $\mathtt{ct}.\beta \leftarrow \mathtt{ct}.Y$ and $\mathtt{ct}.v \leftarrow \mathtt{ct}.Y$. This brings the total number of levels required for Nesterov's Accelerated Gradient Descent training to $2 + 3(\mu - 1)$. For reasons of space, we omit details of the full algorithm giving the homomorphic evaluation of μ iterations of Nesterov's Accelerated Gradient Descent for ridge regression.

5.2 Fixed-Hessian

Having described how to achieve ridge regression training using either Gradient Descent or Nesterov's Accelerated Gradient Descent, which both require a learning rate α, we now propose a fixed-Hessian method for ridge regression training, which draws heavily from the results of Lindsay and Böhning [11] and Bonte and Vercauteren [14]. To make our work speak more naturally to theirs we replace the minimisation of (2) with the (equivalent)

maximisation of

$$F(\beta) = -\frac{1}{2}(\lambda \sum_{i=1}^{d} \beta_i^2 + \sum_{i=1}^{n} (y_i - \beta^T x_i)^2),$$

which has gradient vector given component wise by

$$\frac{\partial F}{\partial \beta_j} = -(\lambda \beta_j \mathbb{1}_{j\neq 0} + \sum_{i=1}^{n} x_{ij}(\beta^T x_i - y_i)).$$

Therefore, the full Hessian $H(\beta) \in \mathbb{R}^{(d+1)\times(d+1)}$ is given element wise by

$$H(\beta)_{jk} = -\left(\lambda \mathbb{1}_{j=k, j\neq 0} + \sum_{i=1}^{n} x_{ij}x_{ik}\right).$$

This Hessian is already "fixed", but as in Section 3 we simplify our computation by replacing H with a diagonal matrix $\tilde{H} \in \mathbb{R}^{(d+1)\times(d+1)}$ with diagonal elements given by

$$\tilde{H}_{kk} = \lambda \mathbb{1}_{k\neq 0} + \sum_{j=0}^{d} \sum_{i=1}^{n} x_{ij}x_{ik}. \tag{16}$$

We assume all entries x_{ij} are scaled to the range $[0, 1]$ so that by [14, Lemma 1] we have $-H \geq -\tilde{H}$. Then, by [11, Theorem 4.1], using $\tilde{H}$ to perform Newton-Raphson updates gives us linear monotonic convergence. However, unlike for logistic regression, our cost function F is not bounded above, so we lose the guaranteed convergence property in [11].

Substituting the full Hessian for our diagonal matrix with entries given by (16), updates are now given component wise by:

$$\beta_j \leftarrow \beta_j + \frac{1}{H_{j,j}} \nabla F(\beta), \tag{17}$$

where the right hand side of Equation (17) is given by

$$(1 - \frac{\lambda \mathbb{1}_{k\neq 0}}{H_{j,j}}) + \frac{1}{H_{j,j}} \left(\underbrace{\sum_{i=1}^{n} y_i x_{ij}}_{\tilde{Y}_j} - \sum_{k=0}^{d} \beta_k \underbrace{\sum_{i=1}^{n} x_{ij}x_{ik}}_{\tilde{M}_{jk}} \right).$$

Comparing this to Y_j and M_{jk} as given in (12), we see that the only difference is the factor of α, so that we can homomorphically evaluate a ciphertext $\mathtt{ct}.\tilde{Y}$ which corresponds to $(\tilde{Y}_0, ..., \tilde{Y}_d)$ and ciphertexts $\{\mathtt{ct}.\tilde{M}_j\}_{j=0,...,d}$ where $\mathtt{ct}.\tilde{M}_j$ corresponds to $(\tilde{M}_{j0}, ..., \tilde{M}_{jd})$ using Algorithms 5 and 6 respectively, simply substituting $\mathtt{pt}.I_j$ in place of $\mathtt{pt}.C_j$.

To homomorphically evaluate the entries of H given in Equation (16), we first rearrange the entries $\tilde{H}_{kk}$ as $\tilde{H}_{kk} = \lambda \mathbb{1}_{k\neq 0} + \sum_{j=0}^{d} \tilde{M}_{jk}$, so that the entries $\mathtt{ct}.\tilde{H}_{kk}$ can be easily evaluated as the homomorphic sum of the $(d+1)$ ciphertexts $\mathtt{ct}.M_{jk}$ as generated in lines 4 and 10 in Algorithm 6 and a plaintext $\mathtt{pt}.\lambda$. We can therefore calculate $(d+1)$ ciphertexts $\mathtt{ct}.H_j$ which each correspond to $(\tilde{H}_{jj}, ..., \tilde{H}_{jj})$ using as part of the calculation of $\mathtt{ct}.\tilde{M}_j$. In particular this process consumes no extra levels.

To invert these entries, we call Algorithm 2, observing that the scaling of the features x_{ij} enables us to bound the values $\tilde{H}_{kk}$ to the range $[1, n(d+1)]$, so that we can take

$$T_1 = \frac{8(1+n(d+1))}{1+8n(d+1)+n^2(d+1)^2}, \quad T_2 = \frac{-1}{1+8n(d+1)+n^2(d+1)^2}.$$

At this point, we have $(d+1)$ ciphertexts $\mathtt{ct}.H_j$ which each correspond to (an approximation of) $(\frac{1}{\tilde{H}_{jj}}, ..., \frac{1}{\tilde{H}_{jj}})$, using $2 + 2\kappa$ levels. To save time and levels when updating the weights β, we now use plaintext multiplication by $\mathtt{pt}.I_j$ to create $(d+1)$ ciphertexts $\mathtt{ct}.h_j$, where h_j has $\frac{1}{\tilde{H}_{jj}}$ in the j^{th} slot and zeroes elsewhere, as well as a ciphertext $\mathtt{ct}.H$ which corresponds to $(\frac{1}{\tilde{H}_{00}}, \frac{1}{\tilde{H}_{11}}, ..., \frac{1}{\tilde{H}_{dd}})$, enabling us to use SIMD operations.

Observe that if we initialise the parameters to zero, we have that the first iteration is given simply by $\beta_j = \frac{1}{\tilde{H}_{jj}} \tilde{Y}_j$. We outline how to perform μ fixed-Hessian updates to the parameters β in Algorithm 8, which consumes $1 + 2(\mu - 1)$ levels, bringing the total level consumption to $1 + 2(\mu + \kappa + 1)$. As before, we parallelise the loop given by lines 6 - 10.

Algorithm 8 Homomorphic Evaluation of μ iterations of fixed-Hessian Updates for Ridge Regression

Input: Ciphertexts $\mathtt{ct}.Y, \mathtt{ct}.H, \{\mathtt{ct}.h_j\}_{j=0,...,d}, \{\mathtt{ct}.M_j\}_{j=0,...,d}$, a plaintext $\mathtt{pt}.\lambda$ corresponding to $(0, \lambda, ...\lambda)$ and an iteration number μ

Output: A ciphertext $\mathtt{ct}.\beta$ which corresponds to $(\beta_0, \beta_1, ..., \beta_d)$ after μ fixed-Hessian updates

1: $\mathtt{ct}.Y \leftarrow \text{RS}(\text{Mult}(\mathtt{ct}.H, \mathtt{ct}.Y))$
2: $\mathtt{ct}.\beta \leftarrow \mathtt{ct}.Y$
3: $\mathtt{pt}.\lambda \leftarrow \text{RS}(\text{Mult}(\mathtt{pt}.\lambda, \mathtt{ct}.H))$
4: **for** $k = 2, ..., \mu$ **do**
5: $\quad \mathtt{ct}_{\text{grad}} \leftarrow \mathtt{ct}.Y$
6: $\quad$ **for** $j = 0, ..., d$ **do**
7: $\quad\quad Z \leftarrow \text{RS}(\text{Mult}(\mathtt{ct}.\beta, \mathtt{ct}.M_j))$
8: $\quad\quad Z \leftarrow \text{AllSum}(Z)$
9: $\quad\quad Z \leftarrow \text{RS}(\text{Mult}(\mathtt{pt}.h_j, Z))$
10: $\quad\quad \mathtt{ct}_{\text{grad}} \leftarrow \text{Subtract}(\mathtt{ct}_{\text{grad}}, Z)$
11: $\quad$ **end for**
12: $\quad Z \leftarrow \text{RS}(\text{Mult}(\mathtt{ct}.\beta, \mathtt{pt}.\lambda))$
13: $\quad \mathtt{ct}.\beta \leftarrow \text{Subtract}(\mathtt{ct}.\beta, Z)$
14: $\quad \mathtt{ct}.\beta \leftarrow \text{Add}(\mathtt{ct}.\beta, \mathtt{ct}_{\text{grad}})$
15: **end for**
16:

5.3 Implementation and comparison

We implemented ridge regression training homomorphically using all three minimisation algorithms in SEAL. In order to evaluate their effectiveness, we used the Boston Housing dataset available through scikit-learn [54] which consists of 506 records with 13 covariates. The goal is to predict the median house price for a region from 13 characteristics of the neighbourhood [36]. By using the closed form for the parameters β, this dataset achieves a maximum cross validation (CV) r^2 of about 0.7.

5.3.1 Parameter selection. For each minimisation approach we adopt the SEAL recommended parameter set for ring dimension

Descent	$\log \triangle$	μ	Time	r^2
GD	40	9	207.27	0.4165
	30	13	450.53	0.4058
NAD	40	6	127.71	0.4054
	30	9	415.54	0.4566
FH	40	6	137.68	0.3206

Table 2: Comparison of a fixed-Hessian minimisation (FH) for ridge regression to a Gradient Descent (GD) approach and a Nesterov's Accelerated Gradient Descent (NAD) approach. All rows correspond to parameter settings targetting 128-bit security and in all cases a feature-by-feature encoding is used. The column 'μ' denotes the number of iterations. The column 'Time' gives the 5 fold average training time in seconds. The column 'r^2' gives the average 5-fold CV coefficient of determination. All computations were carried out in the same manner as Table 1.

$N = 2^{15}$ to achieve an estimated 128-bit security according to the Homomorphic Encryption Standard [2]. This fixes a maximum bit length of ciphertext modulus Q_L. On the other hand, the number of iterations μ, and the requirement to decrypt with a certain precision, imposes a minimum bit length of Q_L. We set μ to the maximum value permitted by the security analysis. To implement μ iterations of Gradient Descent and decrypt at 10 bits of precision, we require a minimum bit length of Q_L of $2(\log \triangle + 10) + 2\mu \log \triangle$. For Nesterov's Accelerated Gradient Descent, a minimum bit length of Q_L is given by $2(\log \triangle + 10) + \log \triangle(2 + 3(\mu - 1))$. For our fixed-Hessian method, we set $\kappa = 2$ iterations of Newton-Raphson to invert the entries of $\tilde{H}$ (Algorithm 2). Then, a lower bound on the minimum bit length of Q_L is given by $2(\log \triangle + 10) + \log \triangle(1 + 2(\kappa + \mu + 1))$.

We set $\log \triangle = 40$ and tested all three methods at this precision. Using such a high scale is necessary for the fixed-Hessian method due to the very small values being operated on, which risk being lost at lower precision. We are additionally able to present results for a lower precision for the Gradient Descent methods.

For all three minimisation methods we set the regularisation parameter $\lambda = 1$.

Unlike for logistic regression, we used a fixed learning rate α for both Gradient Descent variants. We did not observe a better rate of convergence when trying either harmonic or anneal learning rates. For Gradient Descent, we chose $\alpha = 0.00125$, while for Nesterov's Accelerated Gradient Descent we chose $\alpha = 0.00099$, and again set the smoothing parameters γ_t according to the FISTA framework.

5.3.2 Results and discussion. In Table 2 we present the results of our implementation. Given the maximum CV r^2 for this dataset is around 0.7, we see that the accuracy achieved by all three minimisation methods is quite disappointing. This may be a consequence of how slowly the ridge regression cost function converges to a minimum: for example, plaintext experiments give that, for example, even in 50 iterations, GD still only exhibits an average CV accuracy of 65%. This means that good accuracy cannot be attained in the very limited number of iterations afforded to us by (leveled) CKKS.

We found suitable learning rates α for both Gradient Descent variants via plaintext experiments. We found that these learning rates had a huge impact on the accuracy of the derived model: choosing α too small led to very slow convergence as expected, while choosing α even 10^{-4} too large led to the parameters growing uncontrollably in magnitude. We suspect this behaviour occurs for ridge regression as its loss function is not Lipschitz continuous, unlike logistic regression. Hence, although Table 2 shows worse results for the fixed-Hessian method, we remark that for even slightly different choices of α the situation would be reversed. Given the difficulties associated with choosing this value securely, we would still suggest a fixed-Hessian method is better for outsourced training.

5.4 Future work

Our results for the three iterative approaches that we examined for ridge regression give convergence that is either unstable or slow. It would therefore be interesting to instead homomorphically evaluate the closed form solution to ridge regression directly.

Table 2 also gives an interesting insight on CKKS precision and noise. Comparing the two parameter settings for Gradient Descent, even though decreasing the precision enabled us to perform 4 more iterations, the accuracy of the derived model actually fell slightly. This suggests that the speed of convergence of Gradient Descent was insufficient to overcome the noise being accrued in the lower bits of the ciphertext at low precision. Future work could seek to make these observations precise, for example by comparing the rate of precision lost in CKKS operations and the rate of precision gained each iteration.

Along similar lines, it would also be helpful to have a stronger understanding of the noise growth in CKKS, both as a result of encoding and operations, particularly in the context of these minimisation methods. Such results could be helpful not only in the choice of $\log \triangle$ but also in the judicious use of bootstrapping.

Acknowledgements. This work was supported by EPSRC grants EP/S021817/1 and EP/P009301/1 and by the European Union Horizon 2020 Research and Innovation Program Grant 780701.

REFERENCES

[1] A. Akavia, H. Shaul, M. Weiss, and Z. Yakhini. 2019. Linear-Regression on Packed Encrypted Data in the Two-Server Model. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 21–32.

[2] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan. 2018. *Homomorphic Encryption Security Standard*. Technical Report. HomomorphicEncryption.org.

[3] M. R. Albrecht, R. Player, and S. Scott. 2015. On the concrete hardness of Learning with Errors. *J. Mathematical Cryptology* 9, 3 (2015), 169–203.

[4] Y. Aono, T. Hayashi, L. Phong, and L. Wang. 2016. Scalable and Secure Logistic Regression via Homomorphic Encryption. 142–144. https://doi.org/10.1145/2857705.2857731

[5] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca. 2016. A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes. In *SAC 2016 (LNCS)*, Roberto Avanzi and Howard M. Heys (Eds.), Vol. 10532. Springer, Heidelberg, 423–442. https://doi.org/10.1007/978-3-319-69453-5_23

[6] J. Barzilai and J. M. Borwein. 1988. Two-point step size gradient methods. *IMA journal of numerical analysis* 8, 1 (1988), 141–148.

[7] A. Beck and M. Teboulle. 2009. A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems. *SIAM Journal on Imaging Sciences* 2, 1 (2009), 183–202. https://doi.org/10.1137/080716542

[8] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. 2016. New directions in nearest neighbor searching with applications to lattice sieving. In *27th SODA*, Robert Krauthgamer (Ed.). ACM-SIAM, 10–24. https://doi.org/10.1137/1.9781611974331.ch2

[9] M. Blatt, A. Gusev, Y. Polyakov, K.Rohloff, and V. Vaikuntanathan. 2019. Optimized Homomorphic Encryption Solution for Secure Genome-Wide Association Studies. Cryptology ePrint Archive, Report 2019/223. (2019).

[10] F. Boemer, Y. Lao, R. Cammarota, and C. Wierzynski. 2019. nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*. 3–13.

[11] D. Böhning and B. Lindsay. 1988. Monotonicity of quadratic-approximation algorithms. *Annals of the Institute of Statistical Mathematics* 40 (02 1988), 641–663.

[12] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth. 2017. Practical Secure Aggregation for Privacy-Preserving Machine Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1175–1191.

[13] Charlotte Bonte, Carl Bootland, Joppe W. Bos, Wouter Castryck, Ilia Iliashenko, and Frederik Vercauteren. 2017. Faster Homomorphic Function Evaluation Using Non-integral Base Encoding. In *CHES 2017 (LNCS)*, Wieland Fischer and Naofumi Homma (Eds.), Vol. 10529. Springer, Heidelberg, 579–600. https://doi.org/10.1007/978-3-319-66787-4_28

[14] C. Bonte and F. Vercauteren. 2018. Privacy-preserving logistic regression training. *BMC medical genomics* 11, Suppl 4 (October 2018), 86. https://doi.org/10.1186/s12920-018-0398-y

[15] J. Bos, K. Lauter, and M. Naehrig. 2014. Private Predictive Analysis on Encrypted Medical Data. *Journal of biomedical informatics* 50 (05 2014). https://doi.org/10.1016/j.jbi.2014.04.003

[16] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. 2018. Fast Homomorphic Evaluation of Deep Discretized Neural Networks. In *CRYPTO 2018, Part III (LNCS)*, Hovav Shacham and Alexandra Boldyreva (Eds.), Vol. 10993. Springer, Heidelberg, 483–512. https://doi.org/10.1007/978-3-319-96878-0_17

[17] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha. 2019. Low Latency Privacy Preserving Inference *(Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, Long Beach, California, USA, 812–821.

[18] A. Brutzkus, R. Gilad-Bachrach, and O. Elisha. 2019. Low latency privacy preserving inference. In *International Conference on Machine Learning*. 812–821.

[19] H. Chen, R. Gilad-Bachrach, K. Han, Z. Huang, A. Jalali, K. Laine, and K. Lauter. 2018. Logistic regression over encrypted data from fully homomorphic encryption. *BMC Medical Genomics* 11, 4 (10 2018). https://www.microsoft.com/en-us/research/publication/logistic-regression-over-encrypted-data-from-fully-homomorphic-encryption-2/

[20] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. 2018. Bootstrapping for Approximate Homomorphic Encryption. Cryptology ePrint Archive, Report 2018/153. (2018).

[21] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. 2018. A full RNS variant of approximate homomorphic encryption. In *International Conference on Selected Areas in Cryptography*. Springer, 347–368.

[22] J. H. Cheon, A. Kim, M. Kim, and Y. Song. 2016. Homomorphic Encryption for Arithmetic of Approximate Numbers. Cryptology ePrint Archive, Report 2016/421. (2016).

[23] J. H. Cheon, D. Kim, Y. Kim, and Y. Song. 2018. Ensemble Method for Privacy-Preserving Logistic Regression Based on Homomorphic Encryption. *IEEE Access* 6 (2018), 46938–46948. https://doi.org/10.1109/ACCESS.2018.2866697

[24] J. L. H. Crawford, C. Gentry, S. Halevi, D. Platt, and V. Shoup. 2018. Doing Real Work with FHE: The Case of Logistic Regression. In *Proceedings of the 6th Workshop on Encrypted Computing and Applied Homomorphic Cryptography (WAHC '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3267973.3267974

[25] B. R. Curtis and R. Player. 2019. On the Feasibility and Impact of Standardising Sparse-secret LWE Parameter Sets for Homomorphic Encryption. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2019, London, UK, November 11-15, 2019*, Michael Brenner, Tancrède Lepoint, and Kurt Rohloff (Eds.). ACM, 1–10.

[26] J. E. Dennis Jr and R. B. Schnabel. 1996. *Numerical methods for unconstrained optimization and nonlinear equations*. SIAM.

[27] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. (2012). http://eprint.iacr.org/2012/144.

[28] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *41st ACM STOC*, Michael Mitzenmacher (Ed.). ACM Press, 169–178. https://doi.org/10.1145/1536414.1536440

[29] I. Giacomelli, S. Jha, M. Joye, C. D. Page, and K. Yoon. 2018. Privacy-preserving ridge regression with only linearly-homomorphic encryption. In *International Conference on Applied Cryptography and Network Security*. Springer, 243–261.

[30] T. Graepel, K. Lauter, and M. Naehrig. 2012. ML confidential: Machine learning on encrypted data. In *International Conference on Information Security and Cryptology*. Springer, 1–21.

[31] W. W. Hager and H. Zhang. 2005. A new conjugate gradient method with guaranteed descent and an efficient line search. *SIAM Journal on optimization* 16, 1 (2005), 170–192.

[32] S. Halevi and V. Shoup. 2013. Design and implementation of a homomorphic-encryption library. *IBM Research (Manuscript)* 6 (2013), 12–15.

[33] S. Halevi and V. Shoup. 2014. Algorithms in HElib. In *Annual Cryptology Conference*. Springer, 554–571.

[34] R. Hall, S. E. Fienberg, and Y. Nardi. 2011. Secure multiple linear regression based on homomorphic encryption. *Journal of Official Statistics* 27, 4 (2011), 669.

[35] K. Han, S. Hong, J. H. Cheon, and D. Park. 2018. Efficient Logistic Regression on Large Encrypted Data. *IACR Cryptology ePrint Archive* 2018 (2018), 662.

[36] D. Harrison Jr and D. L. Rubinfeld. 1978. Hedonic housing prices and the demand for clean air. (1978).

[37] S. Hu, Q. Wang, J. Wang, S. S. M. Chow, and Q. Zou. 2016. Securing fast learning! Ridge regression over encrypted big data. In *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 19–26.

[38] C. Huang and A. Mintz. 1990. Ridge regression analysis of the defence-growth tradeoff in the United States. *Defence and Peace Economics* 2, 1 (1990), 29–37.

[39] R. L. Kennedy, H. S. Fraser, L. N. McStay, and R. F. Harrison. 1996. Early diagnosis of acute myocardial infarction using clinical and electrocardiographic data at presentation: derivation and evaluation of logistic regression models. *European heart journal* 17, 8 (1996), 1181–1191.

[40] A. Kim, Y. Song, M. Kim, and J. H. Lee, K.and Cheon. 2018. Logistic regression model training based on the approximate homomorphic encryption. *BMC medical genomics* 11, 4 (2018), 83.

[41] M. Kim, Y. Song, S. Wang, Y. Xia, and X. Jiang. 2018. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR medical informatics* 6, 2 (2018), e19.

[42] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. 2013. A Toolkit for Ring-LWE Cryptography. Cryptology ePrint Archive, Report 2013/293. (2013). http://eprint.iacr.org/2013/293.

[43] X. Ma, F. Zhang, X. Chen, and J. Shen. 2018. Privacy preserving multi-party computation delegation for deep learning in cloud computing. *Information Sciences* 459 (2018), 103–116.

[44] D. W. Marquardt and R. D. Snee. 1975. Ridge regression in practice. *The American Statistician* 29, 1 (1975), 3–20.

[45] P. Mishra, R. Lehmkuhl, A. Srinivasan, W. Zheng, and R. A. Popa. 2020. Delphi: A Cryptographic Inference Service for Neural Networks. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2505–2522.

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

[47] K. P. Murphy. 2012. *Machine learning: a probabilistic perspective*. MIT press.

[48] Y. Nesterov. A method of solving a convex programming problem with convergence rate $O(1/k^2)$. In *Sov. Math. Dokl*, Vol. 27.

[49] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh, and N. Taft. 2013. Privacy-preserving ridge regression on hundreds of millions of records. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 334–348.

[50] J. Nocedal and S. Wright. 2006. *Numerical optimization*. Springer Science & Business Media.

[51] S. F. Obermann and M. J. Flynn. 1997. Division algorithms and implementations. *IEEE Trans. Comput.* 46, 8 (1997), 833–854.

[52] Tabitha Ogilvie, Rachel Player, and Joe Rowell. 2020. FHLR: Fixed Hessian Logistic Regression. (2020). https://github.com/TabOG/Fixed-Hessian-Logistic-Regression Available at https://github.com/TabOg/Fixed-Hessian-Logistic-Regression.

[53] Tabitha Ogilvie, Rachel Player, and Joe Rowell. 2020. PPRR: Privacy Preserving Ridge Regression. (2020). https://github.com/TabOG/Privacy-Preserving-Ridge-Regression Available at https://github.com/TabOg/Privacy-Preserving-Ridge-Regression.

[54] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[55] B. Price. 1977. Ridge regression: Application to nonexperimental data. *Psychological Bulletin* 84, 4 (1977), 759.

[56] Oded Regev. 2005. On lattices, learning with errors, random linear codes, and cryptography. In *37th ACM STOC*, Harold N. Gabow and Ronald Fagin (Eds.). ACM Press, 84–93. https://doi.org/10.1145/1060590.1060603

[57] M. Georgieva J. R. Troncoso-Pastoriza S. Carpov, N. Gama. 2020. Privacy-preserving semi-parallel logistic regression training with fully homomorphic encryption. *BMC medical genomics* 13 (2020), 88. https://doi.org/10.1186/s12920-020-0723-0

[58] M. J. Schulte, J. Omar, and E. E. Swartzlander. 1994. Optimal initial approximations for the Newton-Raphson division algorithm. *Computing* 53, 3-4 (1994), 233–242.

[59] SEAL 2020. Microsoft SEAL (release 3.5). https://github.com/Microsoft/SEAL. (April 2020). Microsoft Research, Redmond, WA.

[60] E. Setakis, H. Stirnadel, and D. J. Balding. 2006. Logistic regression protects against population structure in genetic association studies. *Genome research* 16, 2 (2006), 290–296.

[61] R. Shokri and V. Shmatikov. 2015. Privacy-Preserving Deep Learning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1310–1321.

[62] I. Theodossiou. 1998. The effects of low-pay and unemployment on psychological well-being: a logistic regression approach. *Journal of health economics* 17, 1 (1998), 85–104.

[63] H. J. P. Timmermans. 1981. Multiattribute shopping models and ridge regression analysis. *Environment and Planning A* 13, 1 (1981), 43–56.

[64] S. Truex, N. Baracaldo, A. Anwar, T. Steinke, H. Ludwig, R. Zhang, and Y. Zhou. 2019. A hybrid approach to privacy-preserving federated learning. In *Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security*. 1–11.

[65] Panayiotis N. V., D. Eastwood, H. J. Yun, M. V. Spanaki, L. H. Bey, C. Kessaris, and T. A. Gennarelli. 2006. Impact of a neurointensivist on outcomes in patients with head trauma treated in a neurosciences intensive care unit. *Journal of Neurosurgery JNS* 104, 5 (2006), 713 – 719.

[66] P. Xie, M. Bilenko, T. Finley, R. Gilad-Bachrach, K. Lauter, and M. Naehrig. 2014. Crypto-nets: Neural networks over encrypted data. *arXiv preprint arXiv:1412.6181* (2014).