Chen *et al. BMC Medical Genomics* 2018, **11**(Suppl 4):81
https://doi.org/10.1186/s12920-018-0397-z

BMC Medical Genomics

RESEARCH Open Access

Logistic regression over encrypted data from fully homomorphic encryption

Hao Chen[1], Ran Gilad-Bachrach[1], Kyoohyung Han[2], Zhicong Huang[3], Amir Jalali[4], Kim Laine[1]* and Kristin Lauter[1]

From iDASH Privacy and Security Workshop 2017
Orlando, FL, USA. 14 October 2017

Abstract

Background: One of the tasks in the 2017 iDASH secure genome analysis competition was to enable training of logistic regression models over encrypted genomic data. More precisely, given a list of approximately 1500 patient records, each with 18 binary features containing information on specific mutations, the idea was for the data holder to encrypt the records using homomorphic encryption, and send them to an untrusted cloud for storage. The cloud could then homomorphically apply a training algorithm on the encrypted data to obtain an encrypted logistic regression model, which can be sent to the data holder for decryption. In this way, the data holder could successfully outsource the training process without revealing either her sensitive data, or the trained model, to the cloud.

Methods: Our solution to this problem has several novelties: we use a multi-bit plaintext space in fully homomorphic encryption together with fixed point number encoding; we combine bootstrapping in fully homomorphic encryption with a scaling operation in fixed point arithmetic; we use a minimax polynomial approximation to the sigmoid function and the 1-bit gradient descent method to reduce the plaintext growth in the training process.

Results: Our algorithm for training over encrypted data takes 0.4–3.2 hours per iteration of gradient descent.

Conclusions: We demonstrate the feasibility but high computational cost of training over encrypted data. On the other hand, our method can guarantee the highest level of data privacy in critical applications.

Keywords: Cryptography, Homomorphic encryption, Logistic regression

Background

Since 2014, iDASH (integrating Data for Analysis, Anonymization, and Sharing) has hosted yearly international contests around the theme of genomic and biomedical privacy. Teams from around the world participate to test the limits of secure computation on genomic and biomedical tasks, and benchmark solutions on real data sets. Such contests serve to bring together experts in security, cryptography, and bioinformatics to quickly make progress on interdisciplinary challenges. The task for outsourced storage and computation this year was to implement a method for private outsourced training of a logistic regression model.

Motivation

Machine Learning (ML) over encrypted data has important applications for cloud security and privacy. It allows sensitive data, such as genomic and health data, to be stored in the cloud in encrypted form without losing the utility of the data. For the third task in the 2017 iDASH Secure Genome Analysis Competition, participants were challenged to train a machine learning model on encrypted genomic data that will predict disease based on a patient's genome. In a non-interactive (with outsourced storage) setting, training ML models on encrypted data had up until now only been done for very simple models, such as Linear Means Classifiers and Fisher's Linear Discriminant Analysis [1]. Interactive settings, where multiple parties hold shares of the data and communicate throughout the training process, have been

*Correspondence: kim.laine@microsoft.com
[1] Microsoft Research, Redmond, WA USA
Full list of author information is available at the end of the article

developed for several more complicated models, but they require high communication costs and a non-colluding assumption between several clouds [2].

The 2017 iDASH competition task was to train a logistic regression model, and although in theory it can be done using Fully Homomorphic Encryption (FHE) [3, 4], until now the feasibility and efficiency of this approach had not been studied.

Summary of results

In this work, we show that training a logistic regression model over binary data is possible using FHE. In particular, we use gradient descent and stochastic gradient descent algorithms with mini-batches, and demonstrate that it takes several minutes to one hour to run each gradient descent step. Our solution can run for an arbitrary number of steps, as opposed to the now commonly used practical homomorphic encryption (PHE) approach [5], where the size of the computation is determined beforehand, and parameters chosen once and for all to support a computation of that size. This is possible using Craig Gentry's bootstrapping operation [4], which we have implemented for the first time for the Fan-Vercauteren scheme [6] using the publicly available homomorphic encryption library SEAL (http://sealcrypto.org; accessed on 9 April, 2018).

More precisely, in fully homomorphic encryption each ciphertext contains a component called the noise, which grows in all homomorphic operations, and eventually reaches a maximum value. Once this maximum is reached, the ciphertext cannot be decrypted correctly anymore. Bootstrapping is the process of "refreshing" FHE ciphertexts to reduce the noise levels during deep computations to ensure correct decryption at the end of the computation.

Another challenge in the approach we take is the plaintext data type supported by the homomorphic encryption scheme. Namely, it is only possible to encrypt fairly small integers with SEAL, and indeed with many homomorphic encryption schemes. In machine learning the model weights are typically rational numbers, which need to be scaled to integers. Unfortunately, this quickly causes an overflow to occur in our rather small integer data type, unless the integers can be scaled down. We describe a modified bootstrapping operation which merges bootstrapping and such a scaling into one step, significantly reducing the complexity of our algorithm.

Besides noise growth and message expansion, another challenge in implementing Logistic Regression with FHE is applying the sigmoid function. We present two methods to approximate this function with a polynomial, and compare them both in terms of the accuracy of the trained model and in terms of computation time.

Related work

At the time of writing this, very little directly comparable prior work exists. The closest to our approach is [7], where the authors achieve remarkably good performance in training small logistic regression models; in their solution it is necessary that the number of features is very small (logarithmic in the number of training records).

A slightly different approach is taken in [8], where the authors use the homomorphic encryption library HEAAN, that natively supports scaling down of plaintext numbers [9, 10]. The authors report good performance numbers, but unlike us and [7] they only allow a very small number of iterations. Extending to more iterations will be computationally very costly, and require bootstrapping.

Methods

Fan-Vercauteren scheme

Fully Homomorphic Encryption (FHE) refers to a type of encryption scheme, envisioned already a few decades ago [3], that allows arbitrary computations to be performed directly on encrypted data. A blueprint for a solution was first proposed by Gentry [4] in 2009, and since then numerous schemes have been proposed. In this work we use the Fan-Vercauteren scheme (FV) [6], and its implementation in the SEAL library [11].

Parameters and notation.

We start by defining the parameters of the FV scheme. Let $q \gg t$ be positive integers and n a power of 2; often t is a prime such that $2n \mid (t-1)$. Denote $\Delta = \lfloor q/t \rfloor$. We define $R = \mathbb{Z}/(x^n+1)$, $R_q = \mathbb{Z}_q[x]/(x^n+1)$, and $R_t = \mathbb{Z}_t[x]/(x^n+1)$. Here, $\mathbb{Z}$ is the set of polynomials with integer coefficient and $\mathbb{Z}_q[x]$ is the set of polynomials with integer coefficient in range $[0, q-1)$. Therefore, R_q is the set of polynomials of degree at most $n-1$, with coefficients integers modulo q. Multiplications of polynomials in R_q is similar to usual polynomial multiplication, except that x^n should in every step be replaced by -1. In the FV scheme plaintext elements are polynomials in R_t, and ciphertext elements are pairs of polynomials in $R_q \times R_q$. Let χ denote a narrow (centered) discrete Gaussian error distribution. In practice, most implementations of homomorphic encryption use $\sigma[\chi] \approx 3.2$. Finally, let U_k denote the uniform distribution on $\mathbb{Z} \cap [-k/2, k/2)$.

Key generation

The first step in using the FV scheme is generating a public-secret key pair $(\mathtt{pk}, \mathtt{sk})$. To do this, sample $s \leftarrow U_3^n$, $a \leftarrow U_q^n$, and $e \leftarrow \chi^n$; here s, a, and e are all considered as elements of R_q, where the n coefficients are sampled independently from the given distributions. To form the keys, we let

$$\mathtt{pk} = ([-(as+e)]_q, a) \in R_q^2,$$

where $[\cdot]_q$ denotes the (coefficient-wise) reduction modulo q. In reality there are other types of keys involved, in particular so-called evaluation keys and Galois keys, but for the sake of simplicity we will omit discussing them here, and refer the reader to [6, 11].

Encryption.

Let $m \in R_t$ be a plaintext message. To encrypt m with the public key $\texttt{pk} = (p_0, p_1) \in R_q^2$, sample $u \leftarrow U_3^n$, $a \leftarrow U_q^n$, and $e_1, e_2 \leftarrow \chi^n$. Consider u, a, and e_i as elements of R_q as in key generation, and create the ciphertext

$$\texttt{ct} = ([\Delta m + p_0 u + e_1]_q, [p_1 u + e_2]_q) \in R_q^2 .$$

Decryption.

To decrypt a ciphertext $\texttt{ct} = (c_0, c_1)$ given a secret key $\texttt{sk} = s$, write

$$\frac{t}{q}(c_0 + c_1 s) = \widehat{m} + v + bt ,$$

where $c_0 + c_1 s$ is computed as an integer coefficient polynomial, and scaled by the rational number t/q, b is an integer coefficient polynomial, $\widehat{m}$ the underlying message, and v the leftover fractional part.

It is easy to see that when q is sufficiently larger than t, then $\widehat{m} = m$, and $\|v\|_\infty \ll 1/2$. This means that the original message can be recovered by computing

$$m = \left\lfloor \frac{t}{q}(c_0 + c_1 s) \right\rceil_t ,$$

where $\lfloor \cdot \rceil$ denotes rounding to the nearest integer. For details, see [6, 11].

Homomorphic computations

A final fundamental piece in the puzzle is how to enable additions and multiplications of two ciphertexts. For addition, this is easy; we define an operation $\oplus$ between two ciphertexts $\texttt{ct}_1 = (c_0, c_1)$ and $\texttt{ct}_2 = (d_0, d_1)$ as follows:

$$\texttt{ct}_1 \oplus \texttt{ct}_2 = ([c_0 + d_0]_q, [c_1 + d_1]_q) \in R_q^2 .$$

We denote this homomorphic sum by $\texttt{ct}_{\text{sum}} = (c_0^{\text{sum}}, c_1^{\text{sum}})$, and note that if

$$\frac{t}{q}(c_0 + c_1 s) = m_1 + v_1 + b_1 t , \quad \frac{t}{q}(d_0 + d_1 s) = m_2 + v_2 + b_2 t ,$$

then

$$\frac{t}{q}(c_0^{\text{sum}} + c_1^{\text{sum}} s) = [m_1 + m_2]_t + v_1 + v_2 + b_{\text{sum}} t ,$$

as long as $\|v_1 + v_2\|_\infty < 1/2$. Thus, $\oplus$ passes through the encryption to the underlying plaintexts, and results in an encryption of the sum $[m_1 + m_2]_t$ as long as $\|v_1 + v_2\|_\infty < 1/2$.

It is similarly possible to define an operation $\otimes$ between two ciphertexts, that results in a ciphertext decrypting to $[m_1 m_2]_t$, as long as $\|v_1\|_\infty$ and $\|v_2\|_\infty$ are small enough. Since $\otimes$ is much more difficult to describe than $\oplus$, we refer the reader to [6, 11] for details.

Noise

In the decryption formulas presented above the rational coefficient polynomials v are assumed to have small enough infinity-norm, namely less than 1/2. This is clearly necessary, as otherwise the ciphertext will result in the incorrect plaintext being recovered. Given a ciphertext $\texttt{ct} = (c_0, c_1)$ encrypting a plaintext m, let $v \in \mathbb{Q}[x]/(x^n + 1)$ such that

$$\frac{t}{q}(c_0 + c_1 s) = m + v + bt .$$

The polynomial v is called the noise polynomial, $\|v\|_\infty$ is called the noise, and the ciphertext decrypts correctly as long as the noise is less than 1/2 [11].

When operations such as addition and multiplication are applied to encrypted data, the noise in the result may be larger than the noise in the inputs; this is referred to as noise growth. This noise growth is very small in homomorphic additions, but substantially larger in homomorphic multiplications. Thus, given a specific set of encryption parameters (n, q, t, χ), one can only evaluate computations of a bounded size (in practice, of bounded multiplicative depth), until the noise grows too large making the ciphertext impossible to decrypt even with the correct secret key.

To mitigate the problem of high noise growth rates Craig Gentry [4] described a clever approach which is commonly known as bootstrapping. In this process, an encrypted version of the secret key is used to decrypt the message using homomorphic operations. Therefore, the result of this process is similar to a freshly encrypted message and hence it has only a small amount of noise. This bootstrapping process is considered to be a very costly operation in most schemes [10, 12, 13], but not in all [14, 15].

Batching

The FV scheme (and many other homomorphic encryption schemes) inherently support SIMD operations. This capability is commonly called “batching” in literature, and is explained in detail e.g. in [11] in the context of the SEAL library that we use.

The idea is that by choosing the plaintext modulus t appropriately, the plaintext space R_t is isomorphic as a ring to the k-fold product $\mathbb{F}_{t^{n/k}} \times \ldots \times \mathbb{F}_{t^{n/k}}$, for some $k \mid n$. In other words, operations in R_t translate automatically into k concurrent operations in the extension field $\mathbb{F}_{t^{n/k}}$, for example allowing us to perform k-fold SIMD operations on integers up to t by using only the subfield $\mathbb{Z}_t \subset \mathbb{F}_{t^{n/k}}$.

Using batching efficiently can be non-trivial, and typically requires one to carefully design the computation to maximize the benefit.

Logistic regression

Logistic Regression is a common tool used in machine learning to build a model that can discriminate between samples from two or more classes. It arises from the need to model the posterior probabilities of K classes via linear functions of input $x \in \mathbb{R}^D$. In this work we consider two-class classification, so $K = 2$. To simplify the notation, we assume the input vector x always has 1 as the first element, which accounts for the bias term in the linear function. Then the logistic regression model has the form

$$\log \left[\frac{\Pr(Y = 0 \mid X = x)}{\Pr(Y = 1 \mid X = x)} \right] = w^T x \,,$$

where Y denotes the class, and $w \in \mathbb{R}^D$ is the weight vector that we need to learn in model training. The above model is specified in terms of log-odds ratio, reflecting the constraint that the probabilities sum to one. An alternative and more common form is to represent it as the following posterior probability for class 0:

$$\Pr(Y = 0 \mid X = x) = \frac{1}{1 + e^{-w^T x}} = \sigma(w^T x) \,,$$

where $\sigma(t) = 1/(1 + e^{-t})$ is known as the sigmoid function. Next we present two algorithms for learning w.

Training algorithms

Our goal is to evaluate a training algorithm for a logistic regression model on homomorphically encrypted data. In this section we present the two training algorithms that we evaluated for this purpose.

Gradient descent

The standard method for training logistic regression is gradient descent. To fix notation, let D be the number of (binary) features, and N the number of training records of the form (X, y), where $X \in \mathbb{R}^{N \times D}$, $y \in \mathbb{R}^N$. In this case the weight vector w is in $\mathbb{R}^D$.

Gradient descent proceeds in iterations, where in each iteration the weight vector w is updated as

$$w \leftarrow w - \alpha(\sigma(Xw) - y)X^T \,,$$

where σ is the sigmoid function, and $\alpha > 0$ a learning rate parameter. We formalize the gradient descent algorithm below.

Algorithm 1 Gradient Descent for Logistic Regression

```
Require: $X \in \mathbb{R}^{N \times D}, y \in \mathbb{R}^N, \alpha > 0$
Ensure: $w \in \mathbb{R}^D$
Initialize weight vector: $w \leftarrow 0$
for iter in $[0, T)$ do
    for $i$ in $[0, N)$ do
        $V_i \leftarrow \langle \vec{X}_i, w \rangle$
        $U_i \leftarrow \sigma(V_i)$
    end for
    for $j$ in range $[0, D)$ do
        $g_j \leftarrow \sum_i (U_i - y_i) X_{ij}$
        $w_j \leftarrow w_j - \alpha g_j$
    end for
end for
```

1-bit gradient descent

A direct application of Algorithm 1 suffers from the problem of quickly growing plaintext size—a problem which was briefly mentioned in "Summary of results". Namely, the parameter t in the homomorphic encryption scheme is typically quite small, causing integer plaintext data to quickly become reduced modulo t. This is similar to the problem using a too small data type in normal programming, except that in this case it is difficult to switch to a larger one. For this reason, we need to be able to control the growth of our encrypted numbers either by scaling them down, and/or by designing our computation in a way that minimizes the increase in the size of the numbers.

For the first approach, we need a homomorphic floor function, which we discuss in "Fixed point arithmetic over plaintext data". For the second approach, we note that multiplying by just a sign never increases the size of a number, so replacing one multiplicand by its sign allows the plaintext size to remain much smaller. Unfortunately, homomorphic sign extraction is very difficult, but turns out to be still faster than the homomorphic floor function. For this reason, we opt to use sign information instead of evaluating floor function to make our homomorphic training faster. By using the 1-Bit Gradient Descent (1-Bit GD) algorithm, which was invented to compress the gradient in order to reduce communication during training [16], our homomorphic training becomes much faster.

In the 1-Bit GD method, in each iteration we update each weight by a learning rate multiplied by the sign of the corresponding coordinate of the current gradient, plus a residue term. The unused part of the gradient is then added back into the residue. We also introduce a new parameter β, which reduces the magnitude of the accumulated residues in the past. Our modified 1-Bit GD is presented formally in Algorithm 2.

Algorithm 2 Modified 1-Bit Gradient Descent for Logistic Regression

```
Require: $X \in \mathbb{R}^{N \times D}, y \in \mathbb{R}^N, \alpha > 0, \beta > 0$
Ensure: $w \in \mathbb{R}^D$
Initialize weight vector $w \leftarrow 0$; Initialize residue vector $r \leftarrow 0$.
for iter in $[0, T)$ do
    for $i$ in $[0, N)$ do
        $V_i \leftarrow \langle \vec{X}_i, w \rangle$
        $U_i \leftarrow \sigma(V_i)$
    end for
    for $j$ in range $[0, D)$ do
        $g_j \leftarrow \sum_i (U_i - y_i) X_{ij}$
        $r_j \leftarrow \beta \cdot r_j + g_j$
        (Extract sign) $\text{sign} = 1 \text{ if} r_j > 0 \text{ else} - 1$
        $w_j \leftarrow w_j - \alpha \cdot \text{sign}$
        $r_j \leftarrow r_j - \alpha \cdot \text{sign}$
    end for
end for
```

The 1-Bit GD approach can be done easily also in the stochastic setting, where either individual records or mini-batches are processed at a time. In this work, for the sake of simplicity, we will only focus on full gradient descent.

Fixed point arithmetic

Fixed point arithmetic over plaintext data

Logistic Regression is naturally performed over floating point numbers. However, in the FV scheme there is no easy way to encrypt numbers of this type directly, so they need to be first scaled to integers of some fixed precision.

In fixed point number representation we choose an integer base p (in this work we will fix p to be an odd prime), the number of integral digits l, and the number of fractional digits f. Then a fixed point number is a rational number x of the form

$$x = \sum_{i=-f}^{l-1} x_i p^i ,$$

with $x_i \in [-(p-1)/2, \ldots, (p-1)/2] \cap \mathbb{Z}$. That is, every fixed point number has l integral digits and f fractional digits in base p. We need f extra digits to hold an intermediate result from multiplication, hence we let $r = l + 2f$ and set the modulus to be p^r (see also [17]). To encode a number, we multiply by p^f and round to an integer, i.e. the representation of x is $\tilde{x} = p^f x$. See

To add/subtract two fixed numbers, we simply add/subtract their representations modulo p^r. To multiply two fixed point numbers x and y, we compute

$$\tilde{z} = \left\lfloor \frac{\tilde{x}\tilde{y} \pmod{p^r}}{p^f} \right\rceil .$$

Note that although standard fixed point arithmetic requires us to perform scaling after every multiplication, it is not strictly needed. For example, if we are going to compute $\sum_{i=1}^n x_i y_i$, then it is possible to not scale after each product, but only scale after the sum. This may not save a lot of work over plaintext, since scaling is fast; however, since scaling is expensive over encrypted data, this technique is useful in our setting.

Bootstrapping

Even for relatively small examples, Algorithm 1 and Algorithm 2 result in (multiplicatively) high-depth arithmetic circuits; the depth is equal to the number of iterations times the depth of a single iterative step. Recalling the noise growth problem discussed above in "Noise", a straightforward implementation will have to use bootstrapping regularly to maintain the correctness of the final result. Since bootstrapping is a costly operation, we introduce below in "Combining bootstrapping with scaling" a modification to this step that does both the noise cleaning and also scaling, which is used to prevent plaintext size expansion.

We modified the bootstrapping algorithm from [13], where the crucial part of the bootstrapping procedure is a homomorphic digit removal process. Namely, suppose the plaintext modulus of our homomorphic encryption scheme is a prime power $t = p^r$, and the plaintext is (for simplicity) just an integer $m \in \mathbb{Z}_{p^r}$. Then as an intermediate result in bootstrapping we have an encryption of $M = p^{e-r}m + v$, where $e > r$, p^e is an intermediate plaintext modulus, and $|v| < p^r/2$ is the noise to be removed. If we have a polynomial which removes the lowest $e - r$ digits in an integer modulo p^e, then applying it to M will give us $p^{e-r}m$, which is a scalar multiple of the original message. In the FV scheme the scalar multiple can be easily removed when the plaintext modulus is divided by the scalar value. So the bootstrapping procedure finishes by removing the scalar value. Below in "Combining bootstrapping with scaling" we apply these ideas to achieve bootstrapping together with scaling down of encrypted numbers, resulting in encrypted fixed point arithmetic.

Combining bootstrapping with scaling

In order to perform the scaling functionality over encrypted data, we need to express the functionality as a polynomial. This is possible, however the polynomial will often have large degree, forcing us to perform bootstrapping to refresh the noise after each scaling over encrypted data. It turns out that these two steps can be combined for improved performance.

Suppose we have an encryption of a message m modulo p^r, and we wish to obtain an encryption of $\lfloor m/p^i \rceil$.

First, we can apply a free division operation in FV (see e.g. [17]) to obtain an encryption of $\lfloor m/p^i \rfloor + p^{r-1}\alpha$ with full noise, where α represents some "upper garbage". Then we perform modulus-switching followed by a dot product with the bootstrapping key (see e.g. ([13], Section 4.1)) to obtain a low-noise encryption of $v + p^{e-r}\lfloor m/p^i \rfloor + p^{e-i}\alpha \pmod{p^e}$, with $|v| \leq p^{e-r}/2$. Then we follow the bootstrapping algorithm and homomorphically evaluate a polynomial of degree ep^{e-r} to remove the v term. Finally, we apply one extra step to remove the α term. This can be done in a similar fashion, by evaluating a digit removal polynomial of degree rp^{r-i}. As a result, we obtain an encryption of $\lfloor m/p^i \rfloor$. We will use FHE.bscale$(\cdot, i)$ to denote the above bootstrapping plus scaling down by i digits in base p. For convenience of notation, we set the default value of i to be 1. The total degree of the procedure is $ep^{e-r} \cdot rp^{r-i} = erp^{e-i}$.

Results

In this section we describe experiments with the techniques described in previous sections.

Dataset description

We used two datasets to test the performance of our homomorphic machine learning algorithm.

iDASH 2017 competition dataset

The dataset provided by the iDASH competition organizers consists of 1579 training samples, where each sample contains a binary phenotype (cancer/no cancer), and 108 binary genotypes. In the evaluation of the solution, the organizers selected 18 genotypes to use as the features and therefore, in the experiments reported below only these 18 features were used.

MNIST dataset

The MNIST dataset [18] consists of hand written digits, stored as images, and it is commonly used as benchmark for machine learning systems. Each image in the original dataset is a 28 × 28 pixel map, where each pixel is represented in a 256 level gray-scale code. We first selected 1500 images containing handwritten digits '3' and '8' to obtain a binary classification problem. Then we compressed each image into 196 features with each feature an integer in the range $[0, 8)$, by dividing each pixel value by 32 and performing average pooling with window of size 2 × 2.

Parameter selection

Selecting the right parameters can make a big difference in performance in terms of speed, space, and accuracy. Here we described the parameter tuning performed in the experiments.

FHE parameters

The FHE parameters need to be chosen carefully in order to achieve correctness, security, and performance. There are three crucial FHE parameters to be chosen: the ring dimension n, the ciphertext modulus q and the plaintext modulus t.

Smaller n and q imply better speed, while in order to support bootstrapping and scaling operations n and q need to be sufficiently large. In our experiments we chose $n = 2^{15}$ and $q \approx 2^{1020}$, as these parameters are just large enough for bootstrapping and scaling, yet as small as possible for optimal performance. More precisely, we chose q as a product of 17 primes—each 60 bits in size—as required by SEAL. These parameters guarantee around 100 bits of security.

The value of t determines the precision of our computation: the larger t is, the more correct digits we will expect to see in the result. On the other hand, if t is too large, bootstrapping and scaling cannot be supported unless we also increase the value of q. We chose to use $t = p^r = 127^3$ to balance between precision and performance. This configuration supports 64 slots per ciphertext (recall "Batching", and see "Data batching method" below).

ML parameters

We use two training algorithms. The first one uses a linear approximation of the sigmoid function together with 1-bit GD (Algorithm 2), while the second algorithm uses a degree 3 approximation of the sigmoid function and normal gradient descent (Algorithm 1). Note that we chose to use a linear approximation of the sigmoid function in the 1-bit GD method, because there is no need to use higher degree approximation due to only the sign being considered. For the iDASH dataset we let the training algorithm perform 36 iterations over the training data, while for the MNIST dataset we perform 10 iterations. For the iDASH dataset, the learning parameters were set to $\alpha = 0.1$ and $\beta = 0.2$ for Algorithm 2, and $\alpha = 0.0002$ for Algorithm 1. For the MNIST data set, we used $\alpha = 0.01$ and $\beta = 0.2$ for Algorithm 2, and $\alpha = 10^{-5}$ for Algorithm 1.

Approximating the sigmoid function

There are several methods to find an approximate polynomial for a given function. The best known method is probably Taylor polynomials, but it minimizes the error only in the vicinity of one point. For this reason, we instead use an approach similar to [19], and use a so-called minimax approximation.

Let P_d denote the set of polynomials of degree at most d, and for a continuous function $f \in C[a, b]$ denote $\|f\| = \max\{|f(x)| : x \in [a, b]\}$.

Definition 1 *$p \in P_d$ is a d-th minimax approximation of $f \in C[a,b]$ if*

$$\|f - p\| = \inf\{\|f - q\| : q \in P_d\}.$$

For more details, we refer the reader to [20].

A minimax approximation algorithm (or uniform approximation) is a method to find the polynomial p in the above definition. The Remez algorithm [21] is an iterative minimax approximation algorithm, and yields the following results for the interval $[-5,5]$ and degrees 1 and 3:

$$\sigma_1(x) = 0.125x + 0.5\,, \quad \sigma_3(x) = -0.004x^3 + 0.197x + 0.5\,.$$

These functions are illustrated in Fig. 1 and Fig. 2, respectively.

Data batching method

In order to efficiently use the batching capabilities in SEAL (recall "Batching"), we encode the training dataset "vertically", i.e. each ciphertext will store one single genotype/phenotype from k samples, where k is the number of slots in one plaintext. For example, the FHE parameters presented above in "Parameter selection" yield $k = 64$ slots. On the other hand, we will need D plaintexts to represent the weights, where within each plaintext vector the weight is repeatedly encoded k times. As a result, the data matrix X is encoded into a $\lceil N/k \rceil \times D$ matrix $\mathcal{X}$ of plaintexts, and the vector of labels y is encoded into a vector $\mathcal{Y}$ of plaintexts. These plaintexts are then encrypted and sent to the untrusted party (e.g. cloud service), which performs the homomorphic training computation, resulting in an encrypted logistic regression model. The gradient descent training algorithm over encrypted data (Algorithm 3) is presented below.

Algorithm 3 Gradient Descent over encrypted data using batching

```
Require: $\mathcal{X}, \mathcal{Y}$
Ensure: $\mathcal{W}$
  Initialize $\mathcal{W} = (\mathrm{Enc}(\mathbf{0}), \ldots, \mathrm{Enc}(\mathbf{0}))$
  for iter in range $[1, T]$ do
    for i in range $[0, N')$ do
      $V[i] :=$ FHE.innerproduct($\mathcal{X}[i]$, $\vec{W}$)
      $U[i] =$ FHE.evalPoly*($V[i]$, $\sigma_3$)
      $U'[i] =$ FHE.sub($\mathcal{Y}[i]$, $U[i]$)
    end for
    for $j$ in range $[0, D)$ do
      $\Delta[j] =$ FHE.innerproduct($U'$, $\mathcal{X}[\cdot][j]$)
      $\Delta[j] =$ FHE.sumslots($\Delta[j]$)
      $\mathcal{W}[j] :=$ FHE.sum($\mathcal{W}[j]$, FHE.plainmult*($\alpha$, $\Delta[j]$))

    end for
  end for
```

In Algorithm 3, we put a '*' after the evalPoly and plainmult functions to indicate that the corresponding functions are combined with the bootstrapping/scaling function bscale in order to emulate fixed point arithmetic. More details about evaluating σ_3 and multiplying α is in 'incorporating scaling' section below.

The only other place that requires further explanation is the FHE.sumslots function. The input to this function is a batched encryption of a vector $v = (v_0, v_1, \ldots, v_{k-1})$, and the output is an encryption of $v' = (\sum_i v_i, \ldots, \sum_i v_i)$. In general, this function can be implemented based on the slot rotation functionality. More precisely, our choice of FHE parameters guarantees that we can cyclically rotate the values in an encrypted vector. Note that the number of slots k is a divisor of the FHE parameter n, hence is always a power of 2. Let $k = 2^\ell$, and let FHE.rotate(c,j) denote the operation of cyclic rotation to the right by j slots, i.e., it

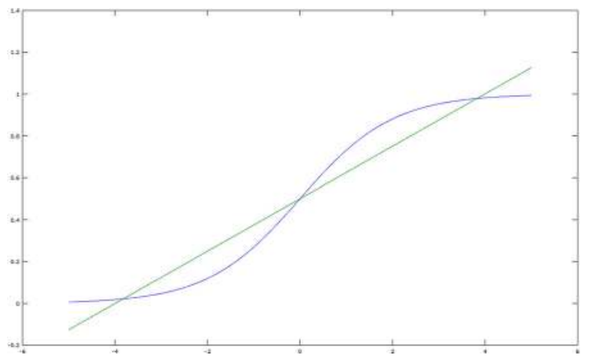

Fig. 1 Linear minimax approximate for sigmoid: $f(x) = 0.5 + 0.125x$

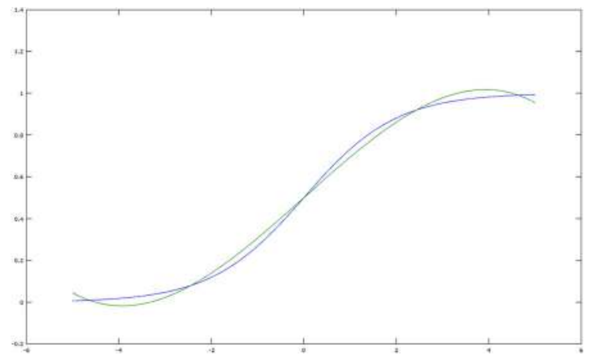

Fig. 2 Degree 3 minimax approximate for sigmoid: $f(x) = 0.5 + 0.197x - 0.004x^3$

sends an encryption of $(v_0, \dots, v_{k-1})$ to an encryption of $(v_{k-1}, v_0, \dots, v_{k-2})$. Then the FHE.sumslots function is as presented in Algorithm 4.

Algorithm 4 FHE.sumslots function

```
Require: c = Enc(v)
Ensure: an encryption of $(\sum_i v_i, \dots, \sum_i v_i)$.
$c' = c$
for i in range $[0, \ell)$ do
    $c' :=$ FHE.add($c'$, FHE.rotate($c', 2^i$)).
end for
return $c'$
```

Lemma 1 *Algorithm 4 is correct, i.e. the output c' is an encryption of $\Sigma v = (\sum_i v_i, \dots, \sum_i v_i)$.*

Proof Since $k = 2^\ell$, we have that the final result c' is equivalent to

$$\sum_{i=0}^{k-1} \mathsf{FHE.rotate}(c, i)\,.$$

The claim now follows, since the sum of all rotations of the vector v is exactly Σv. □

Optimization techniques

We introduce an optimization to further accelerate our implementation. In the last step of Algorithm 3, the FHE.plainmult operations (see [11]) needs to be performed D times. Although these operations themselves are fast, the accompanied homomorphic scaling is expensive. Therefore, we employ an optimization to reduce the number of multiplications from D to D/k. Since $\Delta[j]$ is an encryption of a constant vector, we can combine the content of k of those into one ciphertext, encrypting $(\delta_0, \dots, \delta_{k-1})$. Then multiplying this ciphertext by α would multiply the values in all slots, resulting in an encryption of $(\alpha\delta_0, \dots, \alpha\delta_{k-1})$. After the multiplication, we can "expand" the result back to k ciphertexts, each encrypting a constant vector of $\alpha\delta_i$. This expansion step can be implemented via FHE.sumslots. The precise algorithms FHE.combine and FHE.expand are introduced below.

Algorithm 5 FHE.combine

```
Require: A vector c of ciphertexts, where $c[i] = \mathrm{Enc}((\delta_i, \dots, \delta_i))$ for $0 \le i < k$
Ensure: An encryption of $(\delta_0, \dots, \delta_{k-1})$.
return FHE.innerproduct(c, e);
    Note: e[i] denotes a plaintext that is the batch encoding of the $i$-th unit vector.
```

Algorithm 6 FHE.expand

```
Require: Ciphertext $c$ which is an encryption of $(v_0, \dots, v_{k-1})$.
Ensure: A vector C of $k$ ciphertexts, where C[$i$] is an encryption of $(v_i, v_i, \dots, v_i)$.
for i in range $[0, \ell)$ do
    tmp = FHE.plainmult(c, e[i]);
    C[i] = FHE.sumslots(tmp) ;
end for
return C
```

Incorporating scaling

Some attention to details is needed since the arithmetic system uses fixed point representation.

Evaluating σ_3

Recall that $\sigma_3(x) = 0.5 + 0.197x - 0.004x^3$, and the representation uses fixed point arithmetic with $p = 127$, $l = 1$, $f = 1$. We will scale 0.5 to $\lfloor 0.5p^2 \rfloor = 8064$ with a scaling factor of 2. The second coefficient 0.197 will be scaled to $\lfloor 0.197p \rfloor = 25$. For the coefficient $0.004 \approx 0.063^2$, we scale 0.063 to $\lfloor 0.063p \rfloor = 8$. Then we can compute $\sigma_3(x)$ over encrypted data in the following way (recall that $\tilde{x} = \lfloor xp \rfloor$):

$$\sigma_3(x) \approx \mathsf{bscale}(8064 + 25\tilde{x} - \mathsf{bscale}(\mathsf{bscale}(8\tilde{x})^2) \cdot \tilde{x}).$$

Multiplying by learning rate

In the last step of each iteration of the training algorithms, the ciphertext is multiplied by the learning rate α. The challenge is that the learning rate we use ($\alpha = 0.002$) is so small that it can not be represented by the fixed point representation we use. To see this, note that we have $p = 127$ and $f = 1$, so the smallest positive number that can be represented is $1/127 \approx 0.008$. To resolve this issue, we start by writing $\alpha = (\sqrt{\alpha})^2$. Since $\sqrt{0.002} \approx 0.0447$, it can be represented by our fixed point system, as $[0.00447p] = 6$. Then we multiply the input by this value twice to obtain the result. After each multiplication, bscale is used to put the underlying number to correct scale. That is:

$$\alpha x \approx \mathsf{bscale}(6 \cdot \mathsf{bscale}(6 \cdot \tilde{x}))\,.$$

Sign extraction in 1-Bit GD

In order to implement the 1-Bit GD training algorithm, we need a function FHE.signExtract that homomorphically extracts the sign in a fixed point number. Fortunately, this function can be implemented using the bscale function as a subroutine. Since FHE ciphertexts encrypt scaled integers rather than point numbers, it suffices to extract the sign from an signed integer. Moreover, because the sign of an integer is just the most significant digit in its base-p expansion, we can extract it directly using $\mathsf{bscale}(\cdot, r-1)$.

Note that the total degree of this algorithm is erp^{e-r+1}, which is smaller than the usual fixed point scaling, which has degree erp^{e-f} . This advantage motivates the use of the 1-Bit GD algorithm in our work. The rest of the 1-Bit GD algorithm over encrypted data is exactly the same as Algorithm 3, hence we omit the details.

Performance results

Table 1 presents the performance results for the iDASH dataset, and Table 2 presents the performance numbers for a subset of the MNIST dataset containing only handwritten digits '3' and '8'. In both tables the performance of models trained on plaintext data using MATLAB are compared to models produced by training on encrypted data. We performed the experiments on an Intel(R) Xeon(R) CPU E3-1280 v5 @ 3.70GHz and 16GB RAM. Our experiments use only a single thread, although we note that some of the costliest parts of the computation would be easily parallelizable. We run the same training algorithms on both encrypted and unencrypted data, and compare the results. In order to evaluate the quality of the predictive models obtained, we run a 10-fold cross validation on both training sets, and compute the average Area Under the Curve (AUC) values. Since the unencrypted computation in MATLAB is several orders of magnitude faster than the encrypted computation (less than 1 second), we decided not to compare the unencrypted and encrypted running times side-by-side.

The algorithms, when operated on encrypted data, were able to obtain almost identical accuracy compared to training on unencrypted data. Obviously training on encrypted data is much slower than training on unencrypted data, which can be acceptable in some use-cases, and unacceptable in others; for the datasets that we used, training can take between half a day to few days, although substantial improvements in computational performance can be expected by improving our implementation, and extending it to use multiple threads.

Discussion

In this work we presented new ways to train Logistic Regression over encrypted data, which allow an arbitrary number of iterations due to FHE bootstrapping, thus making our models updatable once new data becomes available without requiring decryption at any point; this is different from other recently proposed approaches that limit the number of iterations in the training process. The time per iteration scales linearly with the data size. Hence, the total time for training N samples with D features per sample using T iterative steps over encrypted data is a linear function in the product $N \cdot D \cdot T$. Therefore, our solutions scale gracefully with the size of the data. Moreover, many of the ideas presented here can be used for training other machine learning models, for example Neural Networks, by using polynomial approximations to the activation functions.

Table 1 Running 10-fold cross-validation on the iDASH dataset with 1579 samples and 18 selected genotypes

Training method	# iterations	Avg. training time	Avg. AUC	Avg. AUC (unencrypted)
GD + σ_3	36	115.33 h	0.690	0.690
1-Bit GD + σ_1	36	14.90 h	0.668	0.690

The first average AUC value is obtained from running the training algorithm using SEAL on encrypted data. The second AUC value is obtained from running the same algorithm on unencrypted data using MATLAB

Table 2 Running 10-fold cross-validation on compressed MNIST dataset with 1500 samples and 196 features

Training method	# iterations	Avg. training time	Avg. AUC	Avg. AUC (unencrypted)
GD + σ_3	10	48.76 h	0.974	0.977
1-Bit GD + σ_1	10	27.10 h	0.974	0.978

The first average AUC value is obtained from running the training algorithm using SEAL on encrypted data. The second AUC value is obtained from running the same algorithm on unencrypted data using MATLAB

Conclusions

There is a growing interest in applying machine learning algorithm to private data, such as medical data, genomic data, financial data, and more. For critical applications homomorphic encryption can guarantee the highest level of data privacy during computation, but it also comes with a high cost, especially in terms of computation time.

Funding
The publication of this article is funded by Microsoft Corporation.

Availability of data and materials
The iDASH 2017 competition data was only available to registered competition participants. The MNIST dataset is publicly available (see [18]).

About this Supplement
This article has been published as part of BMC Medical Genomics Volume 11 Supplement 4, 2018: Proceedings of the 6th iDASH Privacy and Security Workshop 2017. The full contents of the supplement are available online at https://bmcmedgenomics.biomedcentral.com/articles/supplements/volume-11-supplement-4.

Authors' contributions
All authors contributed equally to this work. All authors have read and approved the manuscript.

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Author details
[1] Microsoft Research, Redmond, WA USA. [2] Seoul National University, Seoul, Korea. [3] École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland. [4] Florida Atlantic University, Boca Raton, USA.

Published: 11 October 2018

References

1. Graepel T, Lauter K, Naehrig M. ML confidential: Machine learning on encrypted data. In: International Conference on Information Security and Cryptology. Berlin: Springer; 2012. p. 1–21.
2. Mohassel P, Zhang Y. Secureml: A system for scalable privacy-preserving machine learning. In: Security and Privacy (SP), 2017 IEEE Symposium on. San Jose: IEEE; 2017. p. 19–38.
3. Rivest RL, Adleman L, Dertouzos ML. On data banks and privacy homomorphisms. Found Secure Comput. 1978;4(11):169–80.
4. Gentry C. A fully homomorphic encryption scheme. Stanford: Stanford University; 2009.
5. Lauter K, Naehrig M, Vaikuntanathan V. Can Homomorphic Encryption Be Practical? In: Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop. CCSW '11. New York: ACM; 2011. p. 113–124. Available from: http://doi.acm.org/10.1145/2046660.2046682.
6. Fan J, Vercauteren F. Somewhat Practical Fully Homomorphic Encryption. IACR Cryptol ePrint Arch. 2012;2012:144. https://eprint.iacr.org/2012/144. Accessed 9 Apr 2018.
7. Crawford JLH, Gentry C, Halevi S, Platt D, Shoup V. Doing Real Work with FHE: The Case of Logistic Regression; 2018. https://eprint.iacr.org/2018/202. Accessed 9 Apr 2018. Cryptology ePrint Archive, Report 2018/202.
8. Kim M, Song Y, Wang S, Xia Y. Jiang X; 2018. https://eprint.iacr.org/2018/074. Accessed 9 Apr 2018. Cryptology ePrint Archive, Report 2018/074.
9. Cheon JH, Kim A, Kim M, Song Y. Homomorphic encryption for arithmetic of approximate numbers. In: International Conference on the Theory and Application of Cryptology and Information Security. Cham: Springer; 2017. p. 409–437.
10. Cheon JH, Han K, Kim A, Kim M. Song Y; 2018. https://eprint.iacr.org/2018/153. Accessed 9 Apr 2018. Cryptology ePrint Archive, Report 2018/153.
11. Chen H, Han K, Huang Z, Jalali A. Laine K; 2017.
12. Halevi S, Shoup V. Bootstrapping for HElib. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. Berlin: Springer; 2015. p. 641–670.
13. Chen H. Han K; 2018. https://eprint.iacr.org/2018/067. Accessed 9 Apr 2018. Cryptology ePrint Archive, Report 2018/067.
14. Ducas L, Micciancio D. FHEW: bootstrapping homomorphic encryption in less than a second. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. Berlin: Springer; 2015. p. 617–640.
15. Chillotti I, Gama N, Georgieva M, Izabachene M. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In: International Conference on the Theory and Application of Cryptology and Information Security. Berlin: Springer; 2016. p. 3–33.
16. Seide F, Fu H, Droppo J, Li G, Yu D. 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech DNNs. In: Fifteenth Annual Conference of the International Speech Communication Association. Singapore; 2014.
17. Arita S. Nakasato S; 2016. https://eprint.iacr.org/2016/402. Accessed 9 Apr 2018. Cryptology ePrint Archive, Report 2016/402.
18. LeCun Y, Cortes C. Burges, CJC; 1998. http://yann.lecun.com/exdb/mnist/. Accessed 9 Apr 2018.
19. Cheon JH, Jeong J, Lee J, Lee K. Privacy-preserving computations of predictive medical models with minimax approximation and non-adjacent form. In: International Conference on Financial Cryptography and Data Security. Cham: Springer; 2017. p. 53–74.
20. Fraser W. A survey of methods of computing minimax and near-minimax polynomial approximations for functions of a single independent variable. J ACM (JACM). 1965;12(3):295–314.
21. Remez EY. Sur le calcul effectif des polynomes d'approximation de Tschebyscheff. CR Acad Sci Paris. 1934;199:337–40.