Lab - Machine learning on encrypted data: Logistic Regression implementation with CKKS

Van Thong Nguyen (VTN) Gerd Mund (GM) Yat Wai Wong (YWW)

December 3, 2021

Abstract

scheme

VTN

In homomorphic encryption world, the Cheon-Kim-Kim-Song (CKKS) is currently known as the most efficient homomorphic encryption scheme for approximate arithmetic. It allows approximate addition and multiplication of encrypted ~~messages~~ [Che+17]. In spite of the fact that th~~e encryption~~ scheme can achieve practical performance in some applications, we evaluate Logistic Regression model with the CKKS scheme against encryption performance, training performance, and accuracy of the output model.

Currently, there are two well-known libraries which are built on top of the CKKS scheme: HEAAN (C++), and TenSEAL (Python). We decide to use TenSEAL, because Python programming language is easier than C++ and it is popular in Data Science world. Firstly, we implement the ~~CKKS~~ Logistic Regression and run it on four randomly generated datasets:

- `framingham` (40000 9-D points),
- `LogReg_sample_dataset` (1000 2-D points),
- `HRF_samples_small` (1000 5-D datapoints),
- `HRF_samples_big`: (50000 5-D points).

Secondly, we run a ~~normal~~ Logistic Regression on those datasets. Then we compare the ~~normal~~ Logistic Regression and the CKKS Logistic Regression in term of performance, e.g., runtimes and ~~the~~ accuracy of the final results. In conclusion, we find that the CKKS Logistic Regression ~~model~~ achieves ~~the~~ similar accuracy ~~to~~ the normal Logistic Regression. However, the training CKKS encrypted data does take much more time, because the TenSEAL library ~~has~~ not yet supported encrypted matrix addition and multiplication at the time of the evaluation.

Besides the performance evaluation, we propose a solution to reduce the runtime and the amount of allocated memory drastically when encrypting the datasets with the CKKS scheme by packing multiple datapoints into a single datapoint. Finally, we also suggest a method to enable the CKKS Logistic Regression to deal with ~~the~~ packed datapoints.

fully
Inspired by ?
the
real or complex numbers.
Is
the
more accessible
ing
with CKKS
plain
plain
? training on encrypted data ?
Overcharized
/ms
ing
Hing
ing
than
does HE ing
HE
ing

Matrix multiplication is even more demanding!

Contents

1 Introduction

VTN

Nowadays, data encryption helps us with protecting our data from being leaked or stolen. However, at the very beginning of data encryption, it was impossible to ~~do evaluation over the~~ encrypted data. It raises a problem for companies and organizations which want to gain useful information from their customers' data and have to ensure the privacy regulations at the same time. This is where fully homomorphic encryption plays its part to enable arithmetic operations over the encrypted data. Therefore, the fully homomorphic encryption allow us to apply some machine learning algorithms to homomorphic encrypted data.

compute with

for example

In this report, we would like to use Cheon-Kim-Kim-Song homomorphic encryption scheme (CKKS) along with Logistic Regression to evaluate performance of training process and accuracy of generated models. Alternatively, Brakerski/Fan-Vercauteren homomorphic encryption scheme (BFV) [FV12] could be an option. Nevertheless, it can only do arithmetic calculations over integers, which means we need to transform the data before processing. In contrast to BFV, CKKS is designed to deal with ~~the~~ real numbers. As a result, we choose CKKS over BFV. The theory behind CKKS is explained in detail in ~~a scientific~~ paper *"Homomorphic Encryption for Arithmetic of Approximate Numbers"* [Che+17]. ~~So that, the report does not go too much into how CKKS works and does not prove the theory. Instead, the~~ report will answer following question: *How is the performance of Logistic Regression over CKKS encrypted datasets (i.e. How fast is it to train the CKKS encrypted data compared with training plaintext data? Does Logistic Regression over CKKS encrypted datasets consume more memory? How accurate is the final model of Logistic Regression over CKKS encrypted dataset?)?*

I prefer author year references.

the fully

the fully

the

Based on it, this

ns

than what?

In order to finalize the evaluation, four 2-class datasets ~~is encrypted by CKKS scheme~~:

are used for evaluating our implementation

- `framingham` (40000 9-D points),
- `LogReg_sample_dataset` (1000 2-D points),
- `HRF_samples_small` (1000 5-D datapoints),
- `HRF_samples_big` (50000 5-D points).

Then the encrypted datasets are trained by using gradient-descent Logistic Regression. Measurements are taken on a single machine to prevent bias and any inconsistencies.

In addition, we would like to propose a method to reduce amount of memory needed to encrypt the data by CKKS scheme. The method also saves network bandwidth, if the encrypted data needs to be transferred over the network, i.e., the internet.

2 Preliminaries

2.1 Cyclotomic Ring

YWW

Let $\Phi_M(X)$ be the M-th cyclotomic polynomial of degree N. Let $\mathcal{R} = \mathbb{Z}[X]/\Phi_M(X)$ be the integer ring and write $\mathcal{R}/q\mathcal{R}$ as the residue ring of $\mathcal{R}$ modulo an integer q.
In CKKS scheme, the polynomial with degree of power of 2 is used, and M is always ~~two times the~~ degree, i.e. the ring used in CKKS scheme is $\mathcal{R} = \mathbb{Z}[X]/(X^N+1)$ and $N = M/2$.
For a real polynomial $a \in \mathcal{R} = \mathbb{Z}[X]/\Phi_M(X)$, the canonical embedding $\sigma(a)$ are $\zeta_M^j \in \mathbb{C}^N$ where $j \in \mathbb{Z}_M^*$ and $\zeta_M = e^{2\pi i/M}$ is the primitive Mth root of unity.

the

a

a power-of-two degree ?

twice that

No.

$\sigma(a) = [a(\zeta_n^j)]_{0 \le j < M}$

\exp or \operatorname{exp}

2.2 Gaussian Distribution

For a real number r, we define the the Gaussian function as $\rho_r = exp(-\pi||\mathbf{z}||^2/r^2)$, where $||\mathbf{z}||$ is the L_2 norm of the vector $\mathbf{z}$. A discrete Gaussian distribution is usually used as the RLWE error distribution, the continu~~e~~ous Gaussian distribution has to be discretized by a rounding function. YWW

2.3 RLWE Problem

Which one?

Here we define the (decisional) RLWE problem as follows. We write $\mathcal{R}^\vee$ as the fractional ideal of $\mathcal{R}$ and $\mathcal{R}_q^\vee = \mathcal{R}^\vee/q\mathcal{R}^\vee$. For a secret $s \in \mathcal{R}_q^\vee$, let $q \geq 2$ be the modulus, $\mathbf{r} \in (\mathbb{R}^+)^N$ and an error distribution $\chi := \lfloor \Psi_\mathbf{r} \rceil_{\mathbb{R}^\vee}$, ~~a~~ an RLWE distribution $A_{N,q,\chi}(s)$ over $\mathcal{R}_q \times \mathcal{R}_q^\vee$ is sampled by choosing $a \leftarrow \mathcal{R}_q$ uniformly at random, $e \leftarrow \chi$, and output an RLWE instance as $(a, a \cdot s + e) \in \mathcal{R}_q \times \mathcal{R}_q^\vee$. YWW

Def? (Alternatively, make clear what it is on an intuitive basis and point to its def. elsewhere.)

$\mathbb{R}_{>0}$ (This notation is less ambiguous.)

The decisional RLWE problem is to distinguish between the samples drawn from the RLWE distribution $A_{N,q,\chi}(s)$ and the samples drawn uniformly at random from the distribution $\mathcal{R}_q \times \mathcal{R}_q^\vee$, with non-negligible probability.

Note that the form of RLWE sample is sometimes written as $(a, a \cdot s + e) \in \mathcal{R}_q \times \mathcal{R}_q$ because s and e can be transformed by multiplying them by a tweak factor t, such that $ts, te \in \mathcal{R}_q$, and $(a, a \cdot ts + te) \in \mathcal{R}_q \times \mathcal{R}_q$. As CKKS scheme uses a power-of-two cyclotomic ring, the dual ideal is $\mathcal{R}^\vee = N^{-1} \cdot \mathcal{R}$. Thus the tweak factor for s and e is just N. The RLWE problems in these two forms are entirely equivalent in terms of computation, but it turns out that $\mathcal{R}_q \times \mathcal{R}_q^\vee$ form is the right definition for hardness proof and cryptographic applications when using a (nearly) spherical error e [LPR10] [LPR13].

This is a technicality which I would only mention when necessary. So maybe in those proofs. But not here.

It seems that the dual ideal $R^\vee$ is a fractional ideal of R. But

2.4 Fully Homomorphic Encryption and Bootstrapping

Then you never need to talk about the dual ideal $R^\vee$.

(Fully h) ~~H~~omomorphic encryption is an encryption scheme that enables computation on encrypted data and produces encrypted results matching those of plaintext computations, without needing to decrypt the data or to know the decryption key. More precisely, one can take a set of encrypted message Enc(x_1), ..., Enc(x_n) and homomorphically compute a function on this set, i.e. produce an encryption of the function of them, that is, Enc$(f(x_1, ..., x_n))$. The homomorphic computation should be correct, i.e. YWW

$$\mathsf{Dec}(\overset{sk}{sk}, \mathsf{Enc}(\overset{pk}{}f(x_1, ..., x_n))) = f(x_1, ..., x_n).$$

⚠ Be consistent! There must be a key used in Dec and Enc, either as - say - first argument or in the index.

Homomorphic encryption schemes built from LWE usually have a problem of error growth from homomorphic computations, which makes the scheme being not able to compute functions correctly after the error grows to a certain size. These schemes can only homomorphically evaluate a circuit ~~with a~~ of bounded depth, and they are usually called 'somewhat homomorphic' or 'leveled' a.

A concept called 'bootstrapping' proposed by Gentry [Gen09] allows us to reduce the error rate of a ciphertext, thus enables homomorphic computation with unbounded depth. ~~Homomorphic~~ Such an encryption scheme with unbounded depth is called fully homomorphic encryption. Suppose we have a ciphertext C of the plaintext m, where its error rate is too large for ~~a~~ homomorphic ~~computation~~ a further operation. The idea behind bootstrapping is to encrypt the secret key, i.e. Enc(sk), then homomorphically evaluate the decryption function on C and Enc(sk). Thus, the homomorphic decryption function will produce a 'refreshed' ciphertext C', which is the encryption of m with a lower error rate:

a homomorphic

$$\mathsf{Dec}_{HE}(\mathsf{Enc}(sk), C) = C' = \mathsf{Enc}(m).$$

Notation? problematic, since Enc is not deterministic. Better: Dec..$(C') = m$.

* homomorphic encryption usually means schemes like RSA, which are homomorphic wrt. to one operation. Fully homomorphic = arbitrary operations, i.e. several op's and arbitrary depth.

2.5 Cheon-Kim-Kim-Song homomorphic encryption scheme

The CKKS scheme is a fully homomorphic encryption scheme for approximate arithmetic, which supports approximate addition and multiplication on real number with unbounded depth. Its security is based on the hardness to solve the ring learning with errors problem (RLWE), and the decryption of this scheme results in a small error on data, which is acceptable for some real world applications.

YWW

/ s

Batching and plaintext encoding. The idea of batching in FHE scheme is to batch multiple plaintexts into a single ciphertext, which allows us to perform homomorphic computation efficiently with parallel computation. As CKKS scheme relies on RLWE, the batched vectors must be encoded into polynomials contained in an ~~a~~ appropriate cyclotomic ring. The idea of encoding and decoding is to use canonical embedding, e.g. a plaintext vector $\mathbf{z} \in \mathbb{C}^{N/2}$ is encoded as a polynomial $m(X) \in \mathbb{Z}[X]/(X^N + 1)$ by the inverse of canonical embedding and vice versa.

the

a/the ?

⊔\@{}⊔

More precisely, the image of the canonical embedding $\sigma(a)$ is in the subring $\mathbb{H} = \{(z_j)_{j\in\mathbb{Z}_M^*} \in \mathbb{C}^N :$ $z_j = \overline{z_{-j}}\}$ of $\mathbb{C}^N$. As half of the elements in $\mathbb{H}$ are conjugate to the other half, the plaintext vectors $\mathbf{z}$ are in $\mathbb{C}^{N/2}$. To project from $\mathbb{C}^N$ to $\mathbb{C}^{N/2}$, a projection π is defined by $(z_j)_{j\in\mathbb{Z}_M^*} \mapsto (z_j)_{j \in T}$ where T is a multiplicative subgroup of $\mathbb{Z}_M^*$ satisfying $\mathbb{Z}_M^*/T = \{\pm 1\}$.

σ is the embedding. σ(a) is a value of it.

I prefer $\mathbb{Z}_M^\times$, since...

In the encoding algorithm, we first need to expand ~~the~~ vector $\mathbf{z} \in \mathbb{C}^{N/2}$ to $\mathbb{C}^N$ by using π^{-1} ~~and~~ $\pi^{-1}(\mathbf{z}) \in \mathbb{H}$. As $\sigma(\mathcal{R})$ is countable but $\mathbb{H}$ is isomorphic to $\mathbb{C}^{N/2}$ (defined by π) and uncountable, $\pi^{-1}(\mathbf{z})$ may not be in $\sigma(\mathcal{R})$. Thus we need to discretize $\pi^{-1}(\mathbf{z})$ by the coordinate-wise randomized rounding (written as $\lfloor\ \rceil_{\sigma(\mathcal{R})}$) [LPR13].

/ a

φ

As rounding in encoding process may destroy some significant figures, we need to multiply the plaintext by a scaling factor Δ before encoding, and divide it by Δ^{-1} in the decoding algorithm.

The full encoding and decoding algorithm are defined as follows:

This is a list! \begin{description} \item[Encoding] ... ... NEVER use \\ unless in row-oriented environments like array, tabular, gather, align, ...

Encoding: a vector $\mathbf{z} \in \mathbb{C}^{N/2}$ is taken as input. We first expand it to $\pi^{-1}(\mathbf{z}) \in \mathbb{H}$, then multiply it by the scaling factor Δ, followed by the rounding function. Finally, apply the inverse ~~of~~ canonical embedding σ^{-1}. The resulting polynomial is

/ φ

$$ m(X) = \sigma^{-1}(\lfloor \Delta \cdot \pi^{-1}(\mathbf{z}) \rceil_{\sigma(\mathcal{R})}) \in \mathcal{R}. $$

Decoding: a polynomial $m \in \mathcal{R}$ is taken as input. The resulting vector is

$$ z = \pi \circ \sigma(\Delta^{-1} \cdot m) \in \mathbb{C}^{N/2}. $$

Encryption and decryption. In the CKKS scheme, the encryption of a plaintext m outputs $(c_0, c_1) = (-a \cdot s + e + m, a) \mod q$, and the decryption of a ciphertext is $\mathsf{Dec}((c_0, c_1), s) = c_0 + c_1 \cdot s = (-a \cdot s + e + m) + a \cdot s = m + e \mod q$.

the key must

Bootstrapping. As the decryption algorithm consists of a modular reduction, the bootstrapping of CKKS scheme thus includes a homomorphic modular reduction. Since CKKS scheme is infeasible to do homomorphic modular reduction, Cheon et al. [Che+18] proposed a bootstrapping for CKKS scheme where the decryption formula is approximated by an approximate polynomial of a scaled sine function. A trigonometric function is a good approximation of modular reduction because it is the identity near ~~by~~ zero and periodic with period q.

the

/ φ

This occurs only in the first sentence of this subsection. :)

2.6 Logistic Regression

Logistic Regression is a common supervised machine learning model mainly for two-class classification. The logistic sigmoid function has the form YWW

$$\sigma(a) = \frac{1}{1+\exp(-a)}$$

⚠ Name collision with the embedding σ.

and the posterior probability for a class C_1 can be written as ?

∂?

$$p(C_1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})}$$

?

where $\mathbf{x}$ is the feature vector and $\mathbf{w}$ is the weight vector we need to learn in training phase. The term 'sigmoid' means S-shaped, as you can see in figure 1. This type of function is usually called 'squashing function' because it maps the whole real axis into a finite interval. As CKKS scheme can only evaluate polynomial function, the logistic sigmoid function

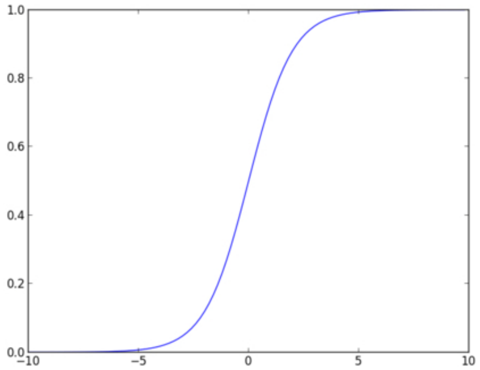


Figure 1: Plot of the logistic sigmoid function

can only by approximated by a specific polynomial. Refer to section 3.1 for details.

2.7 Gradient Descent

Which parameters? How, namely acc. to which goal? And why that goal? 3 ideas???

Logistic regression requires nonlinear optimization methods to estimate the regression parameters. The most well-known cost function optimization approaches in logistic regression are the Newton-Raphson method and the gradient descent method. Since matrix inversion is a must when using the Newton-Raphson method and the CKKS scheme does not support that feature, we put the Newton-Raphson method aside and consider the gradient descent method to train the encrypted datasets. Fortunately, the gradient descent method does not need matrix inversion and any division operations, as a result; it is suitable for training the CKKS encrypted data. VTN YWW /ing

The idea of gradient descent for logistic regression is as follows:
Maximum likelihood is used for adjusting the parameter of the logistic regression model.
We first need the derivative of the logistic sigmoid function:

$$\frac{d\cancel{\sigma}}{d\mathbf{w}} = \sigma(1-\sigma)\mathbf{x}$$

/ $\sigma(w^T\cdot)$

Argh!
You define $\sigma(a) = \frac{1}{1+\exp(-a)}$
but I guess this partial derivative refers to the function $(w,x) \mapsto \sigma(\langle w|x\rangle)$.

For a data set $\{x_n, t_n\}$, where $t_n \in \{0, 1\}$ is the target value, and $n = 1, ..., N$, the likelihood function is:

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n} \{1 - y_n\}^{1-t_n}$$

?

where $\mathbf{t} = (t_1, ...t_n)^T$ and $y_n = p(C_1|x_n)$.

We then take the negative logarithm of the likelihood function as the error function, and this give us a cross-entropy error function:

Braces denote sets! Only!

$$E(\mathbf{w}) = -\ln p(\mathbf{t}|\mathbf{w}) = \sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$

where $y_n = \sigma(\mathbf{w}^T \mathbf{x})$. Using the derivative above to take the gradient of the error function, we get

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N} (y_n - t_n) x_n$$

This gradient of the error function is used for the iterative update of the weight:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla E(\mathbf{w})$$

where $\alpha > 0$ is the learning rate.

I do not understand the idea behind these many ad-hoc calculations. They should be much more prominent than the calculations..

3 Related Work

In this section, we would like to introduce two papers: *Secure Logistic Regression Based on Homomorphic Encryption: Design and Evaluation* [Kim+18b], and *Logistic regression model training based on the approximate homomorphic encryption* [Kim+18a]. Both of the papers present ways to implement logisctic regression based on CKKS encryption scheme. VTN

the

3.1 Secure Logistic Regression Based on Homomorphic Encryption: Design and Evaluation

the

The authors adapted CKKS encryption scheme which is optimized for real number computation. Notably, a least squares approximation of the logistic function is introduced to improve accuracy and efficiency. VTN

We know that the sigmoid function plays an important part in Logistic Regression data training using the gradient descent method. Despite the gradient descent can be used along with CKKS encryption scheme, there is still a computational difficulty arises during the implementation. The sigmoid function becomes that issue, because the CKKS can only evaluate the polynomial functions. Therefore, Taylor polynomials have been formulated as an approximate replacement of the sigmoid function. However, the authors found that even if the Taylor polynomial $T_9(x)$, the accuracy is not enough; because it is a local approximation near a certain point (see figure 2a).

l R

two verbs?

we?

is used!

No reason.

That is inevitable!

It all depends on the range where the values x occurring in the final algorithm are that must be known in advance!

Due to limitation of the Taylor polynomial, Miran Kim at al. [Kim+18b] adopted a global approximation to minimize the the mean squared error which is defines by: $(1/|I|) \int_I f(x)^2 dx$, where $f(x)$ is an integrable function and I is an interval. The least

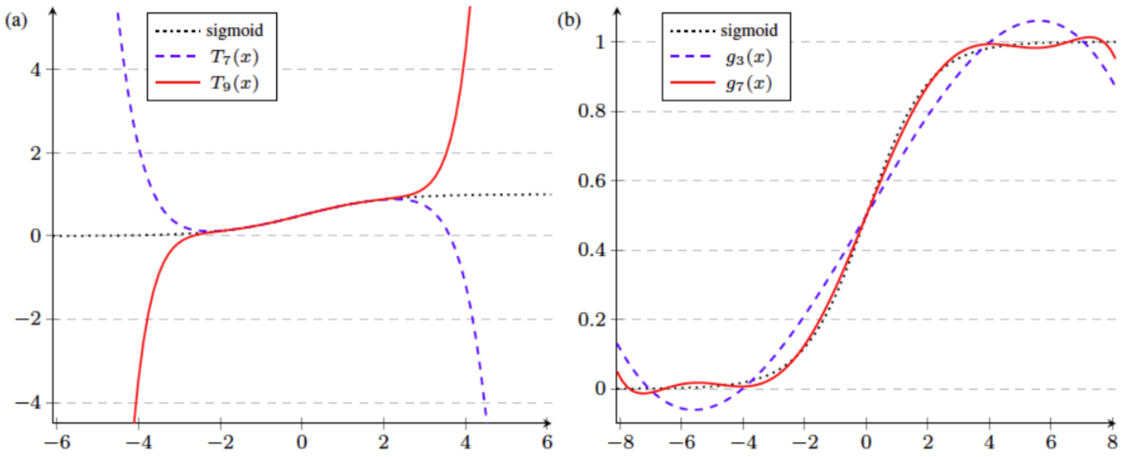


Figure 2: Graphs of (a) sigmoid function and Taylor polynomials and (b) sigmoid function and least squares approximations. [Kim+18b]

$$g_3(x) = 0.5 + 1.20096 * (x/8) - 0.81562 * (x/8)^3$$
$$g_7(x) = 0.5 + 1.73496 * (x/8) - 4.19407 * (x/8)^3 + 5.43402 * (x/8)^5 - 2.50739 * (x/8)^7$$

Figure 3: Degree 3 (g_3) and 7 (g_7) least squares approximation of the sigmoid function. [Kim+18b]

squares method is meant to produce a polynomial $g(x)$ of degree d minimizing the mean squared error $(1/|I|) \int_I (f(x) - g(x))^2 dx$. The authors used degree 3 and 7 least squares approximation of the sigmoid function over the interval [-8,8] (see formulas 3). It has been proved that g_3 only needs a smaller depth for evaluation, while g_7 is more precise (see figure 2b).

This explains what should have been clear in the previous paragraph.

Now you start with something outside the scope of the subsection, I think

In order to evaluate their method, five chosen datasets are described in table 1. These single-binary-labeled datasets can be used to train binary classifiers, e.g., logistic regression. In table 2, the final models of encrypted approach and encrypted logistic regression are compared. We see that the gaps between those models are not huge; therefore, the paper's approach is as good as the unencrypted logistic regression with the original sigmoid function.

Dataset	Number of observations	Number of features
Edinburgh Myocardial Infarction	1253	10
Low Birth Weight Study	189	10
Nhanes III	15,649	16
Prostate Cancer Study	379	10
Umaru Impact Study	575	9

Table 1: Description of datasets. [Kim+18b]

3.2 Logistic regression model training based on ~~the~~ approximate homomorphic encryption

Don't do that manually!

Similar to the paper of ~~Miran~~ Kim et al. [Kim+18b], Kim at al. [Kim+18a] also described a mean to train a logistic regression model without leaking any information by using the CKKS encryption scheme. Especially, they proposed a new encoding method to reduce the size VTN

Dataset and iteration number	Degree of $g(x)$	Our homomorphic encryption-based logistic regression		Unencrypted logistic regression	
		Accuracy	AUC	Accuracy	AUC
Edinburgh Myocardial Infarction					
25	3	86.03%	0.956	88.43%	0.956
20	7	86.19%	0.954	86.19%	0.954
Low Birth Weight Study					
25	3	69.30%	0.665	68.25%	0.668
20	7	69.29%	0.678	69.29%	0.678
Nhanes III					
25	3	79.23%	0.732	79.26%	0.751
20	7	79.23%	0.737	79.23%	0.737
Prostate Cancer Study					
25	3	68.85%	0.742	68.86%	0.750
20	7	69.12%	0.750	69.12%	0.752
Umaru Impact Study					
25	3	74.43%	0.585	74.43%	0.587
20	7	75.43%	0.617	74.43%	0.619

Table 2: Comparison of encrypted/unencrypted logistic regression. [Kim+18b]

of the encrypted database. In addition, adapting Nesterov's accelerated gradient descent helps to reduce the number of iterations as well as computational power while assuring the quality of final models.

that is true for any such method.

The original gradient descent has an issue when dealing with the local minimas. If the learning rate is too low, we might stuck in a local minima; therefore, we cannot get to the global minima. Many gradient descent algorithms have been developed to overcome this problem, for example, momentum gradient descent. Nesterov's accelerated gradient descent [Nes83] is a variant of momentum gradient descent. It applies moving average to the update vector, and calculate the gradient at this "look-ahead" position afterward. It is proved to give a better convergence of $O(1/t^2)$ after iterating t steps theoretically as well as practically. The weight update equations of ~~the~~ Nesterov's accelerated gradient descent are shown in figure 4.

get /um
/um

a

ng

Cryptic w/o further explanation.

$$\theta^{(t+1)} = v^{(t)} - \alpha_t * \nabla J(v^{(t)})$$
$$v(t+1) = (1-\gamma_t) * \theta^{(t+1)} + \gamma_t * \theta^{(t)}$$

o \cdot
oo

Figure 4: Equations of Nesterov's accelerated gradient descent. $0 < \gamma_t < 1$. α_t ?

In order to achieve an efficient computation, the authors pack all data points of a dataset into a single vector in row-by-row order. The concept is shown in figure 5; where each row represents one data point, $(f+1)$ is the number of features, and n is the number of data points in the dataset. To evaluate gradient descent on the vector w, shifting operations of row and column vectors are needed. A shift algorithm *Rot*(w, r) can shift the encrypted vector w by r positions. For example, we get a new dataset Z' (see figure 6), if we do *Rot*$(w, f+1)$ (which means we shift the vector w by $(f+1)$ positions).

Applying the Nesterov's accelerated gradient descent [Nes83] and the data packing

$$Z = \begin{bmatrix} z_{10} & z_{11} & \cdots & z_{1f} \\ z_{20} & z_{21} & \cdots & z_{1f} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n0} & z_{n1} & \cdots & z_{nf} \end{bmatrix}$$

(a) The dataset described as a matrix Z

$$Z \mapsto \mathbf{w} = (z_{10}, \ldots, z_{1f}, z_{20}, \ldots, z_{2f}, \ldots, z_{n0}, \ldots, z_{nf})$$

(b) The matrix Z is packed in a vector w

Figure 5: Data encoding concept of paper [Kim+18a]

$$Z' = \begin{bmatrix} z_{20} & z_{21} & \cdots & z_{2f} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n0} & z_{n1} & \cdots & z_{nf} \\ z_{10} & z_{11} & \cdots & z_{1f} \end{bmatrix}$$

Figure 6: New matrix Z' corresponding to the shifted vector w'. Where $w' = Rot(w, f+1)$ [Kim+18a]

approach above, we the authors achieve similar output models as the approach described in [Kim+18b] with much less runtimes and less allocated memory. The results are shown in the table 3.

4 Evaluation

4.1 Implementation

VTN

VTN

All experiments were executed on free tier Google Colab virtual machine which gives 12 GB of RAM. We chose to use TenSEAL and numpy to perform encryption and arithmetic operations on the data. Before encrypting the data, we need to clarify some important — parameters ~~arguments~~ which are needed for encryption:

- `coeff_mod_bit_sizes` is a list of numbers. The first number and the last number decide the precision of the decrypted result. Each of the remaining numbers decides the bit size of a result each time we apply multiplication on the encrypted data.

? I understood sth. different ...

Dataset	Sample num	Feature num	Method	deg g	Iter num	Enc time	Learn time	Storage	Accuracy	AUC
Edinburgh	1253	9	[Kim+18a]	5	7	2s	3.6 min	0.02 GB	91.04%	0.958
			[Kim+18b]	3	25	12s	114 min	0.69 GB	86.03%	0.956
			[Kim+18b]	7	20	12s	114 min	0.71 GB	86.19%	0.954
lbw	189	9	[Kim+18a]	5	7	2s	3.3 min	0.02 GB	69.19%	0.689
			[Kim+18b]	3	25	11s	99 min	0.67 GB	69.30%	0.665
			[Kim+18b]	7	20	11s	86 min	0.70 GB	69.29%	0.678
nhanes3	15649	15	[Kim+18a]	5	7	14s	7.3 min	0.16 GB	79.22%	0.717
			[Kim+18b]	3	25	21s	235 min	1.15 GB	79.23%	0.732
			[Kim+18b]	7	20	21s	208 min	1.17 GB	79.23%	0.737
pcs	379	9	[Kim+18a]	5	7	2s	3.5 min	0.02 GB	68.27%	0.740
			[Kim+18b]	3	25	11s	103 min	0.68 GB	68.85%	0.742
			[Kim+18b]	7	20	11s	97 min	0.70 GB	69.12%	0.750
uis	575	8	[Kim+18a]	5	7	2s	3.5 min	0.02 GB	74.44%	0.603
			[Kim+18b]	3	25	10s	104 min	0.61 GB	74.43%	0.585
			[Kim+18b]	7	20	10s	96 min	0.63 GB	75.43%	0.617

Table 3: Implementation results for 5 datasets with 5-fold CV [Kim+18a]

Relate this to the prior description. Is that N, N/2, M ?

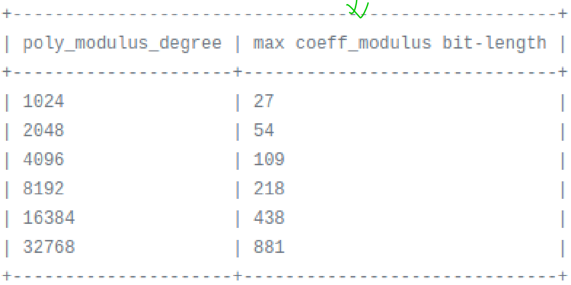
- poly_modulus_degree is a positive power of 2. The larger the value is, the more complicated encrypted computations it allows; however, the slower the operations are. The value selection of poly_modulus_degree depends on the value of coeff_mod_bit_sizes. See figure 7. For example: if coeff_mod_bit_sizes = [40, 20, 40], the total of coefficient mod bit sizes is 100; so that we have to chose the value 4096 of poly_modulus_degree which correspond to the value 109 of the max coeff_modulus bit-length.

Why?

Why?

- global_scale determines the bit-precision of the encoding, so that it affects the precision of the result.

Rb. Isn't this the scaling factor? So it determines the number of correct post-comma bits. „precision" is an ambiguous word.



poly_modulus_degree	max coeff_modulus bit-length
1024	27
2048	54
4096	109
8192	218
16384	438
32768	881

Figure 7: Dependency between coeff_mod_bit_sizes and poly_modulus_degree Microsoft SEAL's Github Repository

In this lab, we chose these settings:
- coeff_mod_bit_sizes $= [37, 28, 28, 28, 28, 28, 28, 28, 37]$
- poly_mod_degree $= 16384$
- global_scale $= 2^{28}$

Because we have a multiplicative depth of 7, the number list coeff_mod_bit_sizes has a size of 9.

With regard to sigmoid approximation, we decided to use g_3, because it needs a smaller depth for evaluation compared with g_7 (see figure 2b). In addition, we did not use any other improved variant of the gradient descent than just the original one. A pseudo-code 1 shows how we implement the Logistic Regression with the gradient descent to train CKKS-encrypted data. In line 11 of the pseudo-code 1, we need to bootstrap the *enc_weight* and *enc_bias*, because our settings only allow the multiplicative depth of 7. If we continue the while loop without bootstrapping, the information loss of the encrypted weights and the encrypted bias will increase drastically, which also prevents information recovery. Since the TenSEAL library did not support bootstrapping in CKKS mode at the time of writing, we simulate the bootstrapping by decrypting the weights and the bias then encrypting them again.

algorithm 1 !?

Algorithm 1 Logistic Regression with the gradient descent to train CKKS-encrypted data

```
Require: $S$ - the CKKS-encrypted dataset consisting of $m$ tuples $(enc\_x, enc\_y)$,
    $enc\_weight$ - initial weight vector consisting of 0s, $enc\_bias$ - initial bias with value
    of 0, $0 \leq lr \leq 1$ - learning rate
while $N \leq EPOCH\_count$ do
    for each $(enc\_x, enc\_y) \in \mathcal{S}$ do
        $\hat{y} \leftarrow approximate\_sigmoid((enc\_x * enc\_weight) + enc\_bias)$
        $\Delta w \leftarrow \Delta w + enc\_x * (\hat{y} - enc\_y)$
        $\Delta b \leftarrow \Delta b + (\hat{y} - enc\_y)$
    end for
    $enc\_weight \leftarrow enc\_weight - lr * (\Delta w/m)$
    $enc\_bias \leftarrow enc\_bias - lr * (\Delta b/m)$
    $\Delta w \leftarrow 0$, $\Delta b \leftarrow 0$
    $N \leftarrow N + 1$
    Bootstrapping
end while
Return $enc\_weight$ and $enc\_bias$
```

4.2 Datasets

VTN

There are 4 datasets that are used to evaluate the experiments:

- framingham: This dataset has 40000 rows, and 16 columns. Since we dropped some unrelated features and remove rows with missing values, only 1114 9-D datapoints were used to train a Logistic Regression model. The final model is to predict if "10 year risk of coronary heart disease CHD" is possible or not based on specific inputs.
- LogReg_sample_dataset: This dataset is randomly generated. It has 1000 2-dimensional (2-D) datapoints. They are labeled with either 0 or 1, such that group of points with label 0 and group of points with label 1 are linearly separable. See definition of linear separability in figure 8.
- HRF_sample_small: This dataset is also randomly generated. It is linearly separable and it has 1000 5-D datapoints labeled with 0 or 1.
- HRF_samples_big: Similar to the dataset HRF_sample_small, it is linearly separable. However, it has 50000 5-D labeled with 0 or 1 datapoints.

probable?

- whether

the

the

Compare

Let X_0 and X_1 be 2 sets of points in an n-dimensional Euclidean space. Then X_0 and X_1 are *linearly separable* if there exists $n+1$ real numbers $w_1, w_2, .., w_n, k$ such that every point $x \in X_0$ satifies $\sum_{i=1}^{n} w_i x_i > k$ and every point $x \in X_1$ satifies $\sum_{i=1}^{n} w_i x_i < k$, where x_i is the i-th component of x.

Figure 8: Definition of linear separability[Wik21]

Shouldn't be a figure, for defs use \begin{definition}\label{...}...

4.3 Experiment Result

VTN

For each dataset, we use 70% of the dataset to train, and the remaining 30% of the dataset is used as a test set. The number of epochs is 100, which means the training set is shuffled and fed to the algorithm ~~100 times,~~ ~~and~~ the learning rate is 0.01.

in each

In table 4, we show the time to encrypt each dataset and the amount of memory to hold the dataset. The first 2 columns present those properties when encrypting the datapoint one by one in each dataset, while the other 2 columns present the same

two

two

Reasoning for these figures?

You shouldn't have to explain the table. Also this explanation seems to refer to an older version...

Dataset	Sample num	Feature num	Type	Parameter set	Encryption time (seconds)	Allocated memory
framingham	1114	9	non-packed	Set 1	73.6	4.677 GB
			packed	Set 2	0.145	0.0075 GB
LogReg_sample_dataset	1000	2	non-packed	Set 1	64.5	4.197 GB
			packed	Set 1	0.07	0.004 GB
HRF_sample_small	1000	5	non-packed	Set 1	63.589	4.197 GB
			packed	Set 1	0.067	0.0044 GB
HRF_samples_big	first 2000 samples	5	non-packed	Set 1	136.2	8.39 GB
			packed	Set 2	0.17	0.008 GB

Table 4: CKKS encryption time and allocated memory for 4 datasets. NOTE: Some packed datasets require a larger value of `poly_mod_degree`, so that we use 2 parameter sets: Set 1: `poly_mod_degree` $= 16384$, `coeff_mod_bit_sizes` $= [37, 28, 28, 28, 28, 28, 28, 28, 37]$ & Set 2: `poly_mod_degree` $= 32768$, `coeff_mod_bit_sizes` $= [37, 28, 28, 28, 28, 28, 28, 28, 37]$.

Dataset	Sample num	Feature num	Method	Epoch num	seconds/epoch	Accuracy	Loss	Note
framingham	1114	9	CKKS LR	100	385	0.706	0.644	
			plaintext LR	100	0.035	0.658	0.64	
LogReg_sample_dataset	1000	2	CKKS LR	100	217	1.0	0.15	
			plaintext LR	100	0.03	1.0	0.16	
HRF_sample_small	1000	5	CKKS LR	100	303	0.89	0.31	
			plaintext LR	100	0.33	0.89	0.30	
HRF_samples_big	50000	5	CKKS LR	100	X	X	X	Error: out of memory
			plaintext LR	100	0.78	0.89	0.32	

Table 5: CKKS LR implementation results for 4 datasets

fundamentale
using
and
numbering
of help
the target
help do
using with
section -

properties when packing all datapoints of each dataset into a single vector in row-by-row order (see figure 5). We also should note that due to difference in matrix shape between original datasets and packed ones, we have to use different sets of `poly_mod_degree` and `coeff_mod_bit_sizes` to make the encryption possible.* The table clearly shows that the time to encrypt the packed datasets and the amount of memory to hold the encrypted packed datasets are greatly smaller than the measurements of non-packed datasets. Because of the google colab's memory limitation, we could not encrypt the non-packed dataset `HRF_samples_big`, which has 50000 rows. As a result, we only use the first 2000 datapoints in the dataset `HRF_samples_big` to do the comparison. ~~the~~ Another difficulty is that we could not train the encrypted-packed datasets, as the library TenSEAL had not supported vector shifting at the time of conducting the experiments.

* I do not u
this o
the cal
change
\ ig

Table 5 not only tells the tiny differences between the final models trained by the CKKS Logistic Regression using the CKKS-encrypted data and the ones trained by the normal Logistic Regression using the original datasets, but also states the huge gaps of the runtime per epoch. The reason why the differences are insignificant is because we do simulate √bootstrapping process of CKKS during training the data, which also means the error produced by applying multiple multiplications on encrypted data is minimized. For the dataset `HRF_samples_big`, the out-of-memory error occurs during CKKS encryption process, which also stop the program from processing further to the learning step.

√the

~ Did?

In attempt to improve the runtime of the CKKS Logistic Regression's training process, we also try to apply concurrency programming pattern to the algorithm (section 5). Furthermore, we propose a way to reduce the memory needed to encrypt the data in section 6 in order to solve the out-of-memory problem we have in table 5.

5 Parallelization of learning

We've shown that learning on encrypted data which are encrypted with the CKKS scheme is possible. But it takes a long time to train the logistic regression. GM

The first attempt to speed up the learning process was to find a way to parallelize the calculation.

$$\Delta w = \sum_{i=}^{N} input_i * (output_i - expected_i)$$

O\cdot!

With a look at the equation to calculate the weight changes during the learning process it is obvious where the approach is to split the equation for the threads.

$$\Delta w = \sum_{i=0}^{\frac{1}{T}N} input_i * (output_i - expected_i) + ... + \sum_{i=\frac{T-1}{T}N}^{N} input_j * (output_j - expected_j)$$

for $0 < i < \frac{1}{T}N$ i the index)

After splitting the sum into thread many sums we can calculate the ~~sum pieces~~ partial sums in parallel and after this we need to sum up the pieces to get the whole sum. This should lead us to a speed up of learning. But we need to take care about some points if we work with threads.

First of all we have only additions and multiplications, which are really fast calculations, in the threads. So we need to think about the overhead of threads. To create a thread some ~~among~~ of time is needed, If the creation needs more time than the calculation then threading is not useful. But we can take care of the overhead if we use the busy waiting method where we start all threads ~~we~~ needed at the beginning of our algorithm and the threads will never closed but they wait for information that they can work on. Here we need to know that all operation systems have an internal key how often a thread can be called in one second. If we create threads above our system kernels, then the operation system needs to set some threads to sleep. But if the down time of these threads is greater than the calculation time then ~~to~~ many threads will not ~~end up~~ in a speed up. We should only use kernel minus one many threads for the calculation because the main process needs one thread too.

— amount

ing

O ing

O ing

O Is that a number?

v is

two — help

Because the threads always run we need a global variable where we can tell the threads that new data are available and another global variable where the threads can tell that they are finished. So our main thread can tell the worker threads when they should work and our main thread knows when he can go on with his calculation.

With ~~this~~ ideas we can create a workflow for the learning algorithm. The pseudo code in Algorithm 2 shows this workflow.

These

Next we need to think about the single threads and how they should work and the calculation they should do.

Because we will work with threads we need to think about race conditions and deadlocks of the threads. We have only one data set for all threads but the threads only need to read on this data set. So there is no problem. To avoid the race condition for the result we can use semaphores but then we maybe slow down the whole calculation because the threads need to wait. Better each thread has his own result and the main process sum up each single thread result after all threads are done with their calculation. For this we set up a global variable for the results. So we don't need semaphores and over all we have no race conditions and no deadlocks.

We start each thread with a unique ID. This ID is the thread number, which starts with zero. With this ID we can specify on which data segment the thread should work.

$$starting\,point = \frac{ID}{number\,of\,threads} * number\,of\,data\,points,$$

$$ending\,point = \frac{ID+1}{number\,of\,threads} * number\,of\,data\,points$$

We also use the ID to tell the thread the place where it can write his result and as an identification of the thread so the main thread knows which thread has finished.
Now ~~the~~ thread needs to calculate his Δw on his data segment and need to wait until all other threads are done with their calculations and the main thread gives feedback to calculate another Δw for the next learning step. The pseudo code in Algorithm 3 shows this thread function.

\each

With this modifications we can do the training like in Algorithm 1 but in parallel.

5.1 Results and conclusions

After several runs with different hardware set ups. We can't find a speed up for the calculations. By searching the web we found out that python has a global interpreter lock. This lock prevents multiple threads from executing codes at the same time. So the threads don't run parallel but linear. The multi threading lib still exist because with the lib the threads are executed more efficiently but not parallel. GM

v in

v in

Instead of using multi threading we can use multi processing which python supports. But for this we need to think about the structure of our code because working with global variables isn't that easy like for multi threading.
Also we have a problem with the space requirements of the algorithm because we encrypt each data point individually. So we can maybe get a speed up by stack ~~some~~ data points before encryption and working with vector arithmetic.
So the parallelization is still an idea for big data sizes but not for smaller ones.

/king

6 Vectorization of data

After the first attempt to make learning more efficient through parallelization, without any effect, we thought about vectorization of the data. For this we take a look at the encryption of data. The ckks library gives us two options for the encryption. We can encrypt the data as vectors or as tensors. If we encrypt the input as tensors we can encrypt whole matrices in one step. GM

CKKS

6.1 Space requirements

The first drawback of our idea for learning on encrypted data was the required space for the encrypted data. So we take a closer look on this topic. We used several matrices as inputs and encrypt them all with the same ckks context that we have used so far. We have measured the space requirement of the matrices if we encrypt the matrix row by row as vectors and took this result as a base line. In the table of figure 9 we can see how ~~much~~ mega byte are use for the encryption. Along a row the dimension of the data points increases. Along a column the number of data points increases. ~~So~~ we encrypted a complete data point with one ckks vector. GM

/many

/ed

In all these cases,

With the graph of figure 9 we can see easily that the ~~difference in~~ dimension of the data points is not measurable for our small dimensions and that the encryption of multi vectors is linear.

— depen

Next, we measured the space requirements of the ckks tensor. The context and the

7

Don't mix tenses!

	[x,3]	[x,4]	[x,5]
[10,y]	18.7	18.6	18.7
[20,y]	38.1	38.0	38.1
[30,y]	57.7	57.6	57.7
[40,y]	77.1	77.0	77.1
[50,y]	96.7	96.7	96.7

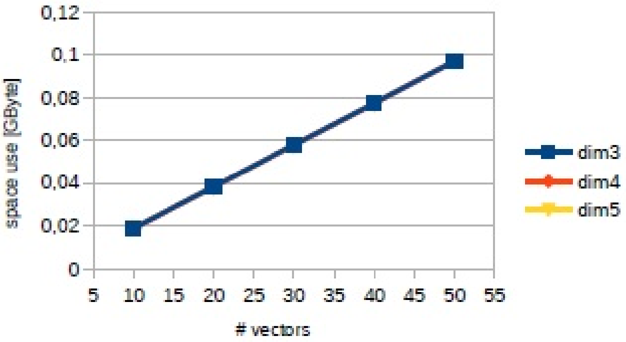


Figure 9: Table: Space requirements of encrypted [mxn]-matrices as vectors row by row in megabyte where m is the number and n is the dimension of data points. Graph: Show table as graph.

input arrays ~~was~~ is the same like before. In the table of figure 10 we can see how ~~much~~ many mega byte are use for the encryption. Along a row the dimension of the data points increases. Along a column the number of data points increases. We encrypted each ~~[mxn]~~-matrix as one ckks tensor.

	[x,3]	[x,4]	[x,5]
[10,y]	116.1	155.0	194.1
[20,y]	174.3	233.1	291.8
[30,y]	232.0	311.4	389.4
[40,y]	288.1	389.4	487.1
[50,y]	336.5	467.4	584.8

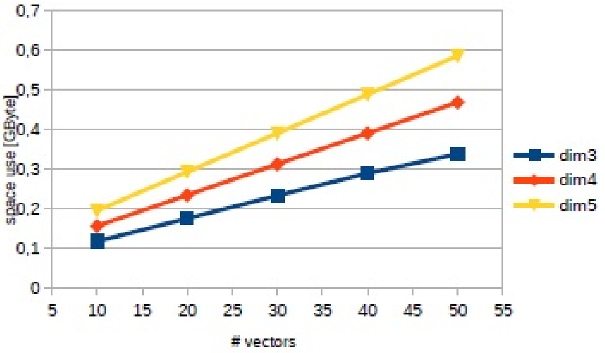


Figure 10: ~~Table:~~ Space requirements of encrypted [mxn]-matrices as one tensor in megabyte where m is the number and n is the dimension of data points. ~~Graph: Show table as graph.~~

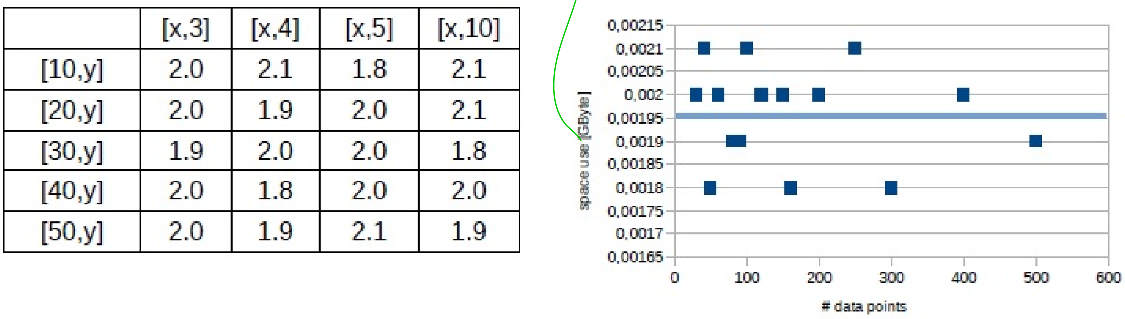
With the graph of figure 10 we can see that the dimension of the data points is now measurable. Higher dimension use more space than lower dimension and over all the space requirements are much more than the encryption row by row. So the encryption as a ckks tensor doesn't solve the space requirement problem we have.
With the fact that the dimension of a vector is not important for the space requirements we flatten the matrix before encryption. The context is still the same like before.

In figure 11 we can see that the flatten matrix use nearly the same space for each vector ~~without respect to~~ independent of the dimension at least for our data range. So we can handle the high space requirement of our algorithm if we use flatten matrices as an input and we can found a way to calculate with such a flatten matrix.

6.2 Calculation with vectorization

Lets assume that all users stores their encrypted data points as a struct where you know the dimension of the data points and the number of data points that are encrypted. Maybe GM per coordinate?

No new info...



	[x,3]	[x,4]	[x,5]	[x,10]
[10,y]	2.0	2.1	1.8	2.1
[20,y]	2.0	1.9	2.0	2.1
[30,y]	1.9	2.0	2.0	1.8
[40,y]	2.0	1.8	2.0	2.0
[50,y]	2.0	1.9	2.1	1.9

Figure 11: ~~Table:~~ Space requirements of encrypted flatten [mxn]-matrices as vector in megabyte where m is the number and n is the dimension of data points. ~~Graph: Space requirements of encrypted flatten [mxn]-matrices as vector by number of data points.~~

/ned
$m\n$

you need the context how the data points are encrypted and at least the encrypted data points as a vector.

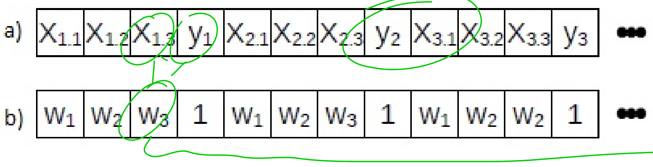
/turn

With this struct we can create another vector, a weight vector, ~~which is dimension of data point plus one times number of data points long~~. This vector consists of number of data points many sub sequences. A sub sequence consists of dimension of data point many weights and a additional 1. Figure 12 shows examples.

an

are g
— whose s
— "

Reformulate, use m, n... "data point many"... is really clumsy!



a)
X1.1 X1.2 X1.3 y1 X2.1 X2.2 X2.3 y2 X3.1 X3.2 X3.3 y3 •••
b)
W1 W2 W3 1 W1 W2 W3 1 W1 W2 W2 1 •••

Why a mix low upper letters?

Should be math! (Font!)

Figure 12: a) Encrypted data point vector. b) Weight vector

After the encryption of the weight vector we can calculate our weights much faster. As an example we calculate the output of the neuron. We take the input vector multiply dot wise with the weight vector and multiply dot wise with the selection vector. The result can we take to approximate the sigmoid function. With the selection vector we can now choose the ~~right~~ expected output. Now we need to switch the expected output into the right position to subtract him from the calculated output. And with this result we need to create a new vector in such a way that we can multiply him to the input vector for calculate the weight changes. But the implementation of this switch vector is still missing so there are no measures about speed up and learning results.

??
m?

is a
60
— thes
" " if

*encryption -> faster? That's probably not the connection that you want to express.

...?

7 Summary and outlook

Stay consistent! No dash?

We could show that logistic regression and learning on ckks-encrypted data is possible. We found some drawbacks with space requirements and learning speed. And without bootstrapping for encrypted data we need to send data back to do bootstrapping on user side.

GM

We found a way for parallelization but in future work we need to rebuild the algorithm to fit with python. We can't use multi-threading but we need to use multi processing.

With the vectorization we have an idea to handle the drawback of space requirements and also the speed up of learning. Unfortunately the ckks-scheme has less matrix operations implemented yet, so we need to build our own library in future work to proof our

prove

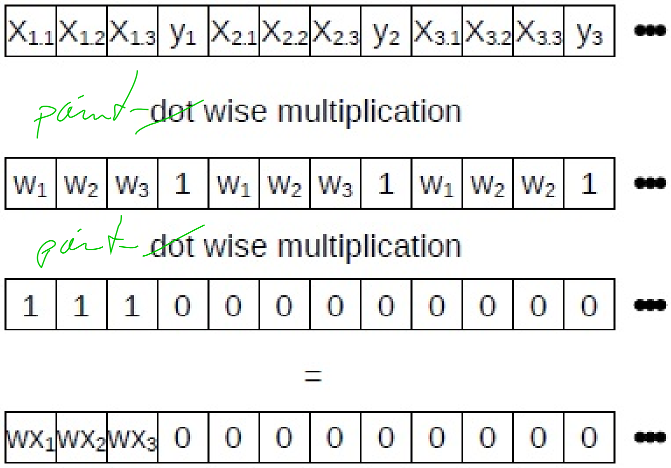

X1.1 X1.2 X1.3 y1 X2.1 X2.2 X2.3 y2 X3.1 X3.2 X3.3 y3
point ~~dot~~ wise multiplication
W1 W2 W3 1 W1 W2 W3 1 W1 W2 W2 1
point ~~dot~~ wise multiplication
1 1 1 0 0 0 0 0 0 0 0 0
=
WX1 WX2 WX3 0 0 0 0 0 0 0 0 0

Figure 13: Scheme for calculating weights for single encrypted data point

ideas of vectorization with respect to ~~of~~ space requirements and speed up.

As another programming task for the future, we need bootstrapping on encrypted data to get ~~better~~ true security. If the ckks-scheme will not give a bootstrapping function to use we need to ~~programm~~ implement one.

After the programming tasks are done we need to think about the security of our algorithm and how secure we are and maybe how we can increase the security level if necessary.

~~And at a last point~~ Finally, for future work we need to think about how data points from various sources can come together to generate a training data pool. Maybe here we can use ~~the~~ parallelization ~~but maybe we need to use cpp as our language instead of python~~.

C++

H ig

/ efficient

CKKS have a

It migh

be imple

in the u

softwa

References

[Nes83] Y. Nesterov. “A method for unconstrained convex minimization problem with the rate of convergence o(1/k^2)”. In: *Soviet Mathematics Doklady* 27 (1983), pp. 372–6.

[Gen09] Craig Gentry. “Fully Homomorphic Encryption Using Ideal Lattices”. In: *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. STOC ’09. Bethesda, MD, USA: Association for Computing Machinery, 2009, pp. 169–178. ISBN: 9781605585062. DOI: 10.1145/1536414.1536440. URL: https://doi.org/10.1145/1536414.1536440.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. “On Ideal Lattices and Learning with Errors over Rings”. In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–23. ISBN: 978-3-642-13190-5.

[FV12] Junfeng Fan and Frederik Vercauteren. “Somewhat practical fully homomorphic encryption.” In: *IACR Cryptol. ePrint Arch.* 2012 (2012), p. 144.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. “A Toolkit for Ring-LWE Cryptography”. In: *Advances in Cryptology – EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 35–54. ISBN: 978-3-642-38348-9.

[Che+17] Jung Hee Cheon et al. “Homomorphic encryption for arithmetic of approximate numbers”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2017, pp. 409–437.

[Che+18] Jung Hee Cheon et al. “Bootstrapping for Approximate Homomorphic Encryption”. In: *Advances in Cryptology – EUROCRYPT 2018*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Cham: Springer International Publishing, 2018, pp. 360–384. ISBN: 978-3-319-78381-9.

[Kim+18a] Andrey Kim et al. “Logistic regression model training based on the approximate homomorphic encryption”. In: *BMC Medical Genomics* 11.4 (Oct. 2018), p. 83. ISSN: 1755-8794. DOI: ~~10.1186/s12920-018-0401-7~~. URL: https://doi.org/10.1186/s12920-018-0401-7.

[Kim+18b] Miran Kim et al. “Secure Logistic Regression Based on Homomorphic Encryption: Design and Evaluation”. In: *JMIR Med Inform* 6.2 (Apr. 2018), e19. ISSN: 2291-9694. DOI: 10.2196/medinform.8805. URL: http://www.ncbi.nlm.nih.gov/pubmed/29666041.

[Wik21] Wikipedia contributors. *Linear separability ~~— Wikipedia, The Free Encyclopedia~~*. [Online; accessed 10-October-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Linear_separability&oldid=1042160240.

Full list of authors here!

. in

→ .

Not the

Mayb

sen

Appendix

Algorithm: Workflow of learning

GM

Algorithm 2 Workflow of learning

```
DECLARE GLOBAL $float$ weight ← 1.0
DECLARE GLOBAL $bool$[] startFlag[threads] ← $FALSE$
DECLARE GLOBAL $float$[] resultArray[threads]← 0.0
DECLARE GLOBAL $bool$[] doneArray[threads] ← $FALSE$

for $i = \#threads$ do
    START thread ← {i, ThreadFunction}
end for

while TRUE do
    doneArray ← $FALSE$
    startFlag ← $TRUE$                                  ▷ Threads start calculation

    while True do
        if doneArray.ALL(TRUE) then
            break                                 ▷ Waiting for all threads are done
        end if
    end while

    Δ w = resultArray.SUM

    if Δ w < $treshold$ then
        break                                               ▷ Training done
    end if

    weight ← weight + Δ w
end while
```

Why is th
not in Li

Both and supported in encrypted setting!

might be approximated.

would break security and thus cannot be done!

Algorithm: ThreadFunction

GM

Algorithm 3 Thread function

```
integer threadID ← i                                              ▷ Given from main threat
integer startPoint = $\lfloor (threadID)/(\#threads) \rfloor * \#dataPoints$
integer endPoint = $\lfloor (threadID+1)/(\#threads) \rfloor * \#dataPoints$
float result ← 0.0

while $TRUE$ do
    startFlag[threadID] ← $FALSE$                                   ▷ Global variable

    for $i$ = startPoint to endPoint do
        result + = input$_i$ * (output$_i$ − expected$_i$) ▷ Calculating Δw from data segment
    end for

    resultArray[threadID] ← result      ▷ Global variable. Give result to main thread
    result ← 0.0
    doneArray ← $TRUE$                      ▷ Global variable. Tell main thread done

    while $TRUE$ do
        if startFlag[threadID] = $TRUE$ then                         ▷ Global variable
            break                                         ▷ Waiting for start command
        end if
        Do nothing
    end while
end while
```

[illegible]

O kdot

Font use inconsistent!