Optimized Privacy-Preserving Clustering with Fully Homomorphic Encryption

Chen Yang[1,2], Jingwei Chen[1,2(✉)], Wenyuan Wu[1,2], and Yong Feng[1,2]

[1] Chongqing Key Laboratory of Secure Computing for Biology, Chongqing Institute of Green and Intelligent Technology, Chinese Academy of Sciences, Chongqing, China
{yangchen,chenjingwei,wuwenyuan,yongfeng}@cigit.ac.cn
[2] Chongqing School, University of Chinese Academy of Sciences, Chongqing, China

Abstract. Clustering is a crucial unsupervised learning method extensively used in the field of data analysis. For analyzing big data, outsourced computation is an effective solution but privacy concerns arise when involving sensitive information. Fully homomorphic encryption (FHE) enables computations on encrypted data, making it ideal for such scenarios. However, existing privacy-preserving clustering based on FHE are often constrained by the high computational overhead incurred from FHE, typically requiring decryption and interactions after only one iteration of the clustering algorithm. In this work, we propose a more efficient approach to evaluate the one-hot vector for the index of the minimum in an array with FHE, which fully exploits the parallelism of single-instruction-multiple-data of FHE schemes. By combining this with FHE bootstrapping, we present a practical FHE-based k-means clustering protocol whose required round of interactions between the data owner and the server is optimal, i.e., accomplishing the entire clustering process on encrypted data in a single round. We implement this protocol using the CKKS FHE scheme. Experiments show that our protocol significantly outperforms the state-of-the-art FHE-based k-means clustering protocols on various public datasets and achieves comparable accuracy to plaintext result. Additionally, We adapt our protocol to support mini-batch k-means for large-scale datasets and report its performance.

Keywords: Clustering · k-means · Mini-batch k-means · Privacy-preserving computation · Fully homomorphic encryption · Outsourced computation

1 Introduction

Nowadays, computation power has become a commercial good. It is more and more common for one party who owns data to utilize the computation power of another party to analyze the data. In this two-party scenario, data privacy is of important concern. There are many solutions proposed to protect data privacy in this outsourced computation scenario, such as differential privacy, secure multi-party computation, and fully homomorphic encryption (FHE). The

S. Alrabaee et al. (Eds.): SecureComm 2024, LNICST 627, pp. 76–96, 2026.
https://doi.org/10.1007/978-3-031-94445-1_5

idea of FHE is to allow one to operate on ciphertexts and get the expected result after decryption. This idea emerged and has been pursued a long time ago [26]. For example, the famous RSA public key encryption system supports homomorphic addition [27]. But a scheme can be called fully homomorphic only when it supports both addition and multiplication simultaneously and has no limit on the number of these two operations that can be consecutively done. It was not until 2009 that Gentry proposed the first FHE scheme [13]. Afterward, many FHE schemes have been proposed, such as BGV [5], BFV [4,11], CKKS [7], GSW [14], TFHE [9] and FHEW [10], and they are less complicated and much more efficient.

Clustering is a crucial unsupervised machine learning method to analyze data, which can reveal intrinsic patterns and characteristics hidden in the data and is widely used in many fields. There are already some privacy-preserving k-means clustering algorithms, such as [16,17,24,30,31], which are based on multi-party computation. Here, we narrow our focus only on those based on FHE. Jäschke and Armknecht [15] proposed an algorithm based on the TFHE scheme, which supports logic operations AND, OR, NOT. So, they construct their algorithm circuit with many logic gates, resulting in their algorithm requiring too much run time. Lu et al. [22] proposed a scheme named PEGASUS which supports switch between two different FHE schemes, CKKS and FHEW, and they accomplished only one iteration of k-means clustering algorithm using PEGASUS. Recently, Zhang et al. [32] proposed a scheme based on CKKS, but did not utilize bootstrapping, so their scheme only supports a few times of iterations of k-means clustering.

1.1 Results

In this paper, we propose a protocol named COPPk-means (Protocol 6) for Completely Outsourced Privacy-Preserving k-means based on FHE. It is accurate, efficient, and completely outsourced.

- To test the effectiveness of our protocol, we compare the accuracy of our privacy-preserving protocol on ciphertexts with that of the original k-means clustering on plaintexts, and the result shows almost equivalent accuracy of them, with differences smaller than 3%.
- To test the efficiency of our protocol, we compare the run time of our protocol with some other previous works. Compared with the work of Jäschke and Armknecht [15], which is completely outsourced but too costly, our work is 56683× faster. Compared with the work of Lu et al. [22], which accomplishes only one iteration of k-means clustering, our work is up to 44.3× faster. Compared with the work of Zhang et al. [32], which will require decryption and re-encryption after only one or two iterations of k-means clustering, our work is about 2× to 3× faster than theirs under the same setup (i.e., without bootstrapping).
- In particular, our protocol is based on the CKKS scheme and utilizes the bootstrapping of CKKS to enable a limitless number of iterations of k-means

clustering. Hence, it requires only one round of interaction between the data owner and the computation server. We also report the performance (run time and memory consumed) of our COPPk-means protocol with bootstrapping on several popular datasets, which shows that our protocol performs well in both efficiency and memory consumption.

Furthermore, for large-scale datasets, we present a mini-batch variant of our COPPk-means. And we report its performance on the MNIST dataset compressed by PCA to extract 64 features from 784 features for all the 60000 samples in the training set.

1.2 Techniques

Comparison and ciphertext division are two unfriendly operations for CKKS. However, k-means clustering requires comparing distances and also requires divisions to update centroids.

In the "compare distances" step of k-means clustering, which is meant to allocate points to their nearest centroids by comparing their distances to centroids, we use a polynomial to approximate the comparison function [8]. In the "update centroids" step, to avoid dividing by the ciphertext of the numbers of points that allocated to the same centroid, we adopt the stabilized variant of k-means (see, e.g., [15]), which let the centroids in the last iteration to play a role in the next iteration. In this way, dividing by a ciphertext (that encrypts the number of points allocated to each centroid) can be replaced by dividing by a plaintext (i.e., the whole number of points in the dataset). We also leverage the batching technique of CKKS, which enables multiple values to be encoded and encrypted in one ciphertext.

Since the sign function is approximated by a polynomial, which requires multiple times of ciphertext addition and multiplication operations, we propose a parallel method (Sect. 3.4), against the usual serial method used in comparison operation on plaintexts, to find the minimum element in an array, and thus save plenty of levels needed to finish these addition and multiplication operations on ciphertexts. By integrating the parallel comparison method with the batching technique of the CKKS scheme, our protocol theoretically achieves high efficiency in terms of program run time.

2 Background

In this section, we introduce the basic concepts of k-means clustering and FHE that are necessary for the subsequent discussions.

2.1 Original K-means Clustering Algorithm

K-means is a classic and well-known clustering algorithm, which requires no prior knowledge about datasets and uses an iterative method to obtain the final clusters. It contains the following steps.

1. **Initialization**: set the value of "k", the number of clusters you want. Then choose k points randomly from the dataset, to be the initial centroids of the k clusters.
2. **Repeat the following steps**, until the k centroids change little or reach the predetermined times of iterations.
   (a) *Compute distances*: compute distances between all points in the dataset and all the k centroids. Here, the distance could be l_2-norm distance or other distance metrics.
   (b) *Compare distances*: allocate points in the dataset to its nearest centroid. These points that are allocated to the same centroid form a cluster. So by the end of this step, we obtain k clusters.
   (c) *Update centroids*: recalculate k centroids of these k new clusters, then go back to Step (a).

2.2 Fully Homomorphic Encryption

As mentioned in Sect. 1, there are many FHE schemes have been proposed since Gentry's seminal work [13], such as BGV [5], BFV [4,11], CKKS [7], GSW [14], TFHE [9] and FHEW [10]. Among them, GSW, TFHE and FHEW can homomorphically evaluate AND, OR, NOT operations on 0, 1. While BGV and BFV can homomorphically evaluate addition and multiplication over integers, and CKKS can homomorphically evaluate addition and multiplication on real numbers or complex numbers.

2.3 The CKKS Scheme

In CKKS [7], the plaintext space is $\mathcal{M} = \mathbb{Z}[x]/\langle X^N + 1\rangle =: R$ while messages are complex vectors in $\mathbb{C}^\ell$ with $\ell = N/2$, where N is a power-of-two integer. The ciphertext space of CKKS is $\mathcal{C} = R/qR$, where q is the *ciphertext modulus*, a large integer. The canonical embedding $\mathbb{R}[X]/\langle X^N + 1\rangle \to \mathbb{C}^\ell$ maps $m(X) \in R$ into $\boldsymbol{m} \in \mathbb{C}^\ell$ by evaluating $m(X)$ at the primitive $2N$-roots of unity $\xi_j = \xi^{5^j}$ for $0 \le j < \ell$. The inverse of the canonical embedding encodes a message $\boldsymbol{m}$ as a plaintext $m(X)$. Thus, CKKS naturally support single-instruction-multiple-data (SIMD) operations, i.e., performing an operation on a ciphertext corresponds to performing the same operation on $\ell = N/2$ entries of $\boldsymbol{m}$ in parallel. Each entry of the message $\boldsymbol{m} \in \mathbb{C}^\ell$ is called a *plaintext slot*.

For $\boldsymbol{x} = (x_i)_{1\le i\le \ell}$ and $\boldsymbol{y} = (y_i)_{1\le i\le \ell}$, let $\mathsf{ct}.\boldsymbol{x}$ and $\mathsf{ct}.\boldsymbol{y}$ be the ciphertext encrypted by CKKS under a same public key. Then CKKS supports the following basic operations:

- $\mathsf{Enc}(\boldsymbol{x})$ encrypts $\boldsymbol{x}$ and returns the ciphertext.
- $\mathsf{Add}(\mathsf{ct}.\boldsymbol{x}, \mathsf{ct}.\boldsymbol{y})$ returns an encryption of $\boldsymbol{x} + \boldsymbol{y}$. The Add operation can also accept multiple input parameters and return the sum of them. The input parameters can also be some messages from $\mathbb{C}^\ell$, but there must be at least one ciphertext in these input parameters.

- $\mathsf{Sub}(\mathsf{ct}.\boldsymbol{x}, \mathsf{ct}.\boldsymbol{y})$ returns an encryption of $\boldsymbol{x}-\boldsymbol{y}$. One of the two input parameters can also be a message from $\mathbb{C}^\ell$.
- $\mathsf{Mul}(\mathsf{ct}.\boldsymbol{x}, \mathsf{ct}.\boldsymbol{y})$ returns an encryption of $\boldsymbol{x} \odot \boldsymbol{y}$, where $\odot$ is for Hadamard product, i.e., component-wise multiplication.
- $\mathsf{CMul}(\boldsymbol{m}, \mathsf{ct}.\boldsymbol{x})$ returns an encryption of $\boldsymbol{m} \odot \boldsymbol{x}$, where $\boldsymbol{m}$ is a message $\mathbb{C}^\ell$; for $m \in \mathbb{C}$, $\mathsf{CMul}(m, \mathsf{ct}.\boldsymbol{x})$ is a special case of $\mathsf{CMul}(\boldsymbol{m}, \mathsf{ct}.\boldsymbol{x})$ with $\boldsymbol{m} = (m, \ldots, m)$.
- $\mathsf{Square}(\mathsf{ct}.\boldsymbol{x})$ returns an encryption of $\boldsymbol{x} \odot \boldsymbol{x}$.
- $\mathsf{Avg}(\mathsf{ct}.\boldsymbol{x}, n) = \mathsf{Enc}(a_1, \mathbf{0}, a_2, \mathbf{0}, \ldots, a_{\ell/n}, \mathbf{0})$, where $(a_i)_{1 \le i \le \ell/n}$ is the average of $(x_{1+(i-1)n}, \ldots, x_{n+(i-1)n})$. It should be ensured that n divides ℓ.

The security of almost all existing FHE schemes, including CKKS, is based on the Learning With Errors (LWE) assumption [25] or its variants. In particular, CKKS is *semantic secure* under the Ring-LWE assumption [23]. For a detailed discussion on the security of CKKS, we refer to [21].

2.4 Bootstrapping of the CKKS Scheme

When a ciphertext is newly encrypted, it is at a high level (the specific value depends on the setup parameters). As more operations have been done on the ciphertext, the lower the level the ciphertext is at. When the ciphertext is at the lowest level, no further operations can be done on it. To support further operations, the ciphertext must be refreshed to go back to a higher level. Such a process is called "bootstrap", introduced by Gentry in [13]. In the literature, there exists a series of papers working on bootstrapping for the CKKS scheme, such as [3,6,19,20].

- $\mathsf{Bootstrap}(\mathsf{ct}.\boldsymbol{x})$ return a bootstrapped version of the ciphertext $\mathsf{ct}.\boldsymbol{x}$, which is refreshed back to the initial high level to support further operations.

2.5 Comparison Function Approximated by Polynomial

The CKKS scheme can readily support addition and multiplication, but comparison is an unfriendly operation for CKKS. There have already been some researches on polynomial approximation of comparison function in CKKS context. Actually, comparison function is equivalent to sign function, since to compare two values is equivalent to evaluate sign function on their difference. We adopt the approximation in [8], which proves a theoretically optimal way of polynomial approximation of sign function in the interval $[-1, 1]$ in CKKS context. And the general expression given in that paper is

$$f_t(x) = \sum_{i=0}^{t} \frac{1}{4^i} \cdot \binom{2i}{i} \cdot x \left(1 - x^2\right)^i. \tag{1}$$

But a more accurate polynomial approximation requires more times of addition and multiplication, which means more levels to be consumed. There is a

trade-off between accuracy and numbers of levels consumed. And since the sign function is approximated by a polynomial, the output value of it will not be exactly -1 or 1 but a value in $[-1, 1]$.

– $\mathsf{Sign}(\mathsf{ct}.\boldsymbol{x})$ utilize addition and multiplication according to Eq. (1) on ciphertext $\mathsf{ct}.\boldsymbol{x}$ to accomplish an approximate evaluation of sign function on plaintext $\boldsymbol{x}$. All the elements in $\boldsymbol{x}$ must be between $[-1, 1]$.

3 Building Blocks

In this section, we introduce some building blocks that will be used in next section to construct our whole protocol.

First, we fix some notations that will be used throughout this paper. Let ℓ be the number of slots in a plaintext. The dataset is denoted by a matrix $\boldsymbol{P} = (p_{i,j}) \in \mathbb{R}^{d \times n}$, where d is the dimension of each sample and n is the number of samples, i.e. each column of $\boldsymbol{P}$ represents a sample. Let $\boldsymbol{p}_i$ be the i-th row of $\boldsymbol{P}$, and $\boldsymbol{P}_{[i,j]}$ be the submatrix consists of i-th to j-th rows of $\boldsymbol{P}$. Let $\mathsf{ct}.\boldsymbol{x}$ be an encryption of $\boldsymbol{x}$, and let $\mathsf{ct}.\boldsymbol{P}$ be an encryption of the matrix $\boldsymbol{P}$, viewed as a vector row by row. And k is the number of centroids. In addition, $\mathsf{R}_m(\boldsymbol{x}) = (\boldsymbol{x}, \ldots, \boldsymbol{x}) \in \mathbb{R}^{md}$ for $\boldsymbol{x} \in \mathbb{R}^d$.

3.1 Encode and Encrypt

For simplicity, we assume that $d \cdot n \leq \ell$, which implies that the dataset matrix $\boldsymbol{P}$ can be encrypted into a single ciphertext. But note that our protocol actually also works for larger datasets (like MNIST used in Sect. 6), where $d \cdot n > \ell$.

Algorithm 1 encrypts all data points in the dataset into one ciphertext, minimizing the communication cost between the data owner and computation server.

Algorithm 1. Encode and Encrypt

Input: A matrix $\boldsymbol{P} \in \mathbb{R}^{d \times n}$ with $d \cdot n \leq \ell$.
Output: $\mathsf{ct}.\boldsymbol{P}$ which is an encryption of $\boldsymbol{P}$.
1: Convert $\boldsymbol{P} \in \mathbb{R}^{d \times n}$ to a vector $\boldsymbol{p} = (\boldsymbol{p}_1, \ldots, \boldsymbol{p}_d) \in \mathbb{R}^{nd}$.
2: $\mathsf{ct}.\boldsymbol{P} \leftarrow \mathsf{Enc}(\boldsymbol{p})$.
3: **return** $\mathsf{ct}.\boldsymbol{P}$.

3.2 Extract Points and Centroids

Algorithm 2 is meant to extract the information of data points and centroids from the output ciphertext of Algorithm 1, and the output $(\mathsf{ct}.\boldsymbol{P}_i)_{1 \leq i \leq d}$ and $(\mathsf{ct}.\boldsymbol{C}_i)_{1 \leq i \leq d}$ of Algorithm 2 represent data points and centroids respectively.

The arrangement of data points and centroids in ciphertexts of the algorithm 2 is meant to better utilize the SIMD of the CKKS scheme, such that operations on each pair of $\mathsf{ct}.\boldsymbol{P}_i$ and $\mathsf{ct}.\boldsymbol{C}_i$ accomplish the calculation for the i-th dimension of all k centroids, no matter how large the k is. If ndk is smaller than the number of slots of one ciphertext, all these d ciphertexts representing data points can be combined into one ciphertext and all the d ciphertexts representing centroids also can be combined into one ciphertext, such that operations on the two ciphertexts accomplish the calculation for all d dimension of all k centroids, achieving a better utilization of SIMD of the CKKS scheme.

Algorithm 2. Extract Points and Centroids

Input: $\mathsf{ct}.\boldsymbol{P}$, k, d and n.

Output: $(\mathsf{ct}.\boldsymbol{P}_i)_{1\le i\le d}$ and $(\mathsf{ct}.\boldsymbol{C}_i)_{1\le i\le d}$, where $\mathsf{ct}.\boldsymbol{P}_i$ is an encryption of $\mathsf{R}_k(\boldsymbol{p}_i) \in \mathbb{R}^{kn}$ and $\mathsf{ct}.\boldsymbol{C}_i$ is an encryption of $(\mathsf{R}_n(p_{i,r_j}))_{1\le j\le k} \in \mathbb{R}^{kn}$.

1: Generate k random numbers $[r_1, r_2, \ldots, r_k]$, indicating which data points will be chosen as initial centroids.
2: **for** $1 \le i \le d$ **do**
3: Set $\mathsf{ct}.\boldsymbol{p}_i$ to be the ciphertext of $\boldsymbol{p}_i$ extracted from $\mathsf{ct}.\boldsymbol{P}$ and then repeat $\mathsf{ct}.\boldsymbol{p}_i$ for k times to obtain $\mathsf{ct}.\boldsymbol{P}_i$.
4: **for** $1 \le i \le d$ **do**
5: **for** $1 \le j \le k$ **do**
6: Extract a ciphertext $\mathsf{ct}.\boldsymbol{p}_{i,r_j}$ from $\mathsf{ct}.\boldsymbol{P}$.
7: Repeat $\mathsf{ct}.\boldsymbol{p}_{i,r_j}$ for n times to obtain $\mathsf{ct}.\tilde{\boldsymbol{p}}_{i,r_j}$.
8: Concatenate these $\mathsf{ct}.\tilde{\boldsymbol{p}}_{i,r_j}$'s as one ciphertext $\mathsf{ct}.\boldsymbol{C}_i$.
9: **return** $(\mathsf{ct}.\boldsymbol{P}_i)_{1\le i\le d}$ and $(\mathsf{ct}.\boldsymbol{C}_i)_{1\le i\le d}$

3.3 Compute Distances

Algorithm 3 is meant to compute the square of distances between all data points and all centroids, which will be used as input of Algorithm 4 later. We note that the for-loop in Algorithm 3 can be executed in parallel for further acceleration.

Algorithm 3. Compute Distances

Input: d, $(\mathsf{ct}.\boldsymbol{P}_i)_{1\le i\le d}$ and $(\mathsf{ct}.\boldsymbol{C}_i)_{1\le i\le d}$ where $\mathsf{ct}.\boldsymbol{P}_i$ is an encryption of $\mathsf{R}_k(\boldsymbol{p}_i) \in \mathbb{R}^{kn}$ and $\mathsf{ct}.\boldsymbol{C}_i$ is an encryption of $(\mathsf{R}_n(p_{i,r_j}))_{1\le j\le k} \in \mathbb{R}^{kn}$.

Output: $\mathsf{ct}.\boldsymbol{D}$ which is an encryption of $\boldsymbol{D} = (d_{i,j}) \in \mathbb{R}^{k\times n}$, and $d_{i,j}$ is the square of the Euclidean distance between the i-th centroid and the j-th data point.

1: **for** $i = 1, \ldots, d$ **do**
2: $\mathsf{ct}.\boldsymbol{t}_i \leftarrow \mathsf{Square}(\mathsf{Sub}(\mathsf{ct}.\boldsymbol{P}_i, \mathsf{ct}.\boldsymbol{C}_i))$.
3: $\mathsf{ct}.\boldsymbol{D} \leftarrow \mathsf{Add}((\mathsf{ct}.\boldsymbol{t}_i)_{1\le i\le d})$.
4: **return** $\mathsf{ct}.\boldsymbol{D}$

3.4 Compare Distances

Parallel One-to-One Comparison to Speedup. The most difficult part of k-means clustering algorithm realized with CKKS is that to allocate points in a dataset to its nearest centroid, because this step requires finding the smallest value in an array. As said above, the comparison operation is demanding in CKKS, which is approximated by a polynomial and needs many levels. If we practice the find-minimum operation in CKKS as the usual serial way does on plaintexts, the SIMD advantage of the CKKS scheme can not be utilized. And most frustrating is that if these comparison operations are done in series, a much more levels will be needed to support serial comparison operations.

Can we utilize the SIMD advantage of the CKKS scheme and make sure a lower level is enough to accomplish the find-minimum operation in CKKS? The key is to avoid serial comparison operations. Consider that if the comparison results of every two elements in an array are known, then this information is enough to determine which is the largest or smallest element in this array. Though it may cost additional comparison operations to obtain the comparison results of every two elements, the point is that now the comparison operations can be done in parallel and the number of levels required is much less. And considerable time can be saved by utilizing the SIMD feature of CKKS. We name such a method the parallel one-to-one comparison method.

We remark that this parallel one-to-one comparison method can also be used in other FHE applications that need to find the biggest or smallest element in an array to save considerable time and levels.

Below, in Table 1, we give an easy example of this parallel one-to-one comparison method. The final result in the rightmost column is a one-hot vector indicating the location of the minimum element. It can be calculated from the comparison results of every two elements, by multiplying every 0 or 1 in each row together, dismissing the asterisk.

Table 1. An easy example illustrating parallel one-to-one comparison method

	8	7	9	6	result
8	*	0	1	0	0
7	1	*	1	0	0
9	0	0	*	0	0
6	1	1	1	*	1

Compare Distance Algorithm. We adopt the parallel one-to-one comparison method into Algorithm 4 to save considerable run time and levels needed.

Algorithm 4 packs all the distances that need to be compared into two ciphertexts (Step 1), requiring only one time of the Sign evaluation on the subtracted difference of the two ciphertexts (Step 2 and 3), rather than evaluate Sign on

every difference of these distances. Such pack method saves many times of addition and multiplication, since the sign function is approximated by a polynomial which is consisted of lots of addition and multiplication. After Sign evaluation, convert the result of sign function to the format of comparison result (Step 4 and 5). At last, unpack the result to the matrix format and multiply them together (Step 6 to 8) to obtain the final output.

Algorithm 4. Compare Distances

Input: k and $\mathsf{ct}.\boldsymbol{D}$ which is an encryption of $\boldsymbol{D} \in (0,1)^{k\times n}$.

Output: $\mathsf{ct}.\boldsymbol{B}$ which is an encryption of $\boldsymbol{B} \in (0,1)^{k\times n}$, and the i-th column of $\boldsymbol{B}$ is an approximate one-hot vector indicating the index of the minimum in the i-th column of $\boldsymbol{D}$.

1: Construct ciphertexts $\mathsf{ct}.\boldsymbol{D}_1$ and $\mathsf{ct}.\boldsymbol{D}_2$ that are encryptions of $(\mathsf{R}_{k-1}(\boldsymbol{d}_1), \mathsf{R}_{k-2}(\boldsymbol{d}_2), \ldots, \mathsf{R}_1(\boldsymbol{d}_{k-1}))$ and $(\boldsymbol{D}_{[2,k]}, \boldsymbol{D}_{[3,k]}, \ldots, \boldsymbol{D}_{[k,k]})$, respectively, where $\boldsymbol{d}_i$ is the i-th row of $\boldsymbol{D}$.
2: $\mathsf{ct}.\boldsymbol{E} \leftarrow \mathsf{Sub}(\mathsf{ct}.\boldsymbol{D}_1, \mathsf{ct}.\boldsymbol{D}_2)$.
3: $\mathsf{ct}.\boldsymbol{E}' \leftarrow \mathsf{Sign}(\mathsf{ct}.\boldsymbol{E})$. ▷ $e'_{i,j} \in (-1, 1)$.
4: $\mathsf{ct}.\boldsymbol{F} \leftarrow \mathsf{CMul}(0.5 \cdot \mathbf{1}, \mathsf{Add}(\mathsf{ct}.\boldsymbol{E}', \mathbf{1}))$.
5: $\mathsf{ct}.\boldsymbol{F}' \leftarrow \mathsf{Sub}(\mathbf{1}, \mathsf{ct}.\boldsymbol{F})$.
6: **for** $i = 1, \ldots, k-1$ **do**
7: Concatenate $(\mathsf{ct}.\boldsymbol{f}'_{i+a_j})_{1\le j\le i}$, where $a_j = (j-1)(k-1) - j(j-1)/2$, and $\mathsf{ct}.\boldsymbol{F}_{[1+(i-1)(k-1)-(i-1)(i-2)/2, i(k-1)-i(i-1)/2]}$ as one ciphertext $\mathsf{ct}.\boldsymbol{G}_i$.
8: $\mathsf{ct}.\boldsymbol{B} \leftarrow \mathsf{Mul}((\mathsf{ct}.\boldsymbol{G}_i)_{1\le i\le k-1})$.
9: **return** $\mathsf{ct}.\boldsymbol{B}$

The following example illustrates how Algorithm 4 works. Suppose now we have $n = 4$, $k = 3$, and the matrix $\boldsymbol{D}$ as

$$\boldsymbol{D} = \begin{pmatrix} 0.2 & 0.3 & 0.9 & 0.5 \\ 0.1 & 0.4 & 0.7 & 0.6 \\ 0.4 & 0.8 & 0.5 & 0.2 \end{pmatrix}$$

We want to find the minimum in each column. Firstly, according to Step 1, we construct

$$\begin{aligned} \boldsymbol{D}_1 &= (\mathsf{R}_2(\boldsymbol{d}_1), \mathsf{R}_1(\boldsymbol{d}_2)) = (\boldsymbol{d}_1, \boldsymbol{d}_1, \boldsymbol{d}_2) \\ &= (0.2, 0.3, 0.9, 0.5,\ 0.2, 0.3, 0.9, 0.5,\ 0.1, 0.4, 0.7, 0.6) \end{aligned}$$

and

$$\begin{aligned} \boldsymbol{D}_2 &= (\boldsymbol{D}_{[2,3]}, \boldsymbol{D}_{[3,3]}) = (\boldsymbol{d}_2, \boldsymbol{d}_3, \boldsymbol{d}_3) \\ &= (0.1, 0.4, 0.7, 0.6,\ 0.4, 0.8, 0.5, 0.2,\ 0.4, 0.8, 0.5, 0.2). \end{aligned}$$

Then after the execution of Step 2, we have $\boldsymbol{E} = \boldsymbol{D}_1 - \boldsymbol{D}_2$:

$$(0.1, -0.1, 0.2, -0.1,\ -0.2, -0.5, 0.4, 0.3,\ -0.3, -0.4, 0.2, 0.4).$$

Then evaluating Sign function (Step 3) on $\boldsymbol{E}$ obtains

$$\boldsymbol{E}' = Sign(\boldsymbol{E}) = (1, -1, 1, -1,\ -1, -1, 1, 1,\ -1, -1, 1, 1).$$

Note that the actual values of $\boldsymbol{E}'$ may not be exactly -1 or 1, because the sign function is approximated by a polynomial in CKKS. We use -1 and 1 here for the purpose of an easy example.

Now execute step 4 and 5 to obtain

$$\boldsymbol{F} = 0.5 \odot \mathbf{1} \odot (\boldsymbol{E}' + \mathbf{1}) = (1, 0, 1, 0,\ 0, 0, 1, 1,\ 0, 0, 1, 1)$$

and

$$\boldsymbol{F}' = \mathbf{1} - \boldsymbol{F} = (0, 1, 0, 1,\ 1, 1, 0, 0,\ 1, 1, 0, 0).$$

Then according to Step 6, we construct

$$\boldsymbol{G}_1 = (\boldsymbol{f}_1', \boldsymbol{F}_{[1,2]}) = (0, 1, 0, 1,\ 1, 0, 1, 0,\ 0, 0, 1, 1)$$

and

$$\boldsymbol{G}_2 = (\boldsymbol{f}_2', \boldsymbol{f}_3', \boldsymbol{F}_{[3,3]}) = (1, 1, 0, 0,\ 1, 1, 0, 0,\ 0, 0, 1, 1).$$

Finally, according to Step 7, multiply $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$ together in element-wise way, obtain

$$\boldsymbol{B} = \boldsymbol{G}_1 \odot \boldsymbol{G}_2 = (0, 1, 0, 0,\ 1, 0, 0, 0,\ 0, 0, 1, 1) \rightarrow \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}.$$

We can write $\boldsymbol{B}$ in matrix format to check that the result is correct. Since in the first column of $\boldsymbol{B}$, "1" is at the second location, so we check that the second location of the first column of $\boldsymbol{D}$ is "0.1", and indeed "0.1" is the minimum value in the first column of $\boldsymbol{D}$. Other columns can also be checked in this way.

3.5 Update Centroids

Stabilized K-means. Since ciphertext division is an unfriendly operation for the CKKS scheme, it is difficult to accomplish the "Update centroids" step in k-means clustering using CKKS. This step requires dividing the sum of values of the points in a cluster by the number of points in this cluster, which is in ciphertext state, to obtain the new centroid of this cluster.

So to avoid the ciphertext division, we adopt the stabilized variant of k-means clustering [15]. Instead of dividing by the number of points in a cluster, which is in ciphertext state and may change in each iteration, we divide by the number of all points in the dataset, which is not encrypted and never change in every iteration. But now the problem is that we are dividing by something bigger. So to keep the result not too different from the original k-means, something else must be added to the numerator, i.e. sum of values of the points in a cluster, before division. Here, we choose the value of centroid in last iteration to add.

And such way makes sense, since the centroid in the last iteration play a role like anchor, preventing the new centroid from changing too much.

Suppose the number of points is n, number of centroids is k, and $newc_i$ represents the i-th centroid in current iteration, $oldc_i$ represents the i-th centroid in last iteration, p_j represents the j-th point, and $b_{i,j}$ indicates whether p_j should be allocated to $oldc_i$, value being 1 if should and 0 if not. Below is the equation of the stabilized method to calculate new centroids.

$$newc_i = \left(\sum_{j=1}^{n} (p_j \cdot b_{i,j} + oldc_i \cdot (1 - b_{i,j})) \right) /n, \ \ i = 1, ..., k \tag{2}$$

Update Centroid Algorithm. Algorithm 5 adopts the stabilized method to update centroids to avoid ciphertext division in CKKS.

Algorithm 5. Update Centroids

Input: d, n, $\mathsf{ct}.\boldsymbol{B}$ which has the same format as the output of Algorithm 4, $(\mathsf{ct}.\boldsymbol{P}_i)_{1\le i\le d}$ and $(\mathsf{ct}.\boldsymbol{C}_i)_{1\le i\le d}$ which are ciphertexts of data points and centroids respectively.

Output: $(\mathsf{ct}.\boldsymbol{C}'_i)_{1\le i\le d}$ which are ciphertexts of updated centroids.

1: **for** $i = 1$ to d **do**
2: Compute $\mathsf{ct}.\boldsymbol{H}_i \leftarrow \mathsf{Add}(\mathsf{Mul}(\mathsf{ct}.\boldsymbol{P}_i, \mathsf{ct}.\boldsymbol{B}), \mathsf{Mul}(\mathsf{ct}.\boldsymbol{C}_i, \mathsf{Sub}(\mathbf{1}, \mathsf{ct}.\boldsymbol{B})))$.
3: $\mathsf{ct}.\boldsymbol{c}_i \leftarrow \mathsf{Avg}(\mathsf{ct}.\boldsymbol{H}_i, n)$. ▷ The decryption of $\mathsf{ct}.\boldsymbol{c}_i$ contains the information of the i-th dimension of all the k updated centroids.
4: Extract the information of i-th ($1 \le i \le d$) dimension of all updated centroids from $(\mathsf{ct}.\boldsymbol{c}_i)_{1\le i\le d}$ and combine them all into one ciphertext $\mathsf{ct}.\boldsymbol{c}$.
5: $\mathsf{ct}.\boldsymbol{c}' \leftarrow \mathsf{Bootstrap}(\mathsf{ct}.\boldsymbol{c})$.
6: Extract d ciphertexts $(\mathsf{ct}.\boldsymbol{c}'_i)_{1\le i\le d}$ from $\mathsf{ct}.\boldsymbol{c}'$ such that they are in the same format as $\mathsf{ct}.\boldsymbol{c}_i$.
7: Construct d ciphertexts $(\mathsf{ct}.\boldsymbol{C}'_i)_{1\le i\le d}$ from $(\mathsf{ct}.\boldsymbol{c}'_i)_{1\le i\le d}$ such that they are in the same format as the output of Algorithm 2.
8: **return** $(\mathsf{ct}.\boldsymbol{C}'_i)_{1\le i\le d}$.

Algorithm 5 actually has already obtained the information of updated centroids when the for-loop (Step 1 to 3) ends. Step 4–6 are meant to combine the information of all centroids into one ciphertext, so to save the times of bootstrapping needed. Step 7 recovers the format of centroids back to the same as the output of Algorithm 2 to continue the next iteration. We note that the bootstrapping operation can be omitted in Step 5 and be operated in following iterations where the levels are used up. But when bootstrapping is operated somewhere else, there could be multiple ciphertexts need to be bootstrapped, consuming much more run time.

4 Our COPPk-means Protocol

Protocol 6 presents our Completely Outsourced Privacy-Preserving k-means clustering protocol, which is constructed by the building blocks in Sect. 3 and consists of two parties, data owner and computation server.

Protocol 6. Completely Outsourced Privacy-Preserving k-means

Input of Data Owner: sk (secret key), ek (evaluation key), k (number of centroids), d (dimension), n (number of data points), T (times of iteration) and the dataset $\boldsymbol{P}$.

Data Owner:

1: Calling Algorithm 1 with dataset $\boldsymbol{P}$ as its input returns $\mathsf{ct}.\boldsymbol{P}$.
2: Send ek, k, d, n, T and $\mathsf{ct}.\boldsymbol{P}$ to the computation server.

Computation Server:

3: **Initialization:** Calling Algorithm 2 with k, d, n and $\mathsf{ct}.\boldsymbol{P}$ as its input returns $2d$ ciphertexts, $(\mathsf{ct}.\boldsymbol{P}_i)_{1\le i\le d}$ and $(\mathsf{ct}.\boldsymbol{C}_i)_{1\le i\le d}$.
4: **Repeat** T **times**:
5: *Compute distances*: Calling Algorithm 3 with d, $(\mathsf{ct}.\boldsymbol{P}_i)_{1\le i\le d}$ and $(\mathsf{ct}.\boldsymbol{C}_i)_{1\le i\le d}$ as its input returns $\mathsf{ct}.\boldsymbol{D}$.
6: *Compare distances*: Calling Algorithm 4 with k and $\mathsf{ct}.\boldsymbol{D}$ as its input returns $\mathsf{ct}.\boldsymbol{B}$.
7: *Update centroids*: Calling Algorithm 5 with d, n, $\mathsf{ct}.\boldsymbol{B}$, $(\mathsf{ct}.\boldsymbol{P}_i)_{1\le i\le d}$ and $(\mathsf{ct}.\boldsymbol{C}_i)_{1\le i\le d}$ as its input returns $(\mathsf{ct}.\boldsymbol{C}'_i)_{1\le i\le d}$.
8: Send $\mathsf{ct}.\boldsymbol{B}$ to the data owner.

Data Owner:

9: **Data owner decrypts:** Decrypt $\mathsf{ct}.\boldsymbol{B}$ with sk to obtain the matrix $\boldsymbol{B}$ with size of $k \times n$. ▷ For i-th $(1 \le i \le n)$ column of $\boldsymbol{B}$, find the location of the largest value in this column to determine which centroid the i-th data point should be allocated to.

Remark 1. *The dataset $\boldsymbol{P}$ owned by the data owner must satisfy that the square of the longest distance between two points in the dataset does not exceed 1, otherwise the data owner should first scale the dataset to satisfy this requirement before encrypting it, since Algorithm 4 in step 6 requires that plaintext of its input ciphertext must be in* $[0, 1]$.

Firstly, Data owner generates secret key sk and evaluation key ek, and owns the dataset $\boldsymbol{P}$ with size of number of points $n \times$ dimension d. Before encrypting $\boldsymbol{P}$, data owner has to operate the scale preprocess on $\boldsymbol{P}$, so as to make the following computation steps on the server side can be operated successfully. The reason for the scale preprocess is explained in Remark 1. Then, data owner encrypts the preprocessed dataset $\boldsymbol{P}$ and sends the ciphertext of it to the computation server. Besides the ciphertext of $\boldsymbol{P}$ and evaluation key ek, data owner has also to send some necessary parameters, including dimension d, number of points n, number of centroids k, and the times of iterations T to be operated.

After receiving the ciphertext of dataset $\boldsymbol{P}$, evaluation key ek, and parameters (n, k, d, T), computation server randomly chooses k data points as initial centroids and then perform T iterations of k-means clustering. After T iterations, the computation server sends the result, which is a ciphertext of a boolean matrix $\boldsymbol{B}$ with size of $k \times n$ indicating which centroids these points should be allocated to, back to the data owner.

Then data owner decrypts $\boldsymbol{B}$ and do some simple calculations to obtain the final result. Each column of $\boldsymbol{B}$ contains the information that which centroid a point should be allocated to. For example, when k is 4, one column with k elements could be (0.01, 0.84, 0.02, 0.10), and the point should be allocated to the second centroid because 0.84 is the maximum in this column.

4.1 Analysis of Our Protocol

Correctness. Correctness is guaranteed since the operations on ciphertexts strictly follow that of stabilized k-means in plain. And the noise introduced by the CKKS scheme has little effect on the final result, which will be checked by an accuracy test in Sect. 5.

Computational Complexity. Since the bootstrapping operation is relatively much more time-consuming compared to other operations, and the primary factor determining the required number of bootstrapping is the levels (multiplication depths) consumed during the computation, so we here focus solely on the number of levels consumed.

In the "Initialization" (Step 3), one level is consumed to extract data points and centroids. In the "Compute distances" (Step 5), only one ciphertext multiplication is required, consuming one level. In the "Compare distances" (Step 6), the Step 1 of Algorithm 4 consumes one level, the Step 3 of Algorithm 4 consumes $\lceil \log_2(p) \rceil$ levels where p is the degree of the approximate polynomial of the sign function, and Step 4–8 of Algorithm 4 consumes $2 + \lceil \log_2(k) \rceil$ levels where k is the number of centroids. And in the "Update centroids" (Step 7), 3 levels are consumed.

Totally, each iteration (Step 5–7 of Protocol 6) consumes $7 + \lceil \log_2(p) \rceil + \lceil \log_2(k) \rceil$ levels, where p is the degree of the approximate polynomial of the sign function and k is the number of centroids.

Communication Complexity. From the description of Protocol 6, it is clear that the entire protocol involves only one round of interaction. Specifically, at the beginning, the data owner sends the encrypted dataset to the computation server, and at the end, the computation server sends the ciphertext results back to the data owner. Therefore, it is optimal in terms of the number of communication rounds. Additionally, the amount of ciphertext data sent by the data owner includes at most $\lceil nd/\ell \rceil$ ciphertexts, while the encrypted results contain at most $\lceil nk/\ell \rceil$ ciphertexts. In particular, when $\max(nd, nk) \leq \ell$, the communication overhead is just two ciphertexts.

Security. Here we only consider the security of Protocol 6 in the *semi-honest* adversarial model. The data owner encrypts the dataset and then sends this ciphertext and evaluation key ek and some parameters (n, k, d, T). It is apparent that T has no relevance to the dataset. The parameters n, k, and d can be considered as public information, because even in the ideal model, such information is also required to complete the clustering computation. All the computations done on the computation server side are on ciphertexts encrypted by CKKS. Therefore, the security follows from the semantic security of the CKKS scheme.

5 Implementation and Experiments

We implemented Protocol 6 with Lattigo v5 [1], and our implementation is available at https://github.com/JohnJimAir/COPPk-means. The machine we use to test our implementation is equipped with Intel Xeon Gold 6248R (3.00GHz, 24Core) and 128G (32G×4) memory. The parameters we use are the default parameters for demonstrating bootstrapping in Lattigo v5 [1], with ring degree $N = 2^{16}$ and $\log q \approx 638$, which achieves a 128 security according to the latest lattice estimator [2].

In our implementation, the specific values of t in the approximate polynomial (Eq. (1)) for the sign function are 3, 7, 15, i.e., $\mathsf{Sign} \approx f_{15}(f_{15}(f_7(f_3(X))))$ which consumes $3 + 4 + 5 + 5 = 17$ levels.

5.1 Clustering Accuracy

We run k-means clustering for $T = 5$ iterations on G2 dataset [12], and $T = 10$ iterations on FCPS dataset [29]. The size of datasets and the average accuracy of clustering result is presented in Table 2, the fifth column for the original k-means algorithm on plaintexts, and the sixth column for our COPPk-means algorithm on ciphertexts. From the result, we can see that there is little accuracy loss of our protocol on ciphertexts compared to the original k-means on plaintexts.

Table 2. Test Accuracy

Dataset	n	d	k	The original	Ours	Difference
G2-1-20	2048	1	2	99.4%	99.3%	−0.1%
G2-2-20	2048	2	2	100.0%	100.0%	0.0%
G2-4-20	2048	4	2	100.0%	100.0%	0.0%
G2-8-20	2048	8	2	100.0%	100.0%	0.0%
G2-16-20	2048	16	2	100.0%	100.0%	0.0%
Chainlink	1000	3	2	65.3%	65.4%	+0.1%
EngyTime	4096	2	2	95.1%	94.8%	−0.3%
Hepta	212	3	7	80.2%	80.2%	0.0%
Lsun	400	2	3	71.0%	71.2%	+0.2%
Tetra	400	3	4	100.0%	96.8%	−3.2%
TwoDiamonds	800	2	2	100.0%	100.0%	0.0%
WingNut	1016	2	2	96.3%	95.3%	−1.0%

5.2 Comparison with Lu et al. [22]

Table 3 shows the time cost of one iteration for datasets whose values are sampled from $[-1, 1]$ uniformly at random with 256, 1024, 4096 points and dimension of 16. We run our implementation of Protocol 6 with 20 threads, as in Lu et al. [22]. From the result, our protocol is 1.5× to 44.3× faster than Lu et al., and the larger the dataset, the faster our protocol compared with the work of them. This is because for the datasets used here, nk does not exceed the number of slots in one ciphertext in our implementation (number of slots = 32768, determined by the setup parameters). So whether n is 256, 1024 or 4096, it makes no difference in terms of run time in our implementation.

Table 3. Compare with Lu et al.

n	k	Lu et al. [22] (min)	Ours (min)	Speedup
256	2	1.35	0.89	1.5×
	4	2.33	0.93	2.5×
	8	4.09	1.16	3.5×
1024	2	3.66	0.89	4.1×
	4	7.57	0.90	8.4×
	8	15.34	1.20	13.2×
4096	2	13.95	0.90	15.4×
	4	26.61	0.90	29.5×
	8	52.04	1.19	44.3×

5.3 Comparison with Jäschke et al. [15] and Zhang et al. [32]

Since Jäschke et al. [15] and Zhang et al. [32] reported their run time on the Lsun dataset with a single thread, here we also test our protocol on the same dataset with a single thread. The work of Jäschke et al. has two versions, exact and approximate. The approximate version is derived from the exact version by dismissing some bits of information encrypted by the TFHE scheme. The result in Table 4 shows that our protocol is 55683× faster than the exact version of Jäschke et al., 1004× faster than the approximate version of Jäschke et al., and 3× faster than Zhang et al..

Since the work of Jäschke et al. is based on the TFHE scheme which operates on logic gates, while ours is based on the CKKS scheme which operates directly on real numbers, the speedup we achieved is an expected result from theory. And the speedup we achieved compared with the work of Zhang et al., which is also based on the CKKS scheme, could result from the different way adopted to deal with the ciphertext division operation in "Update centroids" step. Zhang et al. adopted polynomial to approximate the division operation, which requires

multiple times of addition and multiplication on ciphertexts, while we adopt the stabilized method to avoid the division operation, which requires only one time of ciphertext multiplication.

Table 4. Compare with Jäschke et al. and Zhang et al.

Work	Version	Threads	Time ($T = 10$)
Jäschke et al. [15]	exact	one	363.90 days
Jäschke et al. [15]	approximate	one	154.70 h
Zhang et al. [32]	–	one	1606.36 s
This work	Protocol 6	one	554.68 s

Now we compare our protocol with Zhang et al.'s [32] further. For fairness, we use the same parameters as Zhang et al., and the run time is presented in Table 5. From the results, our protocol is about 2× to 3× faster than Zhang et al.. And note that theirs is not a completely outsourced scheme, requiring decryption and re-encryption after only one or two iterations.

Table 5. Compare with Zhang et al. ($T = 5$ for G2 and $T = 10$ for FCPS)

Dataset	Zhang et al. (s)	Ours (s)	Speedup
G2-1-20	222.41	111.36	2.0×
G2-2-20	221.11	114.35	1.9×
G2-4-20	250.20	117.96	2.1×
G2-8-20	311.55	124.19	2.5×
G2-16-20	441.89	139.22	3.2×
Chainlink	421.09	231.19	1.8×
EngyTime	394.87	227.50	1.7×
Hepta	1213.90	488.78	2.5×
Lsun	442.65	237.18	1.9×
Tetra	620.42	273.55	2.7×
TwoDiamonds	397.73	228.06	1.7×
WingNut	395.45	227.45	1.7×

5.4 Performance of COPPk-Means

In Table 6, we report the performance of Protocol 6 on all datasets mentioned earlier. It should be noted that the run time and memory consumed listed here are recorded by executing Protocol 6 faithfully, i.e. accomplish T iterations of

k-means with just one single round of interaction between the data owner and the computation server (without decryption and re-encryption). To our best knowledge, this is the first practical experimental result on the efficiency of a completely outsourced privacy-preserving k-means clustering protocol via FHE.

Table 6. Performance of COPPk-means ($T = 5$ for G2 and $T = 10$ for FCPS)

Dataset	Run Time (min)	Memory (GB)
G2-1-20	7.42	27.86
G2-2-20	7.54	29.16
G2-4-20	7.57	31.17
G2-8-20	7.67	31.80
G2-16-20	8.12	37.79
Chainlink	14.68	31.77
EngyTime	14.20	30.35
Hepta	16.16	36.46
Lsun	14.38	31.33
Tetra	14.54	32.13
TwoDiamonds	14.40	30.49
WingNut	14.77	31.71

6 Mini-batch K-means

For large-scale dataset clustering, mini batch k-means [28] is a significant method, and its algorithm on plaintexts is presented in Algorithm 7.

Algorithm 7. Mini-batch k-means on plaintexts

Input: s (size of batch, i.e. number of points in one batch), k (number of centroids), dataset $\boldsymbol{P}$.

Output: $(c_i)_{1\le i\le k}$ (updated centroids).

1: According s, split the dataset $\boldsymbol{P}$ into some small batches, suppose obtain q batches.
2: Randomly choose k points from first batch as initial centroids, denoted as $(c_i)_{1\le i\le k}$.
3: **for** $i = 1, \ldots, q$ **do**
4: Allocate data points in i-th batch to its nearest centroid among $(c_i)_{1\le i\le k}$, to obtain k clusters.
5: Update $(c_i)_{1\le i\le k}$ as the centroids of the k clusters just obtained.
6: **return** the updated k centroids $(c_i)_{1\le i\le k}$.

Protocol 6 can be readily adapted to support the mini-batch k-means. And to avoid ciphertext division, we still employ the stabilized variant, resulting in an encrypted version of the stabilized mini-batch k-means protocol. We omit the detailed description of this encrypted version here, we just call it **Protocol 8** for convenience.

We also implemented this Protocol 8 on Lattigo v5. To test the performance of Protocol 8 on large-scale datasets, we use PCA (principal component analysis) to extract 64 features from 784 features for all the 60000 samples in the training dataset of MNIST [18]. And then these 60000 samples are divided into 19 batches, each batch containing 3158 samples (the last batch contains 2 more samples selected randomly from the whole 60000 samples, to maintain the organization coherence of data information in ciphertexts). The experiment results in Table 7 demonstrate that Protocol 8 is capable of dealing with datasets of large scale.

Table 7. Performance of Protocol 8

Dataset size	Batch size	Total time	Time per batch	Memory
64×60000	64×3158	80.42 min	4.23 min	78.17 GB

7 Conclusion

In this paper, we present a protocol that achieves a completely outsourced privacy-preserving k-means clustering based on the CKKS FHE scheme. Our protocol only needs one round of interaction between the data owner and the computation server. It can accomplish limitless times of iteration of k-means clustering on the computation server side. We also give the mini-batch variant of our protocol, which is capable of dealing with large-scale datasets. Experiments based on our proof-of-concept implementation show that our protocols perform well in practice.

Acknowledgments. This work was partially supported by National Key Research and Development Program of China (2020YFA0712303), Natural Science Foundation of Chongqing (2022yszx-jcx0011cstb, cstb2023yszx-jcx0008), and Western Young Scholars Program of CAS.

References

1. Lattigo v5, February 2024. EPFL-LDS, Tune Insight SA. https://github.com/tuneinsight/lattigo
2. Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. J. Math. Cryptol. **9**(3), 169–203 (2015). https://doi.org/10.1515/jmc-2015-0016

3. Bossuat, J.P., Troncoso-Pastoriza, J., Hubaux, J.P.: Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation. In: Ateniese, G., Venturi, D. (eds.) ACNS 2022, LNCS, vol. 13269, pp. 521–541. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-09234-3_26
4. Brakerski, Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: Safavi-Naini, R., Canetti, R. (eds.) CRYPTO 2012. LNCS, vol. 7417, pp. 868–886. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32009-5_50
5. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed.) 3rd ITCS, pp. 309–325. ACM, New York (2012). https://doi.org/10.1145/2633600
6. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: Bootstrapping for approximate homomorphic encryption. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2018, pp. 360–384. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78381-9_14
7. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017. LNCS, vol. 10624, pp. 409–437. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70694-8_15
8. Cheon, J.H., Kim, D., Kim, D.: Efficient homomorphic comparison methods with optimal complexity. In: Moriai, S., Wang, H. (eds.) ASIACRYPT 2020. LNCS, vol. 12492, pp. 221–256. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64834-3_8
9. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: fast fully homomorphic encryption over the torus. J. Cryptol. **33**(1), 34–91 (2019). https://doi.org/10.1007/s00145-019-09319-x
10. Ducas, L., Micciancio, D.: FHEW: bootstrapping homomorphic encryption in less than a second. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9056, pp. 617–640. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_24
11. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive (2012). https://eprint.iacr.org/2012/144
12. Fränti, P., Sieranoja, S.: K-means properties on six clustering benchmark datasets. Appl. Intell. **48**(12), 4743–4759 (2018). https://doi.org/10.1007/s10489-018-1238-7
13. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing, 31 May 31–2 June 2009, Bethesda, USA, pp. 169–178. ACM, New York (2009). https://doi.org/10.1145/1536414.1536440
14. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: conceptually-simpler, asymptotically-faster, attribute-based. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8042, pp. 75–92. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40041-4_5
15. Jäschke, A., Armknecht, F.: Unsupervised machine learning on encrypted data. In: Cid, C., Jacobson Jr., M.J. (eds.) Selected Areas in Cryptography, SAC 2018. LNCS, vol. 11349, pp. 453–478. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-10970-7_21
16. Jha, S., Kruger, L., McDaniel, P.: Privacy preserving clustering. In: di Vimercati, S.d.C., Syverson, P., Gollmann, D. (eds.) Computer Security, ESORICS

2005, vol. 3679, pp. 397–417. Springer, Heidelberg (2005). https://doi.org/10.1007/11555827_23
17. Mohassel, P., Rosulek, M., Trieu, N.: Practical privacy-preserving k-means clustering (2019). https://eprint.iacr.org/2019/1158
18. LeCun, Y., Cortes, C., Burges, C.J.C.: The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/
19. Lee, JW., Lee, E., Lee, Y., Kim, YS., No, JS.: High-precision bootstrapping of RNS-CKKS homomorphic encryption using optimal minimax polynomial approximation and inverse sine function. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021. LNCS, vol. 12696, pp. 618–647. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77870-5_22
20. Lee, Y., Lee, J.W., Kim, Y.S., Kim, Y., No, J.S., Kang, H.: High-precision bootstrapping for approximate homomorphic encryption by error variance minimization. In: Dunkelman, O., Dziembowski, S. (eds.) EUROCRYPT 2022, LNCS, vol. 13275, pp. 551–580. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-06944-4_19
21. Li, B., Micciancio, D.: On the security of homomorphic encryption on approximate numbers. In: Canteaut, A., Standaert, F.X. (eds.) EUROCRYPT 2021, LNCS, vol. 12696, pp. 648–677. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-77870-5_23
22. Lu, W., Huang, Z., Hong, C., Ma, Y., Qu, F.: PEGASUS: bridging polynomial and non-polynomial evaluations in homomorphic encryption. In: IEEE S&P 2021, pp. 1057–1073. IEEE Computer Society, Los Alamitos (2021). https://doi.org/10.1109/SP40001.2021.00043
23. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. J. ACM **60**(6), 43:1–43:35 (2013). https://doi.org/10.1145/2535925
24. Rao, F.Y., Samanthula, B.K., Bertino, E., Yi, X., Liu, D.: Privacy-preserving and outsourced multi-user k-means clustering. In: Proceedings of the 2015 IEEE Conference on Collaboration and Internet Computing, pp. 80–89. IEEE, Los Alamitos (2015). https://doi.org/10.1109/CIC.2015.20
25. Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. J. ACM **56**(6), 34:1–34:40 (2009). https://doi.org/10.1145/1568318.1568324
26. Rivest, R., Adleman, L., Dertouzos, M.: On data banks and privacy homomorphisms. In: DeMillo, R.A., Dobkin, D.P., Jones, A.K., Lipton, R.J. (eds.) Foundations of Secure Computation, pp. 165–179. Academic Press, Atlanta (1978)
27. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
28. Sculley, D.: Web-scale k-means clustering. In: WWW '10, pp. 1177–1178. ACM, New York (2010). https://doi.org/10.1145/1772690.1772862
29. Ultsch, A.: Clustering wih SOM: U*C. In: Proceedings of the Workshop on Self-Organizing Maps (2005)
30. Vaidya, J., Clifton, C.: Privacy-preserving k-means clustering over vertically partitioned data. In: KDD '03, pp. 206–215. ACM, New York, USA (2003). https://doi.org/10.1145/956750.956776

31. Yongkai, F., et al.: PPMCK: privacy-preserving multi-party computing for k-means clustering. J. Parallel Distrib. Comput. **154**, 54–63 (2021). https://doi.org/10.1016/j.jpdc.2021.03.009
32. Zhang, M., Wang, L., Zhang, X., Liu, Z., Wang, Y., Bao, H.: Efficient clustering on encrypted data. In: Pöpper, C., Batina, L. (eds.) ACNS 2024, LNCS, vol. 14583, pp. 213–236. Springer, Cham (2024). https://doi.org/10.1007/978-3-031-54770-6_9