# Unsupervised Machine Learning on Encrypted Data

Angela Jäschke(✉) and Frederik Armknecht

University of Mannheim, Mannheim, Germany
{jaeschke,armknecht}@uni-mannheim.de

**Abstract.** In the context of Fully Homomorphic Encryption, which allows computations on encrypted data, Machine Learning has been one of the most popular applications in the recent past. All of these works, however, have focused on supervised learning, where there is a labeled training set that is used to configure the model. In this work, we take the first step into the realm of unsupervised learning, which is an important area in Machine Learning and has many real-world applications, by addressing the clustering problem. To this end, we show how to implement the $K$-Means-Algorithm. This algorithm poses several challenges in the FHE context, including a division, which we tackle by using a natural encoding that allows division and may be of independent interest. While this theoretically solves the problem, performance in practice is not optimal, so we then propose some changes to the clustering algorithm to make it executable under more conventional encodings. We show that our new algorithm achieves a clustering accuracy comparable to the original $K$-Means-Algorithm, but has less than 5% of its runtime.

**Keywords:** Machine Learning · Clustering
Fully Homomorphic Encryption

## 1 Introduction

### 1.1 Motivation

Fully Homomorphic Encryption (FHE) schemes can in theory perform arbitrary computations on encrypted data. Since the discovery of FHE, many applications have been proposed, ranging from medical over financial to advertising scenarios. The underlying idea is mostly the same: Suppose Alice has some confidential data $X$ which she would like to utilize, and Bob has an algorithm $\mathcal{A}$ which he could apply to Alice's data for money. However, conventionally, either Alice would have to give her confidential data to Bob, or run the algorithm herself, for which she may not have the know-how or computational power. FHE allows Alice to encrypt her data to $C := \mathrm{Enc}(X)$ and send it to Bob. Bob

can convert his algorithm $\mathcal{A}$ into a function $\mathcal{A}'$ over the ciphertext space and apply it to the encrypted data, resulting in $R := \mathcal{A}'(C)$. He can then send this result back to Alice, who can decrypt it with her secret key. FHE promises that indeed $\mathrm{Dec}(R) = \mathrm{Dec}(\mathcal{A}'(\mathrm{Enc}(X))) = \mathcal{A}(X)$. Since Alice's data was encrypted the whole time, Bob learns nothing about the data entries. Note that the functionality where Bob's algorithm is also kept secret from Alice is not traditionally guaranteed by FHE, but can in practice be achieved via a property called *circuit privacy*, in the sense that Alice learns nothing except the result $\mathcal{A}(X)$.

One of the most popular applications of FHE has been Machine Learning, with many works focusing on Neural Networks and different variants of regression. To our knowledge, all works in this line are concerned with *supervised* learning. This means that there is a training set with known outcomes, and the algorithm tries to build a model that matches the desired outputs to the inputs as well as possible. When the training phase is done, the algorithm can be applied to new instances to predict unknown outcomes. However, there is a second branch in Machine Learning that has not been touched by FHE research: *Unsupervised* learning. For these kinds of algorithms, there are no labeled training examples, there is simply a dataset on which some kind of analysis shall be performed. An example of this is clustering, where the aim is to group data entries that are similar in some way. The number of clusters might be a parameter that the user enters, or it may be automatically selected by the algorithm. Clustering has numerous applications like genome sequence analysis, market research, medical imaging or social network analysis, to name a few, some of which inherently involve sensitive data – making a privacy-preserving evaluation with FHE even more interesting.

### 1.2   Contribution

In this work, we approach this unexplored branch of Machine Learning and show how to implement the $K$-Means-Algorithm, an important clustering algorithm, on encrypted data. We discuss the problems that arise when trying to evaluate the $K$-Means-Algorithm on encrypted data, and show how to solve them. To this end, we first present a natural encoding that allows the execution of the algorithm as it is (including the usually challenging division by an encrypted value), but is not optimal in terms of performance. We then present a modification to the $K$-Means-Algorithm that performs comparably in terms of clustering accuracy, but is much more FHE-friendly in that it avoids division by an encrypted value. We include another modification that trades accuracy for efficiency in the involved comparison operation, and compare the runtimes of these approaches.

## 2   Related Work

Encryption schemes that allow one type of operation on ciphertexts have been around for some time and have a comprehensive security characterization [3]. Fully Homomorphic Encryption however, which allows both unlimited additions

and multiplications, was only first solved in [19]. Since then, many other schemes have been developed, for example [8,12–15,18,20,37], to name just a few. An overview can be found in [2]. There are several libraries offering FHE implementations, like [11,16,23], and the one we use, [38].

Machine Learning as an application of FHE was first proposed in [35], and subsequently there have been numerous works on the subject, to our knowledge all concerned with supervised learning. The most popular of these applications seem to be (Deep) Neural Networks (see [7,10,21,26,36]) and (Linear) Regression (e.g., [4,17,32] or [22]), though there is also some work on other algorithm classes like decision trees and random forests [41], or logistic regression ([5,6,29,30]). In contrast, our work is concerned with the clustering problem from unsupervised Machine Learning.

The $K$-Means-Algorithm has been a subject of interest in the context of privacy-preserving computations for some time, but to our knowledge all previous works like [9,24,25,31,42] require interaction between several parties, e.g. via Multiparty Computation (MPC). For a more comprehensive overview of the $K$-Means-Algorithm in the context of MPC, we refer the reader to [34]. While this interactivity may certainly be a feasible requirement in many situations, and indeed MPC is likely to be faster than FHE in these cases, we feel that there are several reasons why a non-interactive solution as we present it is an important contribution.

1. **Client Economics:** In MPC, the computation is split between different parties, each performing computations every round and combining the results. In FHE computations, the entire computation is performed by the service provider. Even if this computation on encrypted data is more expensive than the total MPC computation, the client reduces his effort to zero this way, making this solution attractive to him and thus generating a demand for it.
2. **Function Privacy:** Imagine the $K$-Means-Algorithm in this paper as a placeholder for a more complex proprietary algorithm that the service provider executes on the client's data as a service. This algorithm could utilize building blocks from the $K$-Means-Algorithm that we present in this paper, or involve the $K$-Means-Algorithm as a whole in the context of pipelining several algorithms together, or be something completely new. Here, the service provider would want to prevent the user from learning the details of this algorithm, as it is his business secret. While FHE per se does not guarantee this functionality, all schemes today fulfill the requirement of *circuit privacy* needed to achieve it. Thus for this case, FHE would be the preferred solution.
3. **Future Efficiency Gain:** MPC is much older than FHE, and efficiency for the latter has increased by a factor of $10^4$ in the last six years alone. To argue that MPC is faster and thus FHE solutions are superfluous seems premature at this point, and our contributions are not specific to any implementation, but work on all FHE schemes that support a $\{0,1\}$ plaintext space.

Also, many of these interactive solutions rely on a vertical (in [40]) or horizontal (in [28]) partitioning of the data for security. In contrast, FHE allows a non-interactive setting with a single database owner who wishes to outsource the computation.

# 3    Preliminaries

In this section, we cover underlying concepts like the $K$-Means-Algorithm, encoding issues, our choice of implementation library, and the datasets we use.

## 3.1    The K-Means Algorithm

The $K$-Means-Algorithm is one of the most well-known clustering algorithms in unsupervised learning. Published in [33], it is considered an important benchmark algorithm and is frequently the subject of current research to this day. It takes as input the data $X = \{x_1, \ldots, x_m\}$ and a number $K$ of clusters to be used, and begins by choosing $K$ randomly chosen data entries as so-called **cluster centroids** $c_k$. Then, in a step called `Cluster Assignment`, it computes for each data entry $x_i$ which cluster centroid $c_k$ is nearest, and assigns the data entry to that centroid. When this has been done for all data entries, the second step begins: During the `Move Centroids` step, the cluster centroids are moved by setting each centroid as the average of all data entries that were assigned to it in the previous step. These two steps are repeated for a set number of times $T$ or until the centroids do not change anymore. We use the first method.

The output of the algorithm is the values of the centroids, or the cluster assignment for the data entries (which can easily be computed from the former). We opt for the first approach. The pseudocode for the algorithm as we use it can be found in Appendix A, along with a visualization. Accuracy can either be measured in terms of correctly classified data entries, which assumes that the correct classification is known (there might not even exist a unique best solution), or via the so-called cost function, which measures the (average) distance of the data entries to their assigned cluster centroids. We opt for the first approach because our datasets are benchmarking sets for which the labels are indeed provided, and it allows better comparability between the different algorithms.

## 3.2    Encoding

FHE schemes generally have finite fields as a plaintext space, and any rational numbers (which can be scaled to integers) must be embedded into this plaintext space. There are two main approaches in literature, which we quickly compare side by side in Table 1. Note that for absolute value computation and comparison, we need to use the digitwise encoding.

## 3.3    FHE Library Choice

In [27], it was shown that among all bases $p$ for digitwise $p$-adic encoding in FHE computations, the choice $p = 2$ is best in terms of the number of additions and multiplications to be performed on the ciphertexts. Hence, we use an FHE scheme with a plaintext space of $\{0, 1\}$. The currently fastest FHE implementation for this plaintext space, TFHE [38], states that *"an optimal circuit for*

**Table 1.** Two mainstream encoding approaches.

| | Digitwise | Embedded |
|---|---|---|
| Description | For a base $p$, display the number in $p$-adic[a] representation and encrypt each digit separately | Choose the plaintext space large enough to accommodate all computations |
| Supports comparison? | ✓ | × |
| Supports absolute value? | ✓ | × |
| Supports division? | × | × |
| Efficiency | Slower | Faster |
| Flexibility | Full | The function that is being computed must be known (at least a bound) at setup, as computations fail if the result gets too big. This is actually *Somewhat* Homomorphic Encryption, not *Fully* Homomorphic Encryption |

[a] This can be extended to plaintext spaces $GF(p^k)$ if the scheme supports them.

TFHE is most likely a circuit with the smallest possible number of gates" – thus, this library is a perfect choice for us, and we will use the binary encoding for signed integers and tweaks presented in [26] for maximum efficiency.

### 3.4   Datasets

To evaluate performance, we use four datasets from the FCPS dataset [39]:

– The Hepta dataset consists of 212 data points of 3 dimensions. There are 7 clearly defined clusters.
– The Lsun dataset is 2-dimensional with 400 entries and 3 classes. The clusters have different variances and sizes.
– The Tetra dataset is comprised of 400 entries in 3 dimensions. There are 4 clusters, which almost touch.
– The Wingnut dataset has only 2 clusters, which are side-by-side rectangles in 2-dimensional space. There are 1016 entries.

For accuracy measurements, each version of the algorithm was run 1000 times (with varying starting centroids) for number of iterations $T = 5, 10, ..., 45, 50$ on each dataset. For runtimes on encrypted data, we used the Lsun dataset.

# 4  Approach 1: Implementing the Exact $K$-Means-Algorithm

We now show a method of implementing the K-Means algorithm largely as it is. To this end, we first discuss challenges that arise in the context of FHE computation of this algorithm. We then address these challenges by changing the distance metric, and then present an encoding that supports the division required in computing the average in the `MoveCentroid`-step. As this method is in no way restricted to the $K$-Means-Algorithm, the result is of independent interest. As it turns out, there are some issues with this approach, which we will also discuss.

## 4.1  FHE Challenges

Fully homomorphic encryption schemes can easily compute additions and multiplications on the underlying plaintext space, and most also offer subtraction. Using these operations as building blocks, more complex functionalities can be obtained. However, there are three elements in the $K$-Means-Algorithm that pose challenges, as it is not immediately clear how to obtain them from these building blocks. We list these (with the line numbers referring to the pseudocode on page 20 in Appendix A.2) and quickly explain how we solve them.

- The distance metric (Line 9, $\Delta(x, y) = ||x - y||_2 := \sqrt{\sum_i (x_i - y_i)^2}$): To our knowledge, taking the square root of encrypted data has not been implemented yet. In Sect. 4.2, we will argue that the Euclidean norm is an arbitrary choice in this context and solve this problem by using the $L_1$-distance $\Delta(x, y) = ||x - y||_1 := \sum_i (|x_i - y_i|)$ instead of the Euclidean distance.
- Comparison (Line 10, $\tilde{\Delta} < \Delta$) in finding the centroid with the smallest distance to the data entry: This has been constructed from bit multiplications and additions in [26] for bitwise encoding, so we view this issue as solved. A detailed explanation can be found in the extended version of this paper.
- Division (Line 25, $c_k = c_k/d_k$) in computing the new centroid value as the average of the assigned data points: In FHE computations, division by an encrypted value is usually not possible (whereas division by an unencrypted value is no problem). We present a way of implementing the division with a new encoding in Sect. 4.3, and propose a modified version of the Algorithm in Sect. 5 that only needs division by a constant.

## 4.2  The Distance Metric

Traditionally, the distance measure used with the $K$-Means Algorithm is the Euclidean Distance $\Delta(x, y) = ||x - y||_2 := \sqrt{\sum_i (x_i - y_i)^2}$, also known as the $L_2$-Norm, as it is analytically smooth and thus reasonably well-behaved. However, in the context of $K$-Means Clustering, smoothness is irrelevant, and we

may look to other distance metrics. Concretely, we consider the $L_1$-Norm[1] (also known as the Manhattan-Metric) $\Delta(x, y) := \sum_i (|x_i - y_i|)$. This has a considerable advantage over the Euclidean distance: Firstly, we do not need to take a square root, which to our knowledge has not yet been achieved on encrypted data. Secondly, of course one could apply the standard trick and not take the root, working instead with the sum of squared distances. However, this would mean a considerable efficiency loss due to numerous multiplications and the greatly increased bitlengths of their results. These long numbers are then summed up, and the result is input into the algorithm that finds the minimum (Algorithm 2 on page 12). These two steps already constitute bottlenecks in the entire computation when working with short numbers in the $L_1$ norm, so an increase in the bitlengths would greatly increase computation time.

Taking the absolute value can easily be achieved through a digit-wise encoding like the binary encoding which we use: We can use the MSB as the conditional (it is 1 if the number is negative and 0 if it is positive) and use a multiplexer[2] gate applied to the value and its negative. The concrete algorithm can be seen in the extended version of this paper. Thus, using the $L_1$-Norm is not only justified by the arbitrariness of the Euclidean Norm, but is also much more efficient. We compare the clustering accuracy in Fig. 1.
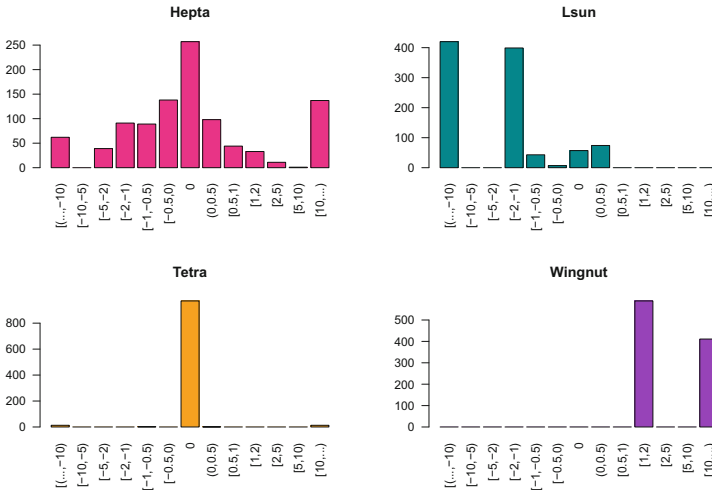


**Fig. 1.** Difference in percent of data points mislabeled for $L_1$-norm compared to the $L_2$-norm $\big((\% \text{ mislabeled } L_1) - (\% \text{ mislabeled } L_2)\big)$.

---

[1] [1] in fact argues that for high-dimensional spaces, the $L_1$-Norm is more meaningful than the Euclidean Norm.

[2] $\text{MUX}(c, a, b) = \begin{cases} a, & c = 1 \\ b, & c = 0 \end{cases}$.

For both versions of the distance metric, we calculated the percentage of wrongly labeled data points for 1000 runs, which we can do because the datasets we use come with the correct labels. We plotted histograms of the difference (in percent mislabeled) between the $L_1$-norm and the $L_2$-norm for each run. Thus, a value of 0.5 means that the $L_1$ norm version misclassified 0.5% more data entries than the $L_2$-version, and $-2$ means that the $L_1$ version misclassified 2% less entries than the $L_2$-version. Each subplot corresponds to one of the four datasets. We see that indeed, it is impossible to say which metric is better – for the Hepta dataset, the performance is very balanced, for the Lsun dataset, the $L_1$-norm performs much better, for the Tetra dataset, they nearly always perform exactly the same, and for the Wingnut dataset, the $L_2$-norm is consistently better.

### 4.3   Fractional Encoding

Suppose we have routines to perform addition, multiplication and comparison on bitwise encoded numbers. The idea is to express the number we wish to encode as a fraction and encode the numerator and denominator separately. Concretely, we choose the denominator $a_d$ randomly in a certain range (like $a_d \in [2^k, 2^{k+1})$ for some $k$) and compute the nominator $a_n$ as $a_n = \lfloor a \cdot a_d \rceil$. We then encode both separately, so we have $a = (a_n, a_d)$. If we then want to perform computations (including division) on values encoded in this way, we can express the operations using the subroutines from the binary encoding through the regular computation rules for fractions. The details can be seen in Appendix B.

**Controlling the Bitlength.** Every single one of these operations requires a multiplication of some sort, which means that the bitlengths of the nominators and denominators double with each operation, as there is no cancellation when the data is encrypted. However, in bitwise encoding, deleting the last $k$ least significant bits corresponds to dividing by $2^k$ and truncating. Doing this for both nominator and denominator yields roughly the same result as before, but with lower bitlengths. As an example, suppose that we have encoded our integers with 15 bits, and after multiplication we thus have 30 bits in nominator and denominator, e.g. $651049779/1053588274 \approx 0.617936$. Then dividing both nominator and denominator by $2^{15}$ and truncating yields $19868/32152$, which evaluates to $0.617939 \approx 0.617936$. The accuracy can be set through the original encoding bitlength (15 here).

### 4.4   Evaluation

While this new encoding theoretically allows us to perform the $K$-Means-Algorithm and solves the division problem in FHE, we now discuss the practical performance in terms of accuracy and runtime.

**Accuracy.** To see how the exact algorithm performs, we use the four datasets from Sect. 3.4. We ran the exact algorithm 1000 times for number of iterations

$T = 5, 10, ..., 45, 50$, and for sake of completeness we include both distance metrics. The results in this section were obtained by running the algorithms in unencrypted form. We first examine the effect of $T$ on the exact version of the algorithm by looking at the average (over the 1000 runs) misclassification rate for both metrics. The result can be seen in Fig. 2 – we see that the rate levels off after about 15 rounds in all cases, so there is no reason to iterate further.
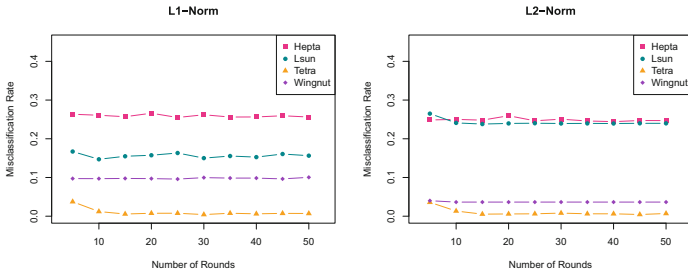


**Fig. 2.** Misclassification rate with increasing rounds for exact algorithms.

In practice, however, our Fractional Encoding does have some problems: The first issue is the procedure to shorten the bitlengths from Subsect. 4.3. While it works reasonably well for short computations, we found it nearly impossible to set the number of bits to delete such that the entire algorithm ran correctly. The reason is simple: If not enough bits are cut off, the bitlength grows, propagating with each operation and resulting in an overflow when the number becomes too large for the allocated bitlength. If too many bits are cut off, one loses too much accuracy or may even end with a 0 in the denominator. Both these cases result in completely arbitrary and unusable results. The reason why it is so hard to set the shortening parameter properly is that generally, nominator and denominator will not require the same number of bits. Also, because the data is encrypted, we cannot see the actual size of the underlying data, so the shortening parameter cannot be set dynamically – in fact, if this were possible, it would imply that the FHE scheme is insecure. Even setting the parameter roughly requires extensive knowledge about the encrypted data, which the data owner may not want to share with the computing party.

**Runtime.** The second issue with this encoding is the runtime. Even though TFHE is the most efficient FHE library with which many computational tasks approach practically feasible runtimes, the fact that this encoding requires several multiplications on binary numbers for each elementary operation slows it down considerably. We compare the runtimes of all our algorithms in Sect. 7, and as we will see, running the $K$-Means-Algorithm on a real-world dataset with this Fractional Encoding would take almost 1.5 years on our computer.

### 4.5    Conclusion

In conclusion, this encoding is theoretically possible, but we would not recommend it for practical use due to its inefficiency and hardness of setting the shortening parameter (or even higher inefficiency if little to no shortening is done). However, for very flat computations (in the sense that there are not many successive operations performed), this encoding that allows division may still be of interest. For the $K$-Means-Algorithm, we instead change the algorithm in a way that avoids the problematic division, which we present in the rest of this paper.

## 5    Approach 2: The Stabilized $K$-Means-Algorithm

In this section, we present a modification of the $K$-Means algorithm that avoids the division in the `MoveCentroid`-step. Recall that conventional encodings in FHE, like the binary one we will use, do not allow the computation of $c_1/c_2$ where $c_1$ and $c_2$ are ciphertexts, but it is possible to compute $c_1/a$ where $a$ is some unencrypted number. We use this fact to exchange the ciphertext division in Line 25 of Algorithm 3 (page 20) for a constant division, resulting in a variant that can be computed with more established and efficient encodings than the one from Sect. 4.3. We present this new algorithm in Sect. 5.2, and compare the accuracy of the results to the original $K$-Means-Algorithm in Sect. 5.3.

### 5.1    Encoding

The dataset we use to evaluate our algorithms consists of rational numbers. To encode these so that we can encrypt them bit by bit, we scaled them with a factor of $2^{20}$ and truncated to obtain an integer. We then used Two's Complement encoding to accommodate signed numbers, and switched to Sign-Magnitude Encoding for multiplication. Note that deleting the last 20 bits corresponds to dividing the number by $2^{20}$ and truncating, so the scaling factor can remain constant even after multiplication, where it would normally square.

### 5.2    The Algorithm

Recall that in the original $K$-Means-Algorithm, the `MoveCentroid`-step consists of computing each centroid as the average of all data entries that have been assigned to it. More specifically, suppose that we have a $(m \times K)$-dimensional cluster assignment matrix $A$, where

$$A_{ik} = \begin{cases} 1, & \text{Data entry } x_i \text{ is assigned to centroid } c_k \\ 0 & \text{else.} \end{cases}$$

Then computing the new centroid value $c_k$ consists of multiplying the data entries $x_i$ with the corresponding entry $A_{ik}$ and summing up the results before dividing by the sum over the respective column $k$ of $A$:

$$c_k = \sum_{i=1}^{m} x_i \cdot A_{ik} \Big/ \sum_{i=1}^{m} A_{ik}.$$

---

**Algorithm 1.** The Stabilized $K$-Means-Algorithm

---

**Input**: Data set $X = \{x_1, \ldots, x_m\}$ // $x_i \in \mathbb{R}^\ell$ for some $\ell$
**Input**: Number of clusters $K$
**Input**: Number of iterations $T$
// Initialization
1   Randomly reorder $X$;
2   Set centroids $c_k = x_k$ for $k = 1$ to $K$;
// Keep track of centroid assignments
3   Generate $(m \times K)$-dimensional boolean matrix $A$ set to 0;
4   **for** $j = 1$ *to* $T$ **do**
     // Cluster Assignment
5      **for** $i = 1$ *to* $m$ **do**
6         $\Delta = \infty$;
7         **for** $k = 1$ *to* $K$ **do**
           // Compute distances to all centroids
8            $\Delta_k := ||x_i - c_k||_1$;
9         **end**
        // The $i^{th}$ row of $A$ has all 0's except at the column corresponding to the centroid with the minimum distance
10        $A[i, \cdot] \leftarrow \texttt{FindMin}(\Delta_1, \ldots, \Delta_K)$;
11      **end**
     // Move Centroids
12      **for** $k = 1$ *to* $K$ **do**
        // Keep old centroid value
13         $\bar{c}_k = c_k$;
14         $c_k = 0$;
15         **for** $i = 1$ *to* $m$ **do**
           // If $A_{ik} == 1$, add $x_i$ to $c_k$, otherwise add $\bar{c}_k$ to $c_k$
16            $c_k = c_k + \texttt{MUX}(A_{ik}, x_i, \bar{c}_k)$;
17         **end**
        // Divide by number of terms $m$
18         $c_k = c_k/m$
19      **end**
20   **end**
    **Output**: $\{c_1, \ldots, c_K\}$

---

Our modification now replaces this procedure with the following idea: To compute the new centroid $c_k$, add the corresponding data entry $x_i$ to the running sum if $A_{ik} = 1$, otherwise add the old centroid value $\bar{c}_k$ if $A_{ik} = 0$. This can be easily done with a multiplexer gate (or more specifically, by abuse of notation, a multiplexer gate applied to each bit of the two inputs) with the entry $A_{ik}$ as the conditional boolean variable:

$$c_k = \sum_{i=1}^{m} \texttt{MUX}(A_{ik}, x_i, \bar{c}_k)\big/m.$$

The sum now always consists of $m$ terms, so we can divide by the unencrypted constant $m$. It is also now obvious why we call it the *stabilized K*-Means-Algorithm: We expect the centroids to move much more slowly, because the old centroid values stabilize the value in the computation. The details of this new algorithm can be found in Algorithm 1, with the changes compared to the original $K$-Means-Algorithm shaded.

**Computing the Minimum.** As the reader may have noticed in Line 10, we have replaced the comparison step in finding the nearest centroid for a data entry with a new function $\text{FindMin}(\Delta_1, \ldots, \Delta_K)$ due the change in data structure of $A$ (from an integer vector to a boolean matrix). This new function outputs

$$A[i, \cdot] \leftarrow \text{FindMin}(\Delta_1, \ldots, \Delta_K)$$

such that the $i^{th}$ row of $A$, $A[i, \cdot]$, has all 0's except at the column corresponding to the centroid with the minimum distance to $x_i$. The idea is to run the $\text{Compare}$ circuit to obtain a Boolean value: $\text{Compare}(x, y) = 1$ if $x < y$, and 0 otherwise.

We start by comparing the first two distances $\Delta_1$ and $\Delta_2$ and setting the Boolean value as $C := \text{Compare}(\Delta_1, \Delta_2)$. Then we can write $A[i, 1] = C$ and $A[i, 2] = \neg C$ and keep track of the current minimum through $\text{minval} := \text{MUX}(C, \Delta_1, \Delta_2)$. We then compare $\text{minval}$ to $\Delta_3$ etc. until we have reached $\Delta_K$. Note that we need to modify all entries $A[i, k]$ with $k$ smaller than the current index by multiplying them with the current Boolean value, preserving the indices if the minimum doesn't change through the comparison, and setting them to 0 if it does. The exact workings can be found in Algorithm 2, and an example of how the algorithm works can be found in the extended version of this paper.

If the encryption scheme is one where multiplicative depth is important, it is easy to modify $\text{FindMin}$ to be depth-optimal: Instead of comparing $\Delta_1$ and $\Delta_2$, then comparing the result to $\Delta_3$, then comparing that result to $\Delta_4$ etc., we could instead compare $\Delta_1$ to $\Delta_2$ and $\Delta_3$ to $\Delta_4$ and then compare those two results etc., reducing the multiplicative depth from linear in the number of clusters $K$ to logarithmic. Since depth is not important for our implementation choice TFHE, we implemented the function as described in Algorithm 2.

---

**Algorithm 2.** $\text{FindMin}(\Delta_1, \ldots, \Delta_K)$

**Input**: Distances $\Delta_1, \ldots, \Delta_K$ of current data entry $i$ to all centroids $c_1 \ldots, c_K$
**Input**: Row $i$ of Cluster Assignment matrix $A$, denoted $A[i, \cdot]$
// Set all entries 0 except the first
1   Set $A[i, \cdot] = [1, 0, \ldots, 0]$;
    // Set the minimum to $\Delta_1$
2   Set $\text{minval} = \Delta_1$;
3   **for** $k = 2$ *to* $K$ **do**
        // C is a Boolean value, C = 1 iff minval $\leq \Delta_k$
4         $C = \text{Compare}(\text{minval}, \Delta_k)$;
5         **for** $r = 1$ *to* $k - 1$ **do**
                // Set all previous values to 0 if new min is $\Delta_k$, don't change if new min is old min
6                 $A[i, r] = A[i, r] \cdot C$;
7         **end**
        // Set $A[i, k]$ to 1 if $\Delta_k$ is new min, 0 otherwise
8         $A[i, k] = \neg C$ ;
9         **if** $k \neq K$ **then**
                // Update the minval variable unless we're done
10           $\text{minval} = \text{MUX}(C, \text{minval}, \Delta_k)$;
11         **end**
12 **end**
**Output**: $A[i, \cdot]$

### 5.3    Evaluation

In this section, we will investigate the performance of our Stabilized $K$-Means-Algorithm compared to the traditional $K$-Means-Algorithm.

**Accuracy.** The results in this section were obtained by running the algorithms in unencrypted form. As we are interested in relative rather than absolute performance, we merely care about the difference in the output of the modified and exact algorithms on the same input (i.e., datasets and starting centroids), not so much about the output itself. Recall that we obtained $T = 15$ as a good choice for number of rounds for the exact algorithm – however, as we have already explained above, the cluster centroids converge more slowly in the stabilized version, so we will likely need more iterations here. We now compare the performance of the stabilized version to the exact version. We perform this comparison by examining the average (over the 1000 iterations) difference in the misclassification rate. Thus, a value of 2 means that the stabilized version mislabeled 2% more instances than the exact version, and a difference of $-1$ means that the stabilized version misclassified 1% less data points than the exact version.
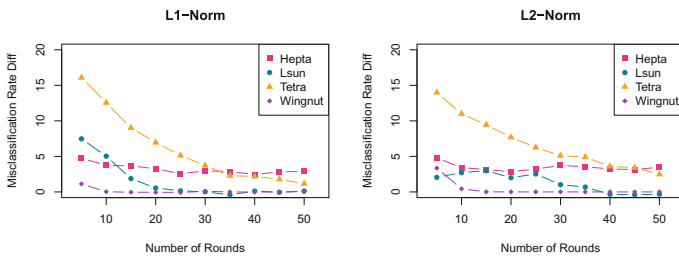


**Fig. 3.** Average difference in misclassification rate between the stabilized and the exact algorithm ((average % mislabeled stabilized) − (average % mislabeled exact)).

The results for both distance metrics can be seen in Fig. 3. We see that while behavior varies slightly depending on the dataset, $T = 40$ iterations is a reasonable choice since the algorithms do not generally seem to converge further with more rounds. We will fix this parameter from here on, as it also exceeds the required amount of iterations for the exact version to converge.

While the values in Fig. 3 do converge, they do not generally reach a difference of 0, which would imply similar performance. However, this is not surprising - we significantly modified the original algorithm, not with the intention of improving clustering accuracy, but rather to make it executable under an FHE scheme at all. This added functionality comes as a tradeoff, and we will now examine the magnitude of the loss in accuracy in Fig. 4. The corresponding histogram for the $L_2$-norm can be found in the extended version of this paper.

We can see that in the vast majority of instances, the stabilized version performs exactly the same as the the original $K$-Means-Algorithm. We also see that
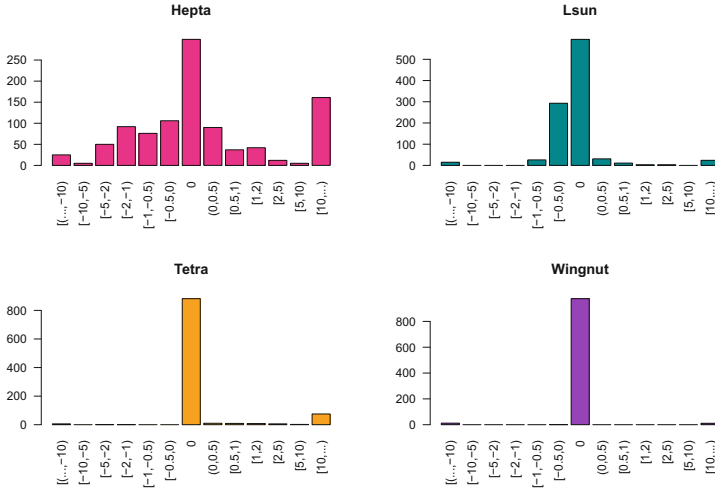
**Fig. 4.** Distribution of the difference in misclassification rate for stabilized vs. exact $K$-Means-Algorithm $\big((\% \text{ mislabeled stabilized}) - (\% \text{ mislabeled exact})\big)$, $L_1$-norm.

concrete performance does depend on the dataset. In some cases, the modified version even outperforms the original one: Interestingly, for the Lsun dataset, the stabilized version is actually slightly better than the original algorithm in about 30% of the cases. However, most of the time, we feel that there will be a slight performance decrease. The fact that there are some outliers where performance is drastically worse can easily be solved by running the algorithm several times in parallel, and only keeping the best run. This can be done under homomorphic encryption much like computing the minimum in Sect. 5.2, but will not be implemented in this paper.

**Runtime.** While we will have a more detailed discussion of the runtime of all our algorithms in Sect. 7, we would like to already present the performance gain at this point: Recall that we estimated that running the exact algorithm from Sect. 4 would take almost 1.5 years. In contrast, our Stabilized Algorithm can be run in 25.93 days, or less than a month. This is less than 5% of the runtime of the exact version.

**Conclusion.** In conclusion to this section, we feel that by modifying the $K$-Means-Algorithm, we have traded a very small amount of accuracy for the ability to perform clustering on encrypted data in a more reasonable amount of time, which is a functionality that has not been achieved previously. The next section will deal with an idea to improve runtimes even more.

## 6    Approach 3: The Approximate Version

We now present another modification which trades in a bit of accuracy for improved runtime. Due to space constraints, the details have been moved to Appendix C and we give only a high-level sketch at this point: Since the `Compare` function is linear in its inputs lengths, speeding up this building block would make the entire computation more efficient. First recall that we encode our numbers bitwise after having scaled them to integers. This means that we have access to the individual bits and can delete the $S$ least significant bits, which corresponds to dividing the number by $2^S$ and truncating. Let $\tilde{X}$ denote this truncated version of a number $X$, and $\tilde{Y}$ that of a number $Y$. Then $\texttt{Compare}(\tilde{X}, \tilde{Y}) = \texttt{Compare}(X, Y)$ if $|X - Y| \geq 2^S$, and may or may not return the correct result if $|X - Y| < 2^S$. However, correspondingly, if the result is wrong, the centroid that is wrongly assigned to the data entry is no more than $2^S$ further from the data entry than the correct one. We propose to pick an initial $S$ and decrease it over the course of the algorithm, so that accuracy increases as we near the end. We call this variant of the (stabilized) algorithm the *approximate* version.

In our experiments with $S = 5$, we saw that accuracy is comparable to the stabilized version, and the gain is around 210.7 min for the entire algorithm. Unfortunately, this is swallowed by the magnitude of the total computation time, as the main bottlenecks lie elsewhere. However, running just the comparison and approximate comparison functions with the same parameters as in our implementation of the $K$-Means-Algorithm (35 bits, 5 bits deleted for approximate comparison) yielded a drop in average runtime from 3.24 to 1.51 s. We see that this does make a big difference and may be of independent interest for computations involving many comparisons, which is why we choose to present the modification even though the effect was outweighed by other bottlenecks in the $K$-Means-Algorithm computation.

## 7    Implementation Results

We now present runtimes for the stabilized and approximate versions of the $K$-Means-Algorithm, and the times for the exact version using Fractional Encoding. Computations were done in a virtual machine with 20 GB of RAM and 4 cores, running an Intel i7-3770 processor with 3.4 GHz. We used the TFHE library [38] without the SPQLIOS_FMA-option, as our processor did not support this.

The dataset we used was the Lsun dataset from [39], which consists of 400 rational data entries of 2 dimensions, and $K = 3$ clusters. We encoded the binary numbers with 35 bits and scaled to integers using $2^{20}$. The timings measured

were for one round, and the approximate version used a deletion parameter of $S = 5$. For the Fractional Encoding, the data was encoded with nominator in $[2^{11}, 2^{12})$ and denominator in roughly the same range. We allotted 35 bits total for nominator and denominator each to allow a growth in required bitlength, and set the shortening parameter to 12, but shortened by 11 every once in a while (we derived this approach experimentally, see the discussion of the shortcoming of this approach in Sect. 4.4). The Fractional exact version was so slow that we ran it only on the first 10 data entries of the dataset - we will extrapolate the runtimes in Sect. 7.1.

## 7.1   Runtimes for the Entire Algorithm on a Single Core

We now present the runtimes for the entire $K$-Means-Algorithm on encrypted data on our specific machine with single-thread computation. There is some extrapolation involved, as the measured runtimes were for one round (so we multiplied by the round number, which differs between the exact version and the other two), and in the Fractional (exact) case, only for 10 data entries, so we multiplied that time by 40. Note that these times (which are with no parallelization) can be found in Table 2. We see that even though the stabilized version needs more rounds than the exact version, the latter is still significantly slower due to the Fractional Encoding. The approximate version (always with $S = 5$ deleted bits in the comparison) would save about 210.7 min.

**Table 2.** Single-thread runtimes (extrapolated) on our machine.

|                   | Exact (fractional)              | Stabilized                   | Approximate                  |
| ----------------- | ------------------------------- | ---------------------------- | ---------------------------- |
| Runtime per round | 873.46 h (36.39 days)           | 15.56 h                      | 15.47 h                      |
| Rounds required   | 15                              | 40                           | 40                           |
| Total runtime     | 545.91 days $\approx$ 17.95 months | 25.93 days $\approx$ 0.85 months | 25.79 days $\approx$ 0.85 months |

## 7.2   Further Speedup

We would now like to address the subject of parallelism. At the moment (last accessed April $24^{th}$ 2018), the TFHE library only supplies single-thread computations - i.e., there is no parallelism. However, version 1.5 is expected soon, and this will allegedly support multithreading. We first explain the huge difference this would make for the runtime, and then quantify the involved timings.

**Parallelism.** It is easy to see that all our versions of the $K$-Means-Algorithm are highly parallelizable: The `Cluster Assignment` step trivially so over the data entries (without any time needed for recombination), and the `Move Centroids` similarly over the cluster centroids (also over the data entries with very small recombination effort). Since both steps are linear in the number $K$ of centroids, the number $m$ of data entries, and the number $T$ of round iterations, we present our runtimes in this subsection as *per centroid, per data entry, per round, per core*. This allows a flexible estimate for when multithreading is supported.

**Round Runtimes.** We now present the runtime results for each of the three variants on encrypted data per centroid, per data entry, per round, per core in Table 3. We do not include runtimes for encoding/encryption and decryption/decoding, as these would be performed on the user side, whereas the computation would be outsourced (encoding/encryption is ca. 1.5 s, and decoding/decryption is around 5 ms). We see that the Fractional Encoding is extremely slow, which motivated the Stabilized Algorithm in the first place.

**Table 3.** Runtimes per centroid, per data entry, per round, per core.

|  | Exact (fractional) | Stabilized | Approximate |
|---|---|---|---|
| `Cluster Assignment` | 1650.91 s $\approx$ 27.5 min | 35.59 s | 35.39 s |
| `Move Centroids` | 969.47 s $\approx$ 16.2 min | 11.09 s | 11.03 s |
| Total | 2620.38 s $\approx$ 43.7 min | 46.68 s | 46.42 s |

# A    Supplementary Material for the $K$-Means-Algorithm

This appendix contains some supplemental material for the $K$-Means-Algorithm.

## A.1    Visualization of the $K$-Means-Algorithm

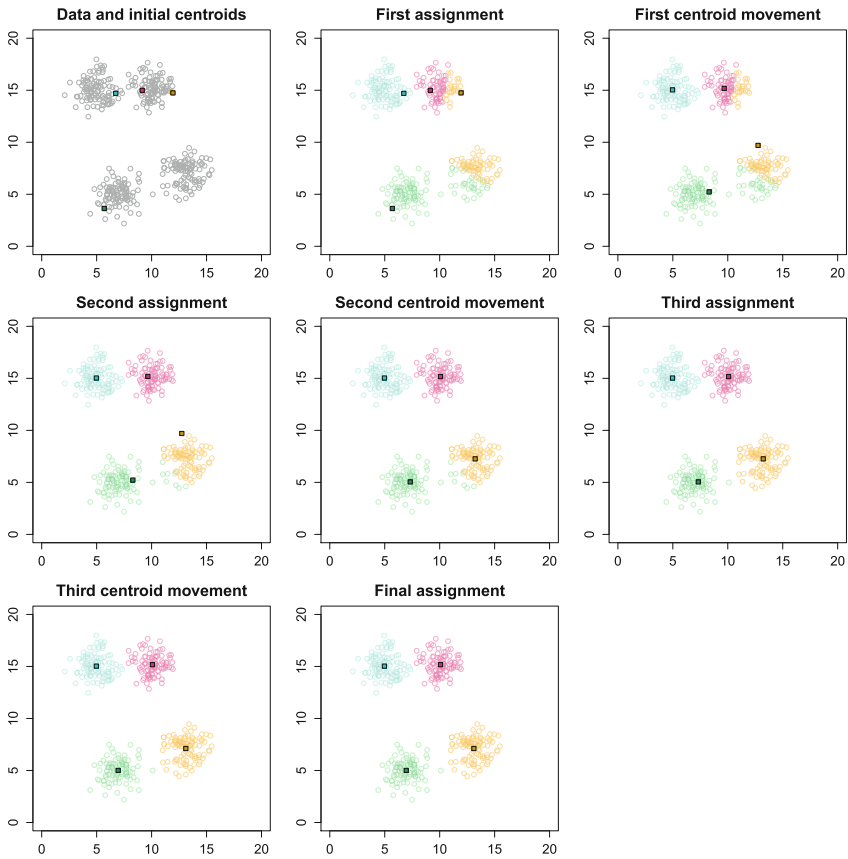We first present a visualization of the $K$-Means-Algorithm in Fig. 5.

**Fig. 5.** An illustration of the $K$-Means-Algorithm.

## A.2    Pseudocode

We now present the exact workings of the $K$-Means-Algorithm in Algorithm 3, where operations like addition and division are performed component-wise if applied to vectors.

---

**Algorithm 3.** The $K$-Means-Algorithm

---

**Input**: Data set $X = \{x_1, \ldots, x_m\}$ // $x_i \in \mathbb{R}^\ell$ for some $\ell$
**Input**: Number of clusters $K$
**Input**: Number of iterations $T$
// Initialization
1  Randomly reorder $X$;
2  Set centroids $c_k = x_k$ for $k = 1$ to $K$;
   // Keep track of centroid assignments
3  Generate $m$-dimensional vector $A$;
   // Keep track of denominators in average computation
4  Generate $K$-dimensional vector $d = (d_1, \ldots, d_K)$;
5  **for** $j = 1$ *to* $T$ **do**
      // Cluster Assignment
6     **for** $i = 1$ *to* $m$ **do**
7        $\Delta = \infty$;
8        **for** $k = 1$ *to* $K$ **do**
9           $\tilde{\Delta} := ||x_i - c_k||_2$;
            // Check if current cluster is closer than previous closest
10          **if** $\tilde{\Delta} < \Delta$ **then**
               // If so, update $\Delta$ and assign data entry to current cluster
11             $\Delta = \tilde{\Delta}$;
12             $A_i = k$;
13          **end**
14       **end**
15    **end**
      // Move Centroids
16    **for** $k = 1$ *to* $K$ **do**
17       $c_k = 0$;
18       $d_k = 0$;
19    **end**
20    **for** $i = 1$ *to* $m$ **do**
         // Add the data entry to its assigned centroid
21       $c_{A_i} = c_{A_i} + x_i$ ;
         // Increase the appropriate denominator
22       $d_{A_i} = d_{A_i} 1$
23    **end**
24    **for** $k = 1$ *to* $K$ **do**
         // Divide centroid by number of assigned data entries to get average
25       $c_k = c_k / d_k$;
26    **end**
27 **end**
**Output**: $\{c_1, \ldots, c_K\}$

---

# B    Operations for Fractional Encoding

This section presents how to build the elementary operations for Fractional Encoding from routines to perform addition, multiplication and comparison on numbers that are encoded in binary fashion. We denote these routines with $\texttt{Add}(a, b), \texttt{Mult}(a, b)$ and $\texttt{Comp}(a, b)$, where the latter returns 1 (encrypted) if $a < b$ and 0 otherwise. Then if we want to operate on values encoded in this way, we can express the operations using the subroutines from the binary encoding as follows:

- $a + b : \texttt{FracAdd}((a_n, a_d), (b_n, b_d))$
  $= \big(\texttt{Add}(\texttt{Mult}(a_n, b_d), \texttt{Mult}(a_d, b_n)), \texttt{Mult}(a_d, b_d)\big)$
- $a \cdot b : \texttt{FracMult}((a_n, a_d), (b_n, b_d)) = \big(\texttt{Mult}(a_n, b_n), \texttt{Mult}(a_d, b_d)\big)$
- $a/b : \texttt{FracDiv}((a_n, a_d), (b_n, b_d)) = \big(\texttt{Mult}(a_n, b_d), \texttt{Mult}(a_d, b_n)\big)$
- $a \leq b : \texttt{FracComp}((a_n, a_d), (b_n, b_d)) :$
  This is slightly more involved. Note that the MSB determines the sign of the number (1 if it is negative and 0 otherwise). Let

$$c := \mathrm{Sign}(a_d) \oplus \mathrm{Sign}(b_d),$$

and let

$$\texttt{MUX}(c, a, b) = \begin{cases} a, & c = 1 \\ b, & c = 0 \end{cases}$$

be the multiplexer gate.
Then we set

$$d := \texttt{MUX}(c, \texttt{Mult}(a_n, b_d), \texttt{Mult}(a_d, b_n))$$

and

$$e := \texttt{MUX}(c, \texttt{Mult}(a_d, b_n), \texttt{Mult}(a_n, b_d))$$

and output the result as $\texttt{Comp}(e, d)$.
A more detailed explanation can be found in the extended version of this paper.

## C     Details of the Approximate Algorithm

In this section, we present the details of the approximate version of our algorithm.

### C.1     The Algorithm

Recall the main idea: Since the $\texttt{Compare}$ function is linear in the length of its inputs, speeding up this building block would make the entire computation more efficient. To do this, first recall that we encode our numbers in a bitwise fashion after having scaled them to integers. This means that we have access to the individual bits and can, for example, delete the $S$ least significant bits, which corresponds to dividing the number by $2^S$ and truncating. Let $\tilde{X}$ denote this truncated version of a number $X$, and $\tilde{Y}$ that of a number $Y$. Then $\texttt{Compare}(\tilde{X}, \tilde{Y}) = \texttt{Compare}(X, Y)$ if $|X - Y| \geq 2^S$, and may or may not return the correct result if $|X - Y| < 2^S$. However, correspondingly, if the result is wrong, the centroid that is wrongly assigned to the data entry is no more than $2^S$ further from the data entry than the correct one. We propose to pick an initial $S$ and decrease it over the course of the algorithm, so that accuracy increases as we near the end. The exact workings of this approximate comparison, denoted $\texttt{ApproxCompare}$, can be seen in Algorithm 4.

---

**Algorithm 4.** ApproxCompare$(X, Y, S)$

---

**Input**: The two arguments $X, Y$, encoded bitwise
**Input**: The accuracy factor $S$
// Corresponds to $\tilde{X} = \lfloor X/2^S \rfloor$
**1** Remove last $S$ bits from $X$, denote $\tilde{X}$;
// Corresponds to $\tilde{Y} = \lfloor Y/2^S \rfloor$
**2** Remove last $S$ bits from $Y$, denote $\tilde{Y}$;
// Regular comparison function, $C \in \{0, 1\}$
**3** $C = $ Compare$(\tilde{X}, \tilde{Y})$;
**Output**: $C$

---

### C.2    Evaluation

In this section, we compare the performance of the stabilized $K$-Means-Algorithm using this approximate comparison, denoted simply by "Approximate Version", to the original and stabilized $K$-Means-Algorithm on our data sets.

**Accuracy.** Recall from Sect. 5.1 that we scaled the data with the factor $2^{20}$ and truncated to obtain the input data. This means that for $S = 5$, a wrongly assigned centroid would be at most $2^5$ further from the data entry than the correct centroid on the scaled data - or no more than $2^{-15}$ on the original data scale. We set $S = \min\{7, (T/5) - 1\}$ where $T$ is the number of iterations, and
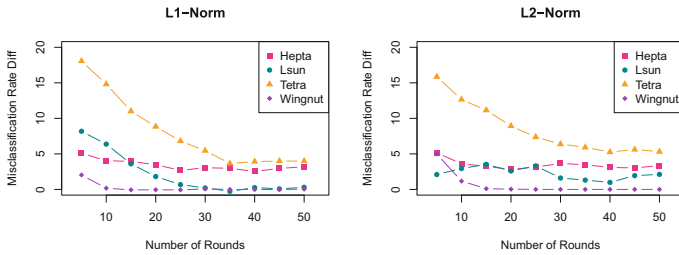


**Fig. 6.** Average difference in misclassification rate for approximate vs. stabilized algorithm ((average % mislabeled approximate) − (average % mislabeled stabilized)).
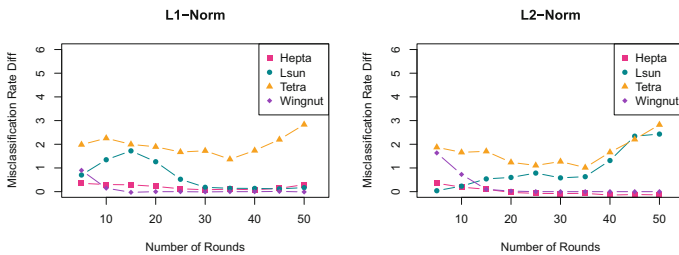


**Fig. 7.** Average difference in misclassification rate for approximate vs. stabilized algorithm ((average % mislabeled approximate) − (average % mislabeled stabilized)).

reduce $S$ by one every 5 rounds. We again examine the average (over 1000 iterations) difference in the misclassification rate to both the exact algorithm and the stabilized algorithm.

The results for both distance metrics can be seen in Figs. 6 and 7. We see that again, $T = 40$ iterations is a reasonable choice because the algorithms do not seem to converge further with more rounds. We now again look at
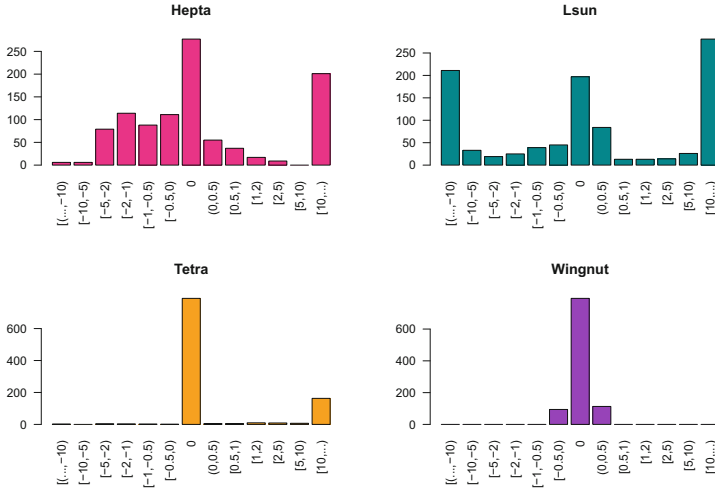


**Fig. 8.** Distribution of the difference in misclassification rate for approximate vs. exact $K$-Means-Algorithm $\big((\%\ \text{mislabeled approximate}) - (\%\ \text{mislabeled exact})\big)$, $L_1$-norm.
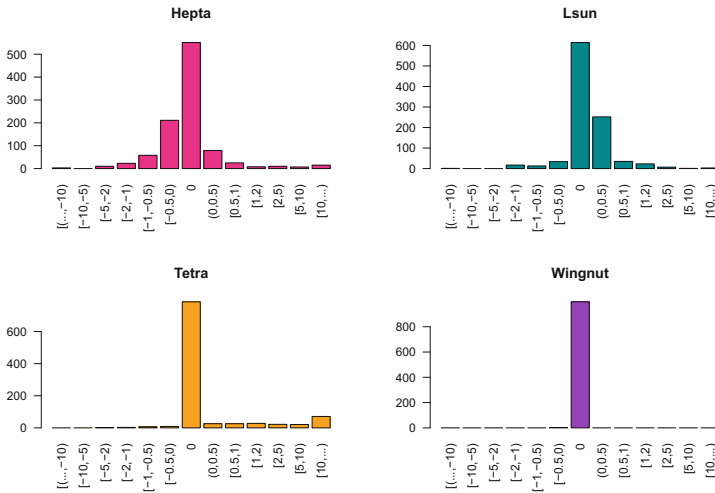


**Fig. 9.** Distribution of the difference in misclassification rate for approximate vs. stabilized $K$-Means-Algorithm $\big((\%\ \text{mislabeled approx.}) - (\%\ \text{mislabeled stab.})\big)$, $L_1$-norm.

the distribution of the ratios in Fig. 8 (for the approximate versus the exact $K$-Means-Algorithm) and Fig. 9 (for the approximate versus the stabilized $K$-Means-Algorithm). Figures for the $L_2$-norm can be found in the extended version of this paper.

We see that usually, the approximate version performs only slightly worse than the stabilized version. There is still the effect in the Lsun dataset that the approximate version outperforms the original $K$-Means-Algorithm in a significant amount of cases (though this effect mostly occurs for the $L_1$-norm), but it rarely does better than the stabilized version. This is not surprising, as it is in essence the stabilized version but with an opportunity for errors.

**Runtime.** We now examine how much gain in terms of runtime we have from this modification. Recall that it took about 1.5 years to run the exact algorithm, and 25.93 days to run the stabilized version. The approximate version runs in 25.79 days, which means a difference of about 210.7 min.

Obviously, the effect of the approximate comparison is not as big as anticipated. This is due to the bottleneck actually being the computation of the $L_1$-norm rather than the `FindMin`-procedure. Thus, for this specific application, the approximate version may not be the best choice - however, for an algorithm that has a high number of comparisons relative to other operations, there can still be huge performance gains in terms of runtime. To see this, we ran just the comparison and approximate comparison functions with the same parameters as in our implementation of the $K$-Means-Algorithm (35 bits, 5 bits deleted for approximate comparison). The average (over 1000 runs each) runtime was 3.24 s for the regular comparison and 1.51 s for the approximate comparison. We see that this does make a big difference, which is why we choose to present the modification even though the effect was outweighed by other bottlenecks in the $K$-Means-Algorithm computation.

**Conclusion.** In conclusion, the approximate comparison provides the user with an easy method of trading in accuracy for faster computation, and most importantly, this loss of accuracy can be decreased as computations near the end. However, for the specific application of the $K$-Means-Algorithm, these gains were unfortunately swallowed by the rest of the computation.

# References

1. Aggarwal, C.C., Hinneburg, A., Keim, D.A.: On the surprising behavior of distance metrics in high dimensional space. In: Van den Bussche, J., Vianu, V. (eds.) ICDT 2001. LNCS, vol. 1973, pp. 420–434. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44503-X_27
2. Armknecht, F., et al.: A guide to fully homomorphic encryption. IACR Cryptology ePrint Archive (2015/1192)
3. Armknecht, F., Katzenbeisser, S., Peter, A.: Group homomorphic encryption: characterizations, impossibility results, and applications. DCC **67**, 209–232 (2013)

4. Barnett, A., et al.: Image classification using non-linear support vector machines on encrypted data. IACR Cryptology ePrint Archive (2017/857)

5. Bonte, C., Vercauteren, F.: Privacy-preserving logistic regression training. IACR Cryptology ePrint Archive 233 (2018)

6. Bos, J.W., Lauter, K.E., Naehrig, M.: Private predictive analysis on encrypted medical data. J. Biomed. Inform. **50**, 234–243 (2014)

7. Bost, R., Popa, R.A., Tu, S., Goldwasser, S.: Machine learning classification over encrypted data. In: NDSS (2015)

8. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: Fully homomorphic encryption without bootstrapping. In: ECCC, vol. 18 (2011)

9. Bunn, P., Ostrovsky, R.: Secure two-party k-means clustering. In: CCS (2007)

10. Chabanne, H., de Wargny, A., Milgram, J., Morel, C., Prouff, E.: Privacy-preserving classification on deep neural network. IACR Cryptology ePrint Archive (2017/035)

11. Chen, H., Laine, K., Player, R.: Simple encrypted arithmetic library - SEAL v2.1. IACR Cryptology ePrint Archive 2017, 224 (2017)

12. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster fully homomorphic encryption: bootstrapping in less than 0.1 seconds. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016. LNCS, vol. 10031, pp. 3–33. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-53887-6_1

13. Coron, J.-S., Lepoint, T., Tibouchi, M.: Scale-invariant fully homomorphic encryption over the integers. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 311–328. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54631-0_18

14. Coron, J.-S., Naccache, D., Tibouchi, M.: Public key compression and modulus switching for fully homomorphic encryption over the integers. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 446–464. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29011-4_27

15. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 24–43. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13190-5_2

16. 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

17. Esperança, P.M., Aslett, L.J.M., Holmes, C.C.: Encrypted accelerated least squares regression. In: Singh, A., Zhu, X.J. (eds.) AISTATS (2017)

18. Fan, J., Vercauteren, F.: Somewhat practical fully homomorphic encryption. IACR Cryptology ePrint Archive (2012/144)

19. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009)

20. 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

21. Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K.E., Naehrig, M., Wernsing, J.: CryptoNets: applying neural networks to encrypted data with high throughput and accuracy. In: ICML (2016)

22. Graepel, T., Lauter, K., Naehrig, M.: ML confidential: machine learning on encrypted data. In: Kwon, T., Lee, M.-K., Kwon, D. (eds.) ICISC 2012. LNCS, vol. 7839, pp. 1–21. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37682-5_1

23. Halevi, S., Shoup, V.: Algorithms in HElib. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8616, pp. 554–571. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44371-2_31

24. Jagannathan, G., Pillaipakkamnatt, K., Wright, R.N., Umano, D.: Communication-efficient privacy-preserving clustering. Trans. Data Priv. **3**, 1–25 (2010)

25. Jagannathan, G., Wright, R.N.: Privacy-preserving distributed k-means clustering over arbitrarily partitioned data. In: SIGKDD (2005)

26. Jäschke, A., Armknecht, F.: Accelerating homomorphic computations on rational numbers. In: Manulis, M., Sadeghi, A.-R., Schneider, S. (eds.) ACNS 2016. LNCS, vol. 9696, pp. 405–423. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39555-5_22

27. Jäschke, A., Armknecht, F.: (Finite) field work: choosing the best encoding of numbers for FHE computation. In: Capkun, S., Chow, S. (eds.) Cryptology and Network Security. CANS 2017, vol. 11261, pp. 482–492. Springer, Cham (2017). https://doi.org/10.1007/978-3-030-02641-7_23

28. Jha, S., Kruger, L., McDaniel, P.: Privacy preserving clustering. In: di Vimercati, S.C., Syverson, P., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 397–417. Springer, Heidelberg (2005). https://doi.org/10.1007/11555827_23

29. Kim, A., Song, Y., Kim, M., Lee, K., Cheon, J.H.: Logistic regression model training based on the approximate homomorphic encryption. IACR Cryptology ePrint Archive (254) (2018)

30. Kim, M., Song, Y., Wang, S., Xia, Y., Jiang, X.: Secure logistic regression based on homomorphic encryption. IACR Cryptology ePrint Archive (074) (2018)

31. Liu, X., et al.: Outsourcing two-party privacy preserving k-means clustering protocol in wireless sensor networks. In: MSN (2015)

32. Lu, W., Kawasaki, S., Sakuma, J.: Using fully homomorphic encryption for statistical analysis of categorical, ordinal and numerical data. IACR Cryptology ePrint Archive (2016/1163)

33. MacQueen, J., et al.: Some methods for classification and analysis of multivariate observations. In: Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability (1967)

34. Meskine, F., Bahloul, S.N.: Privacy preserving k-means clustering: a survey research. Int. Arab J. Inf. Technol. **9**, 194–200 (2012)

35. Naehrig, M., Lauter, K.E., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: CCSW (2011)

36. Phong, L.T., Aono, Y., Hayashi, T., Wang, L., Moriai, S.: Privacy-preserving deep learning via additively homomorphic encryption. IACR Cryptology ePrint Archive (2017/715)

37. Smart, N.P., Vercauteren, F.: Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Nguyen, P.Q., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 420–443. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13013-7_25

38. TFHE Library. https://tfhe.github.io/tfhe

39. Ultsch, A.: Clustering with SOM: U* c. In: Proceedings of Workshop on Self-Organizing Maps (2005)

40. Vaidya, J., Clifton, C.: Privacy-preserving k-means clustering over vertically partitioned data. In: SIGKDD (2003)
41. Wu, D.J., Feng, T., Naehrig, M., Lauter, K.E.: Privately evaluating decision trees and random forests. PoPETs, (4) (2016)
42. Xing, K., Hu, C., Yu, J., Cheng, X., Zhang, F.: Mutual privacy preserving $k$-means clustering in social participatory sensing. IEEE Trans. Ind. Inform. **13**, 2066–2076 (2017)