UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

# Practicality of Differentially Private Clustering

*Praktikabilität von differenziellen privaten Clusteralgorithmen*

**Bachelorarbeit**

im Rahmen des Studiengangs
**IT-Sicherheit**
der Universität zu Lübeck

vorgelegt von
**Andre Edler**

ausgegeben und betreut von
**Prof. Dr. Esfandiar Mohammadi und M. Sc. Marven Kummerfeldt**

Lübeck, den 12. Juni 2023

# Abstract

Clustering is a machine learning discipline that analysis data sets to find centers that represent a set of similar data points. During the calculation of these cluster centers, the cluster algorithm must use the data to find similarities and patterns. When personal data is used, it poses a potential risk to the privacy of those who participate in the data set. Attackers might be able to identify individual data points and thus also personal data of individuals.
Differential privacy can be applied to clustering algorithms to protect the privacy of participants in a data set.
We implement such a differentially private clustering algorithm proposed by Balcan et al. [Hon17a] in Python. This algorithm works in theory, but in practice some adjustments need to be made to achieve a reasonable runtime and produce proper results. Mistakes also happen quickly during implementation, which is why we also need to correct some mistakes in the Matlab implementation of Balcan et al. [Hon17b]. We compare our Python implementation with the corrected Matlab implementation and show that we obtain similar results.
Various adjustments can be made to the algorithm that affect the results differently. We show that some modifications to the algorithm improve the clustering results and that many other modifications worsen the clustering results and cause more runtime. At last, we compare our improved algorithm with others and discover that it can compete with other differentially private algorithms but is still worse than a non-private algorithm.

# Zusammenfassung

Clustering ist eine Disziplin des maschinellen Lernens, die Datensätze analysiert, um Zentren zu finden, die eine Gruppe ähnlicher Datenpunkte zusammenfassen. Bei der Berechnung dieser Clusterzentren muss der Clusteralgorithmus die Daten nutzen, um Ähnlichkeiten und Muster zu finden. Wenn personenbezogene Daten verwendet werden, stellt dies ein potenzielles Risiko für die Privatsphäre derjenigen dar, die in dem Datensatz enthalten sind. Angreifer könnten in der Lage sein, einzelne Datenpunkte und damit auch persönliche Daten von Personen zu identifizieren.
Differentielle Privatsphäre kann auf Clustering-Algorithmen angewandt werden, um die Privatsphäre der Teilnehmer in einem Datensatz zu schützen.
Wir implementieren einen solchen differenziellen privaten Clustering-Algorithmus, der von Balcan et al. [Hon17a] vorgeschlagen wurde, in Python. Dieser Algorithmus funktioniert lediglich in der Theorie, in der Praxis hingegen müssen einige Anpassungen vorgenommen werden, um eine vernünftige Laufzeit zu erreichen und nützliche Ergebnisse zu erzielen. Auch während der Implementierung passieren schnell Fehler, weshalb wir auch einige Fehler in der Matlab-Implementierung von Balcan et al. [Hon17b] korrigieren müssen. Wir vergleichen unsere Python-Implementierung mit der korrigierten Matlab-Implementierung und zeigen, dass wir sehr ähnliche Ergebnisse erzielen.
An dem Algorithmus können noch verschiedenste Anpassungen vorgenommen werden, die sich alle unterschiedlich auf die Ergebnisse auswirken. Wir zeigen, dass einige Modifikationen am Algorithmus die Clustering Ergebnisse verbessern und das viele andere Modifikationen die Clustering Ergebnisse verschlechtern und mehr Laufzeit verursachen.
Zum Schluss vergleichen wir unseren verbesserten Algorithmus mit anderen Clustering-Algorithmen und stellen fest, dass unser verbesserter Algorithmus im Vergleich zu anderen differenziellen privaten Algorithmen ähnliche Ergebnisse erzielt, aber immer noch schlechter ist als ein nicht-privater Clustering Algorithmus.

## Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 12. Juni 2023

# Contents

# 1 Introduction

Machine learning enables systems to learn patterns in training data sets and then classify new data or create general rules from the training data. One discipline of unsupervised machine learning is clustering, which attempts to group similar data points into clusters and computes a center for each cluster. One way to formalize the problem of clustering is to find $k$ cluster centers in a data set $D$ of $n$ data points and minimize the sum of the distances of each data point to its closest cluster center. This problem is already NP-hard and we also focus on big data sets in high-dimensional Euclidean spaces, which makes it even more complicated. There is already a popular clustering algorithm called the k-means algorithm or Lloyd's algorithm that provides useful results for many scenarios and data sets.

However, when processing data sets containing sensitive and personal data during clustering, the result could reveal sensitive information about individual data points. Thus, the privacy of participants in a data set used for clustering algorithms could be violated.

To protect the privacy of participants in the data set, differential privacy [Dwo06] could be applied. Differential privacy offers a strong privacy guarantee for statistical databases and can also be applied in other applications such as clustering algorithms.

The goal is to compute cluster centers in sensitive data sets while maintaining the privacy of the participants in the data sets. There are already some solutions for this [Nie16, Kap09] but they do not work well in the context of big data in high-dimensional scenarios [Hon17a]. Therefore, Balcan et al. [Hon17a] proposed an algorithm that provides a solution for differentially private clustering in high-dimensional Euclidean spaces. To solve the high-dimensional problem, the algorithm first reduces the dimension of all data points in the data set and continues working with the low-dimensional data set. Then, low-dimensional spaces containing many data points are recursively partitioned and the centers of these spaces are stored. Afterward, a technique of local swapping is used to greedily replace bad centers with good centers and extract just $k$ centers. At last, a recovery from the low-dimensional centers to the high-dimensional centers is performed.

The approach of Balcan et al. is promising but, like many other theoretical algorithms on this topic, has many problems in practice such as too long runtime, too much memory usage, and too poor results. Furthermore, the existing Matlab implementation [Hon17b] of the Balcan et al. algorithm contains many bugs that, for example, violate the privacy guarantees. This work discusses the problems of the algorithm of Balcan et al., fixes the bugs

in their Matlab implementation, and improve the problems. Therefore, we implemented the algorithm in Python [Edl23] and had to make a few modifications to achieve appropriate runtime and appropriate results. But the results were still relatively poor. Thus, we further studied the effects of various adjustments to the algorithm on the quality of the results and the runtime.

## 1.1 Contribution

Our contribution can be summarized as follows:

- Implementation of a differentially private clustering algorithm, based on Balcan et al. [Hon17a], in Python.
- Correction of bugs in the existing Matlab implementation [Hon17b] of Balcan et al.
- Examination of the effects of various adjustments to the algorithm on the quality of the results and the runtime.
- Test of the algorithm with the MNIST data set [Haf98] and synthetic data sets and comparison with other clustering algorithms.

## 1.2 Related Work

Clustering is a classic problem in the field of machine learning and there are already some algorithms that provide good results [Vas07]. The most common clustering algorithm is the standard k-means clustering algorithm, also called Lloyd's algorithm, which is popular because it is simple and fast. However, clustering in high-dimensional Euclidean spaces in a differentially private manner is another matter.
There also exist some algorithms but many of them are either not efficient or work only if the data set fulfills certain assumptions. One of the first attempts of differentially private clustering was proposed by Blum et al. [Kob21], who introduced the SulQ Framework. This framework satisfies differential privacy and can be applied to the Lloyd's algorithm. The problem is that this algorithm offers no guarantees for the quality of the results. Other algorithms have the problem of making strong assumptions about the input data [Smi07, Sin15]. Nock et al. [Nie16] proposed a private version of the k-means++ algorithm (k-variates++) but the algorithm can only run when the number of clusters searched for ($k$) is small. The time and space complexity of the griding algorithm of Su et al. [Jin10] depends on the dimension of the data set. Gupta et al. [Tal16] proposed an efficient algorithm but it does not work in Euclidean spaces. The algorithm of Feldman et al. [Kap09] makes no

assumption about the input data set but there is no computationally efficient algorithm for the high-dimensional space. Our algorithm is based on the algorithm of Balcan et al. [Hon17a], which provides good guarantees for the quality of the results and works on big high-dimensional data sets. Chaturvedi et al. [Eri21] also proposed a differentially private k-means algorithm that outperforms the algorithm of Balcan et al. [Hon17b].

## 2 Problem Statement

Clustering sensitive data carries the risk that sensitive information of participants in the data set could be revealed. For example, when clustering medical data, a person's health status could be exposed. An attacker accomplishes this through linkage attacks, background knowledge, and other attacks. Then he could use this sensitive information to extort a person, for instance. Therefore, the output of such an algorithm should not reveal any individual data. The task is to find clusters while preserving the privacy of the participants in the data set.

To protect against all attacks, regardless of what knowledge an attacker has, differential privacy can be applied. Differential privacy is a privacy guarantee for statistical databases. The basic idea is that every time data is processed, some sort of mechanism must be applied that randomizes the outcome. This could be done, for example, by adding noise or by randomly selecting from a set of candidates. So much noise must be added by the mechanism that an attacker cannot comprehend whether the outcome was affected by a particular data point or by the added noise. These mechanisms can also be applied to clustering algorithms.

Nevertheless, the differentially private implementation of a clustering algorithm is only one goal. Another goal is to achieve useful results with appropriate runtime and memory usage while preserving privacy. Even if no privacy-preserving mechanism is applied, clustering is already an NP-hard problem, and everything gets even more complicated when the input data is big and high-dimensional. We still want to learn patterns and similarities in the data set and then be able to classify new data correctly. So, we also want to get the best clustering result possible. One way to formalize the clustering problem is to find a set of $k$ cluster centers $z_1, ..., z_k \in \mathbb{R}^d$ in a data set $D = \{x_1, ..., x_n\} \in \mathbb{R}^d$ of $n$ data points and minimize the sum of the distances of each data point to its closest cluster center, called clustering loss. To measure the quality of the results, the inertia can be used, which is also called k-means objective and is defined as:

$$\sum_{i=1}^{n} \min_{j} ||x_i - z_j||^2.$$

There are already many theoretical approaches to differentially private clustering algorithms that show promising performance, but have many problems in practice. In prac-

tice, the theoretical algorithms are so elaborate to compute that they do not produce a result in a reasonable runtime under realistic conditions. Furthermore, these algorithms have theoretically good performance guarantees, but do not achieve sufficient performance under realistic conditions. Bugs in the implementation of these algorithms lead to faulty results and violate theoretical privacy guarantees, suggesting wrong safety.
Therefore, in this work, we will practically evaluate these problems using the differentially private clustering algorithm of Balcan et al. [Hon17a] as an example. This algorithm provides good performance guarantees and Balcan et al. already implemented the algorithm in Matlab [Hon17b]. More specifically, we evaluate the following questions:

- Does the algorithm run in a reasonable runtime and have a reasonable memory usage?
- Does the algorithm perform well compared to other algorithms?
- Are there bugs in their Matlab implementation that, for example, violate the privacy guarantees?
- If there are bugs, how can they be fixed?
- How can we improve the performance of the algorithm while preserving differential privacy and maintain runtime and memory usage reasonable?
- How well does the improved algorithm perform compared to other algorithms?

# 3 Preliminaries

In this chapter, we introduce the most important topics for this work. First, we give a brief introduction to high-dimensional data. Then, we give a general overview of clustering and present an important algorithm from this area. At last, a definition of differential privacy is given, and we look into relevant theorems.

## 3.1 High-Dimensional Data

High-dimensional data refers to data where the number of dimensions exceeds or nearly exceeds the number of data points. If the number of dimensions even exceeds the number of data points, we will never be able to describe the relationships between the data points because there are not enough of them. These data sets are common in the field of health care, for example. However, in our work, we will also refer to high-dimensional data when the number of dimensions is simply high. Working and computing with high-dimensional data is difficult and algorithms that use it often require a lot of runtime and memory.
One way to solve this problem is to reduce the dimensions using the Johnson-Lindenstrauss transformation. In summary, the Johnson-Lindenstrauss-Lemma states that a set of $n$ points in a Euclidean space of dimension $d$ can be reduced to a Euclidean space of smaller dimension $p$, while the pairwise distances between the points changing by only a small factor. Therefore, with the Johnson-Lindenstrauss transformation, the distances between the points are essentially preserved. This can be done by multiplying the $(d, n)$ data matrix $X$ by a $(p, d)$ random projection matrix $G$ that draws each element in $G$ from the Gaussian distribution $N(0, 1)$. For the Johnson-Lindenstrauss-Lemma to still hold, $p$ is bounded by a value $0 < \alpha < 1$ and the number of points $n$. The dimension $d$ can be reduced to $p = \alpha^{-2} \cdot \log n$, while the distance change between any data points is at most $1 \pm \alpha$. Thus, when $\alpha \approx 1$, the minimum dimension $p \approx \log n$ and the maximum distance change of approximately $1 \pm 1$ is reached.

## 3.2 Clustering

Machine learning enables systems to learn patterns and regularities in training data and then classify new data or create general rules from the training data. One part of machine

learning is unsupervised learning, which learns with unlabeled data. So, in the training data set, one does not need to know how a certain data point is classified, it is assigned to a certain classifier during computation. One of the most common unsupervised learning techniques is clustering. In general, the task of clustering is to group sets of similar objects into clusters. We only focus on the centroid model in the Euclidean space, where each cluster is represented by a single mean point.
We consider the following problem. The training data set consists of $n$ data points with dimension $d$ in the Euclidean space. Clustering algorithms try to find points, also called cluster centers, in the space of the data set where many data points are located. These cluster centers would then be a representation of all other data points located in that area. After executing a clustering algorithm and finding some cluster centers as a representation of the clusters, new data points can be automatically compared to the cluster centers. Based on the shortest Euclidean distance between the new data point and all cluster centers, the new data point can be classified as the same as the cluster center with the shortest distance. The goal of the clustering algorithms is to reduce the overall summarized Euclidean distances between the data points in the training data and their closest cluster centers.
One way to formalize this problem is to find a set of $k$ cluster centers $z_1, ..., z_k \in \mathbb{R}^d$ in a data set $D = \{x_1, ..., x_n\} \in \mathbb{R}^d$ of $n$ data points and minimize the sum of the distances of each data point to its closest cluster center, called clustering loss. To measure the quality of the results, the inertia can be used, which is also called k-means objective and is defined as:

$$\sum_{i=1}^{n} \min_{j} ||x_i - z_j||^2$$

### Lloyd's algorithm

The most common clustering algorithm is the standard k-means algorithm, also called Lloyd's algorithm, because it is quite simple and fast. First, this algorithm initializes a set of cluster centers by randomly selecting data points in the data set. Then, it calculates all Euclidean distances between all data points and all cluster centers and assigns each data point to the closest cluster center. After that, for each cluster center, the assigned data points are summed and then divided by the number of data points assigned to that cluster to calculate the average values and thereby receive the new cluster centers. That is repeated in several iterations until a certain threshold is reached. The goal is to compute a better set of clustering centers at each iteration that minimizes the inertia. In this way, a local optimum can be found, although it is not guaranteed that this is the best possible clustering.

The Lloyd algorithm can be easily attached to other algorithms that already calculated a set of cluster centers and further improve this set of cluster centers.
Since the results of the Lloyd's algorithm strongly depend on the random initialization of the cluster centers at the beginning, the k-means++ algorithm was proposed to improve this problem. The k-means++ algorithm specifies a procedure for initializing cluster centers and then proceeds with the standard k-means algorithm. This procedure first randomly selects a data point as a cluster center. After that, all subsequent cluster centers are selected from the remaining data points with a probability proportional to the squared distances from the data points to the closest cluster center. As a result, the initialized cluster centers are already well distributed across the data points, making it easier to process them into better cluster centers. With the initialization of k-means++, the k-means algorithm has better performance guarantees.

## 3.3 Differential Privacy

Differential privacy can be described as a promise to anyone who participates in a data set used for a study or analysis that no sensitive information of an individual can be leaked. Therefore, the presence of any data point in the data set must be hidden.
Formally, we consider the following scenario that can be seen as a cryptographic game. An adversary has access to two neighbouring data sets $D$ and $D'$. Neighbouring data sets means that they only differ in one element $x$: $D' = D \cup \{x\}$. Then the adversary queries both data sets to a challenger, who inputs the data sets in some mechanism $M$ and returns the output to the adversary. A mechanism $M$ is formally a function that maps a data set $D$ to a random variable. Based on the output of the mechanism, the adversary should not be able to reliably tell which of the outputs belongs to which of the inputs.
Two parameters $\epsilon$ and $\delta$ need to be defined for the privacy guarantees of the mechanism $M$. $\epsilon$ defines the maximum difference between the output distributions of $M$ on $D$ and $D'$. $\delta$ describes the probability of events that are not covered by $\epsilon$.

**Definition 3.1 (Differential Privacy).** A randomized mechanism $M : \mathcal{D} \to \mathcal{R}$ preserves $(\epsilon, \delta)$-differentially privacy if for all neighbouring data sets $D_1, D_2 \in \mathcal{D}$ and for all $S \subseteq \mathcal{R}$:

$$Pr[M(D_1) \in S] \leq exp(\epsilon) \cdot Pr[M(D_2) \in S] + \delta$$

If $\delta = 0$, $M$ is defined as $\epsilon$-differentially private.

### Sensitivity

Before we define how differentially private mechanisms work, we need to define the sensitivity of data sets. Sensitivity is the value that indicates how much a single data point

can affect the output of a mechanism in the worst case. Therefore, it defines how much uncertainty must be added to the response of a mechanism to hide the participation of a single data point. It is used to specify the minimum amount of noise that a mechanism has to add to the output to perturb its output and thus preserve privacy. The idea behind this is that when using a mechanism, it is not possible to reliably tell from the output whether a certain data point is included in the data set, as the effect of a data point could be completely added by noise [Aar14].

**Definition 3.2 (Sensitivity).** The sensitivity $\Delta$ of a function $f : A \to B$ is:

$$\Delta f = \max_{x,y \in A; ||x-y||_1=1} ||f(x) - f(y)||_1$$

### Laplace Mechanism

The Laplace mechanism is an additive noise mechanism and uses the previously described sensitivity of a data set to achieve differential privacy. It can be used whenever an algorithm calculates a numerical value based on a data set to perturb the actual value with a random variable. As the name already implies, the amount of noise to be added is drawn from the Laplace distribution. The random variable drawn from the Laplace distribution $Lap(x, b)$ has the following probability density function:

$$f(x) = \frac{1}{2b} \cdot \exp(-\frac{|x|}{b}).$$

We simply write $Lap(b)$ to denote the Laplace distribution with scale $b$. So, the mechanism calculates the actual value of a function $f$ and then adds Laplace noise $Lap(b)$. The scale $b$ depends on the sensitivity $\Delta$ of the function $f$ and the privacy parameter $\epsilon$.

**Definition 3.3 (Laplace Mechanism).** Given a function $f$ and the privacy parameter $\epsilon$, the Laplace mechanism $M_{Laplace}(x, f(\cdot), \epsilon)$ is defined as:

$$M_{Laplace}(x, f(\cdot), \epsilon) = f(x) + Lap(\frac{\Delta f}{\epsilon})$$

In summary, the Laplace mechanism adds noise to the output of a function, making it impossible to reliably tell whether a data point was included in the function's input data set [Aar14].

### Exponential Mechanism

To privately select an element $r$ from a candidate set $R$, the exponential mechanism can be used [Aar14]. Therefore, we can accurately return a given value while preserving differential privacy. To do this, we need to define a scoring functions $u$ that outputs a score for

each element in the candidate set, indicating how likely it is that that exact element will be selected. So basically, privacy is preserved by sometimes returning elements that do not have the best score and therefore are not the best option.

**Definition 3.4 (Exponential Mechanism).** Given a data set $D$, a scoring function $u$, a candidate set $R$ and the privacy parameter $\epsilon$, the exponential mechanism $M_E(D, u, R)$ selects and outputs an element $r \in R$ with probability proportional to $\exp(\frac{\epsilon \cdot u(D,r)}{2 \cdot \Delta u})$.

### Other Theorems

- **Definition 3.5 (Sequential Composition).** Let $M_1, ..., M_n$ be differentially private mechanisms whose privacy guarantees are $\epsilon_1, ..., \epsilon_n$. When executed sequentially, the overall mechanism is $(\sum_{i=1}^{n} \epsilon_i)$-differentially private.
- **Definition 3.6 (Parallel Composition).** Let $M_1, ..., M_n$ be differentially private mechanisms whose privacy guarantees are $\epsilon_1, ..., \epsilon_n$. If the mechanisms are computed sequentially but on disjoint data sets, then the overall mechanism would be $(\max_i \epsilon_i)$-differentially private.
- *Differential privacy is immune against post-processing.*
  It is impossible to create a function that uses the output of a differentially private mechanism and make it less differentially private.

# 4 Approach

This chapter focuses on the differentially private clustering algorithm proposed by Balcan et al. [Hon17a]. In the first section, we show how the algorithm works and how we implemented it in Python [Edl23]. The second section then examines some adjustments that had to be made to the algorithm to achieve acceptable runtime, memory usage and results. The third section looks at the mistakes in the Matlab implementation of Balcan et al. [Hon17b] and how we corrected them. We then compare the two implementations in the last section of this chapter to evaluate the performance of both implementations.

## 4.1 Differentially Private Clustering Algorithm for High-Dimensional Data

### 4.1.1 Basic Concepts

Before we explain the basic idea of the algorithm, we introduce some basic concepts, that are introduced and used by Balcan et al.

#### Clustering Loss

There are different loss functions when applying clustering, and Balcan et al. define the clustering loss as follows. When computing a set of cluster centers $z_1, ..., z_k \in \mathbb{R}^d$ to a data set containing data points $x_1, ..., x_n \in \mathbb{R}^d$, the clustering loss is defined as the sum of the distances of each data point to its closest cluster center:

$$\sum_{i=1}^{n} \min_{j} ||x_i - z_j||$$

#### Cubes

Balcan et al. define cubes to subdivide the Euclidean space into sub-spaces. These cubes can be partitioned. By a partition of a cube we mean that it is partitioned into equally sized sub-cubes that completely cover the space of the original cube, and that the side lengths of these sub-cubes are as long as half the side lengths of the original cube. Therefore, in each partition of a cube, a grid of equally sized sub-cubes is created that covers the space of the cube. These sub-cubes can be further partitioned into smaller cubes that are an increasingly fine-grained partitioning of the space.

#### Swap

A swap is in the work of Balcan et al. defined as a swap between an element in a currently used set and an element of another set that does contain all elements that are currently not used. All possible swaps contain all possible combinations of switching one element of the currently used set with one of the other set. Thus, the size of both sets remains the same before and after a swap, only one element changes in each set.

### 4.1.2 Basic Idea

The basic idea of the algorithm can be summarized as follows.

First, the data is projected to a lower dimension to minimize the problems that high-dimensional data brings.

Now a cube is created that covers the Euclidean space of the low-dimensional data. If the number of data points contained in the cube exceeds a pre-defined threshold, the center of that cube is added to the so-called candidate set and the cube is partitioned along all dimensions into equally sized sub-cubes. This is repeated with the sub-cubes until no more cubes are partitioned or the maximum number of iterations is reached. In this way, the algorithm iteratively calculates centers of cubes that better and better represent the spaces where data points are located and therefore where potential clusters are located.

Then, the algorithm selects a set of centers from the candidate set as cluster centers and calculates the clustering loss for each possible swap with the other points in the candidate set. The probabilities for each swap are calculated based on these clustering losses and swaps are selected based on these probabilities. Thereby, swaps that reduce the clustering loss are more likely to be selected. After swapping several cluster centers, a subset of swaps is selected based on the clustering losses resulting from these swaps, thus obtaining the low-dimensional cluster centers.

After that, a recovery of the low-dimensional cluster centers into the high-dimensional cluster centers is performed and $3$ iterations of the Llyod's algorithm are run with these high-dimensional centers.

All of this is done several times to compute different sets of cluster centers, and one of these sets is selected based on its clustering loss.

### 4.1.3 Implementation

For the implementation [Edl23] we used Python 3.11 along with the NumPy library [HMvdW+20]. The algorithm is split into $6$ sub-algorithms for clarity.

### Basic Construct

Algorithm 1 is the basic construct of the algorithm and brings together all other parts of the algorithm. It calls the other algorithms and provides a general overview.
To use the algorithm, the data input must be a matrix containing one data point in each column. Therefore, the number of rows of the matrix is equal to the number of dimensions $d$ and the number of columns is equal to the number of data points $n$. Furthermore, the number of clusters to be found in the data $k$, the privacy budget $\epsilon$, the failure probability $\delta$ and the range of the data $r$ must be entered. The range of the data is defined as the $l\_2$ radius of a single dimension, that is, the Euclidean distance between the minimum possible point in the data set and the maximal possible point in the data set in a single dimension.
A private set of cluster centers $Z$ is computed in each of the $T$ iterations. First, the data is transformed into a lower dimension using the Johnson-Lindenstrauss transformation. Finding clusters in this low-dimensional data is likely to lead to clusters in the high-dimensional data as well, since the distances between data points remain nearly the same. Each data point in the high-dimensional space is related to a data point in the low-dimensional space with the same index. Then, the `candidate` algorithm 2 privately computes a set of potential cluster centers in the low-dimensional space. After that, the `localswap` algorithm 4 privately selects $k$ cluster centers in the low-dimensional space. Afterwards, the `recover` algorithm 5 privately extracts the high-dimensional cluster centers based on the low-dimensional cluster centers. Thereafter, three iterations of the `private k-means` algorithm 6 are computed, which attempts to improve the high-dimensional cluster centers. Last, a set of cluster centers $Z$ is selected using the exponential mechanism.
Here we have already implemented some changes compared to the algorithm proposed in the paper of Balcan et al. [Hon17a]. These changes are mainly based on the Matlab implementation [Hon17b]. Three iterations of the `private k-means` algorithm after the `recover` algorithm are added. Since the `private k-means` algorithm also requires a part of the privacy budget, the distribution of the privacy budget had to be adjusted. Moreover, the paper does not explain how to retrieve the range of the data $r$. For this reason, the range of the data must be specified as a parameter to the algorithm.

**Algorithm 1:** Private Clustering

**input** $(d, n)$ data matrix: $X$, number of clusters: $k$, privacy parameter: $\epsilon$, failure probability: $\delta$, range of data: $r$
**output** Cluster centers: $Z$
$low\_dim = 8 \cdot \log n$
$T = 2 \cdot \log \frac{1}{\delta}$
$Cluster\_Centers\_Sets = []$
**for** *t in range(T)* **do**
    $data\_low\_dim$ = JL-Transform($X$, $low\_dim$) $\cdot \frac{1}{\sqrt{d}}$
    $candidates$ = candidate($data\_low\_dim$, $k$, $\frac{\epsilon}{T} \cdot \frac{8}{18}$, $\delta$, $r$)
    $centers\_low\_dim$ = localswap($candidates$, $data\_low\_dim$, $k$, $\frac{\epsilon}{T} \cdot \frac{1}{18}$, $\delta$, $r \cdot \sqrt{d}$)
    $centers\_high\_dim$ = recover($centers\_low\_dim$, $data\_low\_dim$, $X$, $k$, $\frac{\epsilon}{T} \cdot \frac{1}{9}$, $r$)
    $iter = 3$
    **for** *i in range(iter)* **do**
        $centers\_high\_dim$ = priv_kmean($centers\_high\_dim$, $X$, $k$, $\frac{\epsilon}{iter \cdot T} \cdot \frac{1}{3}$, $r$)
    $Cluster\_Centers\_Sets$.append(($centers\_high\_dim$))
Choose a set of cluster centers $Z$ from $Cluster\_Centers\_Sets$ with probability in proportion to $\exp(-\frac{\epsilon \cdot clustering\_loss(Cluster\_Centers\_Sets(t))}{72 \cdot T \cdot (r \cdot \sqrt{d})^2})$
**return** $Z$

### Computation of a Set of Potential Cluster Centers

The `candidate` algorithm 2 can be seen as a structure that calls the `private_partition` algorithm 3. It saves the different candidates computed by the `private_partition` algorithm. Besides that, the initial cube for the `private_partition` algorithm is created and a random shift vector is added to the cube boundaries to receive different starting cubes and therefore better coverage of the space of the input data set. This potentially results in more and different candidates. However, this function returns a set of possible cluster centers in the low-dimensional space.

**Algorithm 2:** candidate

**input** $(p, n)$ low-dimensional data matrix: $Y$, number of clusters: $k$, privacy parameter: $\epsilon$, failure probability: $\delta$, range of data: $r$
**output** Candidates: $Cand$
$Cand$ = []
$T = 27 \cdot k \cdot \log \frac{n}{\delta}$
**for** $t$ *in range(T)* **do**
    $cube\_initial$ = Cube()
    **for** $i$ *in range(p)* **do**
        $random$ = random_uniform($-r, r$)
        $random\_bounds$ = [$-r$ + $random$, $r$ + $random$]
        $cube\_initial$.add_dimension($random\_bounds$)
    $C$ = private_partition($Y$, $\frac{\epsilon}{T}$, $\frac{\delta}{T}$, $cube\_initial$)
    $Cand$.append($C$)
**return** $Cand$

The `private_partition` algorithm 3 computes the candidates for the `candidate` algorithm 2. These candidates are centers of cubes in the low-dimensional space $\mathbb{R}^p$. The idea is that the initial cube created in the `candidate` algorithm contains the most of the data points and thus all clusters in the data set can be found by partitioning the initial cube into smaller cubes. This initial cube is added to the active cubes and the first depth of the while loop begins. A cube is called active for a while when the center of that cube will be added to the candidates $C$ and will be further partitioned, which has not yet happened. Then, the centers of the active cubes are added to the candidates $C$. After that, the active cubes are partitioned evenly in each dimension, creating for each active cube $2^p$ new Cubes. All currently active cubes are removed from the active cubes and the newly created cubes are added to the active cubes with a certain probability. If there are more data points in a cube than a threshold $\gamma$, that cube is more likely to be added to the active cubes. This is repeated at each depth until no new partitioned cube is added to the active cubes or the depth is greater than $\log n$. Without these termination conditions, the algorithm would not be efficient. In that way, it is likely to find clusters, because the space of the cube containing many data points becomes more accurate at each depth.

**Algorithm 3:** private_partition

**input** $(p, n)$ low-dimensional data matrix: $Y$, privacy parameter: $\epsilon$, failure probability: $\delta$, $cube\_initial$
**output** Cube Centers: $C$
$depth = 0$
$\epsilon' = \frac{\epsilon}{2 \cdot \log n}$
$\gamma = \frac{20}{\epsilon'} \cdot \log \frac{n}{\delta}$
$C = []$
$active\_cubes = [cube\_initial]$
**while** $depth \leq \log n$ *and* $active\_cubes \neq \emptyset$ **do**
    $depth = depth + 1$
    $Cubes\_Next\_Depth = []$
    **for** $cube$ *in* $active\_cubes$ **do**
        $C$.append($cube.center$)
        $active\_cubes$.remove($cube$)
        $new\_Cubes = cube$.partition()
        **for** $q$ *in* $new\_Cubes$ **do**
            $num\_data\_points = q$.getNumberOfDatapointsInCube()
            **if** $num\_data\_points \leq \gamma$ **then**
                $prob = \frac{1}{2} \cdot \exp(-\epsilon' \cdot (\gamma - num\_data\_points))$
            **else**
                $prob = 1 - \frac{1}{2} \cdot \exp(\epsilon' \cdot (\gamma - num\_data\_points))$
            Append $q$ with probability $prob$ to $Cubes\_Next\_Depth$.
        $active\_cubes = Cubes\_Next\_Depth$
**return** $C$

### Selection of $k$ Cluster Centers

The `candidate` algorithm 2 returns a set of potential cluster centers, which are probably more than $k$. Therefore, the `localswap` algorithm 4 reduces this set of cluster centers to only $k$ cluster clusters and attempts to swap centers that do not reduce the clustering loss for centers that do. Accordingly, $k$ centers are first randomly selected from the candidates. Then, each iteration $t$ uses the exponential mechanism to privately select a swap that reduces the clustering loss. To do this, a probability is computed for each possible swap between a candidate not in the current cluster centers and a candidate in the current cluster centers, based on the loss after and before the swap. Because of these probabilities,

it is more likely to do a swap that reduces the clustering loss. The cluster centers after a swap are saved in a list and used as current cluster centers in the next iteration. Thus, in each iteration $t$, a set of cluster centers is computed and stored. Again, the exponential mechanism is used to select one of these cluster center sets. The probabilities here are based on the clustering loss, making it more likely to select a set of cluster centers with low clustering loss. In summary, this algorithm attempts to select $k$ cluster centers from the candidate set that have a low clustering loss for the low-dimensional data.

**Algorithm 4:** localswap

**input** $candidates$, $(p, n)$ low dimensional data matrix: $Y$, number of clusters: $k$, privacy parameter: $\epsilon$, failure probability: $\delta$, range of data: $r$
**output** Cluster Centers (low dimensional)
$Centers\_Init$ = uniformly choose $k$ centers from $candidates$
$list\_Centers$ = [($Centers\_Init$)]
$T = 100 \cdot k \cdot \log \frac{n}{\delta}$
**for** $t$ *in range*$(1, T+1)$ **do**
    $current\_Centers$ = $list\_Centers[t-1]$
    $loss\_before\_swap$ = compute_loss($Y$, $current\_Centers$)
    **for** $x$ *in* $current\_Centers$ **do**
        **for** $y$ *in* $(candidates \setminus current\_Centers)$ **do**
            $loss\_after\_swap$ = compute_loss($Y$, $current\_Centers \cup x \setminus y$)
            $prob$ = $\exp(-\epsilon \cdot \frac{loss\_after\_swap - loss\_before\_swap}{2 \cdot r^2 \cdot (T+1)})$
            Save probability $prob$ for each swap.
    Choose swap $(x, y)$ with computed probabilities
    $list\_Centers$.append(($current\_Centers \cup x \setminus y$))
Choose $t \in \{1, 2, ..., T\}$ with probability in proportion to $\exp(-\frac{\epsilon \cdot compute\_loss(list\_Centers(t))}{2 \cdot r^2 \cdot (T+1)})$
**return** $list\_Centers(t)$

### Recovery of High-Dimensional Cluster Centers

These low-dimensional cluster centers now have to be transformed to high-dimensional cluster centers in the `recover` algorithm 5. Therefore, for each cluster center, a list of data points indexes assigned to that cluster center in the low-dimensional data must be created. A data point is assigned to a cluster center if the distance between the data point and this cluster center is the smallest compared to the distances between this data point and the other cluster centers. These indexes are used to obtain the high-dimensional data and

to form the cluster centers in the high-dimensional data space. For each existing cluster center, all high-dimensional data points assigned to that cluster center are summed and divided by the number of data points. To make this differentially private, Laplace noise is added to the number of data points for noise calculation and the average cluster center in each dimension.

**Algorithm 5:** recover

**input** low dimensional cluster centers: $centers\_low\_dim$, $(p, n)$ low-dimensional data matrix: $Y$, $(d, n)$ high-dimensional data matrix: $X$, number of clusters: $k$, privacy parameter: $\epsilon$, range of data: $r$
**output** high-dimensional Cluster Centers: $centers\_high\_dim$
$centers\_high\_dim$ = []
**for** $index\_of\_cluster$ *in range(k)* **do**
    $indexes\_data\_points\_assigned\_current\_cluster$ = indexes of low-dimensional data points ($Y$) that have the closest distance to current cluster center: $centers\_low\_dim[index\_of\_cluster]$
    $numberDP$ = len($indexes\_data\_points\_assigned\_current\_cluster$)
    $numberDP\_noised$ = $numberDP$ + Laplace($\frac{2}{\epsilon}$)
    **if** $numberDP\_noised < 1$ **then**
        $numberDP\_noised$ = 1
    $sum\_data\_points$ = sum_up_data_points($indexes\_data\_points\_assigned\_current\_cluster$, $X$)
    $noise\_center$ = []
    **for** $dim$ *in range(d)* **do**
        $noise\_center$.append_dimension(Laplace($\frac{2 \cdot r}{\epsilon \cdot numberDP\_noised}$))
    $center\_noised$ = $\frac{1}{numberDP} \cdot sum\_data\_points + noise\_center$
    $centers\_high\_dim$.append( $center\_noised$ )
**return** $centers\_high\_dim$

### Differentially Private k-means Algorithm

To further improve cluster centers, the `private k-means` algorithm 6 is used, which works the same way as the private Lloyd algorithm. The algorithm is also very similar to the `recover` algorithm 5. For each cluster, all data points that have the closest distance to that cluster are summed. Then this sum is divided by the number of data points assigned to that cluster, which is the new cluster center. To make this differentially private, Laplace noise is added to the number of data points for noise calculation and the average cluster center in each dimension.

**Algorithm 6:** private kmeans

**input** cluster centers: $centers\_old$, $(d, n)$ data matrix: $X$, number of clusters: $k$, privacy parameter: $\epsilon$, range of data: $r$
**output** Cluster Centers: $centers\_new$
$centers\_new$ = []
**for** $index\_of\_cluster$ *in range(k)* **do**
    $indexes\_data\_points\_assigned\_current\_cluster$ = indexes of data points ($X$) that have the closest distance to current cluster center: $centers\_old[index\_of\_cluster]$
    $numberDP$ = len($indexes\_data\_points\_assigned\_current\_cluster$)
    $numberDP\_noised$ = $numberDP$ + Laplace($\frac{2}{\epsilon}$)
    **if** $numberDP\_noised < 1$ **then**
        $numberDP\_noised$ = 1
    $sum\_data\_points$= sum_up_data_points($indexes\_data\_points\_assigned\_current\_cluster$, $X$)
    $noise\_center$ = []
    **for** $dim$ *in range(d)* **do**
        $noise\_center$.append_dimension(Laplace($\frac{2 \cdot r}{\epsilon \cdot numberDP\_noised}$))
    $center\_noised$ = $\frac{1}{numberDP} \cdot sum\_data\_points + noise\_center$
    $centers\_new$.append( $center\_noised$ )
**return** $centers\_new$

## 4.2 Improve Runtime and Clustering Results

After implementing and running the algorithm in Python, we noticed that the algorithm could not be executed efficiently even on small data sets with small dimensions and did not produce good clustering results. Therefore, we looked at the Matlab implementation [Hon17b], found many adjustments compared to the paper algorithm of Balcan et al. [Hon17a] and adapted our implementation. This section focuses on how our algorithm needs to be modified to achieve efficient runtime, memory usage and better clustering results. To achieve this goal, many different parameters are changed. We discuss the different problems, apply the changes in the Matlab implementation to our algorithm, and discuss the effect of these changes. Afterwards, our implementation is almost exactly like the Matlab implementation [Hon17b] to get a good comparison between these two implementations later.

### Further Reduction of Dimension

One problem is that the dimension of the low-dimensional data is still too high, which results in too long runtime and too much memory usage. In the `Private Clustering` algorithm 1, the random projection from the high-dimensional data to the low-dimensional data occurs in line 7. To better retain the distances between the $n$ data points and still satisfy the Johnson-Lindenstrauss Lemma, the dimension for the low-dimensional data $low\_dim$ is set to $8 \cdot \log n$ in line 3.
However, this still leads to a much too high dimension, which results in too long runtime and too high memory usage. All algorithms `candidate`, `private_partition`, `localswap and recover` must compute something in each dimension, so each dimension extends the runtime. Furthermore, and this is by far the bigger problem, a large amount of data must be saved at higher dimensions. In the `private_partition` algorithm 3, each active cube at each depth is partitioned into $2^{low\_dim}$ new cubes, and each newly partitioned cube can potentially be partitioned again into $2^{low\_dim}$ new cubes, which can happen up to $log(n)$ times.
Following this, a small dimension is required to achieve efficient runtime and memory usage. Therefore, Balcan et al. chose a much smaller dimension for reduction in their Matlab implementation [Hon17b]: $\frac{\log(n)}{2}$. This is even so small that the Johnson-Lindenstrauss Lemma no longer holds, since the dimension to be reduced to must be bigger than $log(n)$. Now the distances between data points are larger as they were in the high-dimensional data, and the cluster centers computed in the low-dimensional data are not as accurate for the high-dimensional data. In summary, the results will be worse for the high-dimensional data, but it is necessary to provide usability.
Without this reduction, our computing setup was not able to execute the `private_partition` algorithm with a data set that contained only 20 data points. This results in $2^{8 \cdot \log 20 \approx 24} \approx 16$ million new cubes, for each active cube at each depth. With the new reduction, even data sets with 100000 data points are reduced to only $\frac{\log(100000)}{2} \approx 5$ dimensions and there are just $2^5 = 32$ new cubes for each active cube at each depth of the `private_partition` algorithm.
To apply this change in our algorithm, the `Private Clustering` algorithm 1 must be modified in line 3. The new line 3 looks like this: $low\_dim = \frac{\log n}{2}$.

### Reduction of Iterations $T$

The `Private Clustering` 1, `candidate` 2, and `localswap` 4 algorithms all use for loops that run for iteration parameter $T$. These algorithms theoretically work with the iterations $T$ specified by the algorithm in the paper of Balcan et al. [Hon17a], but they

are so high that this leads to runtime problems in practice. Therefore, all of them must be reduced.

In the paper algorithm `Private Clustering` of Balcan et al. [Hon17a] and in our current `Private Clustering` algorithm 1 the number of iterations $T$ is set to $2 \cdot \log \frac{1}{\delta}$. The problem is that even $1$ iteration of the algorithm already takes some time. Each time $T$ is increased by one, the runtime increases by exactly the time it takes to compute one cluster center set. Since each $t \in T$ computes a whole new cluster center set, the algorithm could also work with only $1$ iteration. To shorten the runtime, the Matlab implementation proposed to reduce the iterations $T$ to just $1$. This change can be applied to our algorithm by setting $T$ to $1$ in line $4$ in `Private Clustering`.

In the paper algorithm `candidate` of Balcan et al. [Hon17a] and in our current `candidate` algorithm 2, the number of iterations $T$ is set to $27 \cdot k \cdot \log \frac{n}{\delta}$. Since the `private_partition` algorithm 3, which partitions each cube into sub-cubes etc., takes some time, each of the $T$ computations also takes some time. But there is another problem: computing more candidates is a problem for the runtime of the `localswap` algorithm 4. The more candidates there are, the more swaps the `localswap` algorithm must calculate. Thus, increasing the number of $T$ in `candidate` increases the runtime in two respects. The Matlab implementation proposed reducing the iterations $T$ to $3$. To apply this change to our algorithm, $T$ has to be set to $3$ in line $5$ in `candidate`.

In the paper algorithm `localswap` of Balcan et al. [Hon17a] and in our current `localswap` algorithm 4, the number of iterations $T$ are set to $100 \cdot k \cdot \log \frac{n}{\delta}$. One swap could be done very quickly. The runtime depends heavily on the number of candidates entered into the algorithm and the number of clusters searched for $k$, because the number of swaps is calculated as follows: $k \cdot (|candidates| - k)$. In the Matlab implementation, it was proposed to reduce the iterations $T$ to the number of clusters searched for $k$. These iterations are also capped by $20$. To apply this change, the code in Listing 4.1 could be added to `localswap` for line $5$.

Listing 4.1: Change for the localswap algorithm in line 5

```
1 T = k
2 if T > 20:
3     T = 20
```

The Figure 4.1 illustrates how fast the runtime increases when increasing the $T$ parameters in `candidate` and `localswap`. As a data set, we used the MNIST data set [Haf98] with $70000$ data points and $784$ dimensions. Then we executed the `Private Clustering` algorithm 1 with the following parameters: $data = MNIST, k = 10, \epsilon = 1.0, \delta = 0.1$.

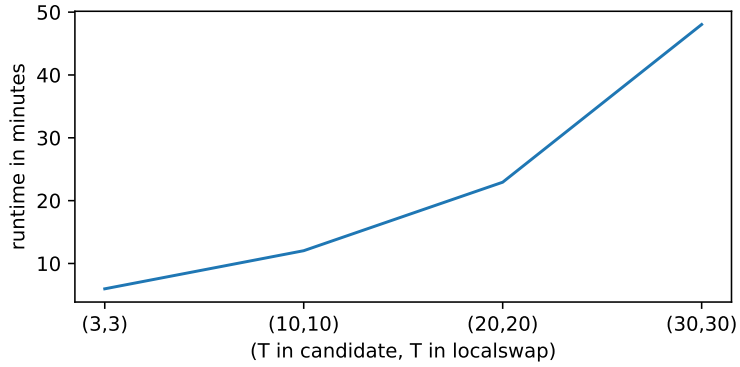


Figure 4.1: Effect of increasing iterations $T$ in the `candidate` and `localswap` algorithms on the runtime of the `Private Clustering` algorithm 1. Algorithm implemented in Python as proposed in Section 4.1 and all other adjustments from Section 4.2 applied. Further configurations: $data$ = MNIST data set, $k = 10$, $\epsilon = 1.0$, $\delta = 0.1$, $r = 255$

Already at $T = 30$, the runtime is over $48$ minutes. As the previous discussion and the Figure 4.1 suggest, the runtime increases rapidly. The values of $T$ before the reduction of $T$ in this scenario are $27 \cdot 10 \cdot \log(\frac{70000}{0.1}) \approx 3633$ for `candidate` and $100 \cdot 10 \cdot \log(\frac{70000}{0.1}) \approx 13458$ for `localswap`. With the previous calculation of iterations $T$, the runtime would thus be much too high. As a result, it is necessary to reduce the number of iteration $T$ in these algorithms.
Another side effect of these reductions is that each iteration of the algorithms receives a larger portion of the privacy budget $\epsilon$, since it must be divided among each iteration. Therefore, better results are obtained at each iteration when $T$ is lower.

### Reduction of $\gamma$

The $\gamma$ value in the `private_partition` algorithm 3 is a bound on whether a new cube is added to the active cubes, and thus whether the cube center of that new cube is added to the candidates. This is decided by whether the number of data points that lay in this cube is greater than the $\gamma$ value. More detailed, the probabilities of adding a new cube to the active cubes depends on the number of data points that lay in that cube. If there are more than $\gamma$, the probability is high, if not, the probability is low.
The problem is that for realistic data sets (such as the MNIST data set), this $\gamma$ value is so large that no new cube is added to the active cubes and, as a result, the candidate set has poor performance, because no cube centers are added as possible candidates. If these candidates already perform poorly, the other algorithms cannot improve the clustering centers later to achieve a good clustering result. So, it is not a runtime or memory problem,

but a problem that leads to poor results.
In the paper algorithm `private_partition` of Balcan et al. [Hon17a] and in our current algorithm `private_partition` 3, the $\gamma$ value is calculated by $\frac{20}{\epsilon'} \cdot \log(\frac{n}{delta})$. This value is reduced to $\frac{2}{\epsilon'} \cdot \log(\frac{n}{delta})$ in the Matlab implementation, which is a reduction by a factor of ten.
If the $\gamma$ value is not reduced, as in our current algorithm `private_partition` 3, it has a value of $40540$ (all other changes from Section 4.2 applied, further configurations: `Private Clustering`($data = MNIST, k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$)) at the MNIST data set. So there had to lay more than $40540$ data points in a cube to add this cube to the active cubes, while the data set has only $70000$ data points. Furthermore, there are $10$ different clusters in the data set and each cluster has the same number of data points, resulting in $\frac{70000}{10} = 7000$ data points per cluster. Therefore, with a bound of $\gamma = 40540$, it is not possible to obtain a candidate set that represent cluster centers sufficiently, because no cubes are added to the active cubes, resulting in no partitioning of cubes and no new cube centers added to the candidate set.
With the new calculation of $\gamma$, as proposed in the Matlab implementation, only $4054$ data points need to be present in a cube to add the cube to the active cubes, which is a realistic assumption. Therefore, the cubes are partitioned more often, and the centers of these cubes are better representations of the clusters.
To apply this change to our algorithm, the `private_partition` algorithm 3 must be changed in line $5$. The new line $5$ looks like this: $\gamma = \frac{2}{\epsilon'} \cdot \log \frac{n}{\delta}$.

## 4.3 Bugs and Corrections in the Matlab Implementation of Balcan et al.

During our first comparison between our Python implementation [Edl23] and the Matlab implementation [Hon17b], we encountered not only the adjustments in the Matlab implementation described previously in Section 4.2, but also encountered some bugs in the Matlab implementation. So, all results in the paper by Balcan et al. [Hon17a] from the Matlab implementation [Hon17b] are not based on faultless implementation. This section shows and explains the mistakes in the Matlab code, and afterwards a solution for them is proposed.

### Too Much Privacy Budget Used in `clustering.m`

The privacy budget is specified by the $\epsilon$ parameter, which must be an input parameter to the algorithm. This parameter must be divided among all algorithms that perform differentially private computations on the data. When summing up all fractions of the $\epsilon$ parameters assigned to the sub-algorithms, the sum must be at most equal to the in-

putted $\epsilon$, which is usually 1, because of the sequential composition theorem. However, the original Matlab implementation used the following privacy budget distribution in `clustering.m`:

- `candidate`: $\frac{2}{3} \cdot \epsilon$ ([Hon17b], `clustering.m`, line 31)
- `localsearch` (equals `localswap`): $\frac{1}{12} \cdot \epsilon$ ([Hon17b], `clustering.m`, line 33)
- `recover`: $\frac{1}{6} \cdot \epsilon$ ([Hon17b], `clustering.m`, line 51)
- `Lloyd / private k-means`: $\frac{1}{2} \cdot \epsilon$ ([Hon17b], `clustering.m`, line 73)
- `"cluster selection"`: $\frac{1}{12} \cdot \epsilon$ ([Hon17b], `clustering.m`, line 84)

The fractions of $\epsilon$ summed up to $1.5 \cdot \epsilon$, which is not equal to the amount of $1 \cdot \epsilon$ entered into the algorithm, and thus is too much, which is not fair. Therefore, given an input $\epsilon$, this algorithm is no longer $\epsilon$-differentially private, but $(1.5 \cdot \epsilon)$-differentially private.
Based on this unfair distribution, we created a fair privacy budget distribution that leads to a $\epsilon$-differentially private algorithm and maintains the ratios of the unfair distribution. Therefore, we calculated the fair distribution by dividing each fraction by 1.5, which leads to a sum of all fractions that equals $1 \cdot \epsilon$. The following privacy budget distribution can be used:

- `candidate`: $\frac{4}{9} \cdot \epsilon$
- `localsearch` / `localswap`: $\frac{1}{18} \cdot \epsilon$
- `recover`: $\frac{1}{9} \cdot \epsilon$
- `Lloyd / private k-means`: $\frac{1}{3} \cdot \epsilon$
- `"cluster selection"`: $\frac{1}{18} \cdot \epsilon$

We used this correct and fair privacy budget distribution for the Matlab implementation for all subsequent comparisons. We also used this distribution for our `Private Clustering` algorithm 1.

#### Too Much Privacy Budget Used in `candidate.m`

A similar mistake was made in the algorithm used to calculate the candidates. In `candidate.m`, instead of $1 \cdot \epsilon$, $1.5 \cdot \epsilon$ was again used as the privacy budget. In line 6, the `partition.m` algorithm is called and the privacy budget passed to the algorithm is $\epsilon$ divided by the iteration parameter $T$, which is set to 2. So even before the for loop

starts in line 8, $\frac{1}{2} \cdot \epsilon$ of the privacy budget is already used. During each of the $T = 2$ loop iterations, the `partition.m` algorithm is called again, and the privacy budget passed to this algorithm is $\frac{1}{2} \cdot \epsilon$. Following this, $\frac{1}{2} \cdot \epsilon$ privacy budget is used three times. The summation yields $1.5 \cdot \epsilon$ privacy budget, which is again $\frac{1}{2} \cdot \epsilon$ too much.
This problem can be solved by simply adding 1 to the denominator $T$ in line 6 and 12. After that, the privacy budget is correctly distributed by one third to each of the three function calls of the `partition.m` algorithm.

#### Incorrect Division After JL-Transform

In `clustering.m` in line 25, the low-dimensional data are divided by the square root of $p$. $p$ here is the dimension to which the high-dimensional data is reduced, i.e., the dimension of the low-dimensional data. Originally, the low-dimensional data should be divided by the square root of the high-dimensional data dimension $d$. This mistake becomes a problem when calculating the shift for the data in `candidate.m`, since the shift in each dimension can be minimum $-side\_length/2$ and maximum $side\_length/2$. In the Matlab implementation, the data is shifted in the `candidate` algorithm instead of the cube, which has the same effect. The $side\_length$ is determined by at least 2 times the infinity norm of the high-dimensional data space. Thus, if the low-dimensional data is divided just by the square root of $p$, the maximum possible value in the low-dimensional data will be higher than when divided by the square root of $d$, and therefore the possible offsets for the shifts are too small for the low-dimensional data because the low-dimensional data may contain higher values. This could lead to poorer results because the data is not shifted much in each iteration $T$ in `candidate.m` and thus the data is more similar and each partition run is more likely to find the same clusters in each run.
This bug can be solved by simply dividing the low-dimensional data by the square root of $d$ instead of the square root of $p$ in line 25 of `clustering.m`.

#### Bug in `partition.m`

In the `partition.m` algorithm is a classic software bug. In line 13, the global variable $side\_length$ is divided by 2 at each iteration of the while loop. This does not only affect `partition.m`, because this is a global variable. Also, the offset for the shift of the data depends on $side\_length$ in `candidate.m`. So, after each function call of `partition.m` in `candidate.m` the variable $side\_length$ becomes smaller and the bounds for the shift of the data also become smaller. This could lead to poorer results because the data in each iteration $T$ in `candidate.m` is not shifted as much and thus the data is more similar and it is more likely that each partition run will find the same clusters.

A solution would be to create a variable $local\_side\_length$ that is assigned the value of $side\_length$ at the beginning of `partition.m`. Then this $local\_side\_length$ can be divided in every loop iteration of `partition.m` and be further used there.

#### Wrong Noise Calculation in `recover.m`

The `recover.m` file in the Matlab implementation is part of our `recover` and `private k-means` algorithm. It simply calculates the average data point of a set of data points that are assigned to a cluster and then adds noise to that average data point.
The following bug was discovered by Chaturvedi et al. [Eri21]. The problem is that the Matlab implementation in `recover.m` in line 8 incorrectly calculates the noise for the average data point. They incorrectly used the actual number of data points to calculate the noise for the average data point, while the noisy number of data points must be used. A solution would be to insert a new line between lines 7 and 8 where $n$ is noised as proposed in line 7 of our `recover` algorithm 5. In Matlab, this line would look like this:
$n = n + random('exp', 2/\epsilon) .* (2 * random('bino', 1, 0.5) - 1);$

### 4.4 Comparison of Clustering Results

In this section, we examine whether our Python implementation [Edl23] produces similar clustering results as the Matlab implementation of Balcan et al. [Hon17b]. After making the proposed algorithm of Balcan et al. [Hon17a] efficiently runnable in Section 4.2, adding the `private k-means` algorithm in Section 4.1, and fixing the bugs in the Matlab implementation in Section 4.3, we can now compare our implementation to theirs. We expect almost the same results since the implementations are set nearly the same. Nevertheless, there are some differences that could affect the clustering results. One difference that should make the Matlab implementation a little better is that the failure probability $\delta$ is ignored. Another difference is that the Matlab implementation does not shift the cubes in the `candidate.m` algorithm. They shift the data instead of the cube, which has the same effect and should not affect the results. Last, what should make our Python algorithm better is that we chose a smaller range and therefore add less noise, when computing noise for each dimension. In our implementation, we set the range to the $l\_2$ radius of only one dimension of the data space, while the Matlab implementation sets it to the $l\_2$ radius of the data space. Our idea was to adjust the range as needed, for example, when noise is calculated for an entire data point and the $l\_2$ radius of the data space is needed. Since differentially private determination of the range of a data set is still a open question in computer science, both approaches are not differentially private in this respect.
However, as already said before, bugs are quickly made and we also have a bug in our

Python implementation [Edl23] that affects the calculation of private selections when using the exponential mechanisms. The range used in the exponential mechanisms is incorrect since we set the range to the $l\_2$ radius of only one dimension of the data space. In this scenario, it must be the $l\_2$ radius of the data space, because we need to consider the effect of one data point on the clustering loss. Our bug leads to using too small a range and too low a sensitivity, which in turn leads to too high a probability of selecting the best candidate in the set. Thus, each time we use the exponential mechanism in the `localswap` algorithm 4 and the `Private Clustering` algorithm 1, the probabilities calculated for private selection are wrong and too good. However, this bug does not effect the performance of the algorithm in our practical setting, since all objects in the candidate set have the same probability, as we will show in Section 5.3. If the range were even larger, the probabilities for each object in the candidate set would be even more evenly distributed, but as we said, it cannot be more even in our practical setting. In summary, this bug has no effect on the performance of the algorithm in our practical settings.

### 4.4.1 Experimental Setup

We will execute both implementations with different input parameters to examine how well the implementations work with different parameters and what their effects are. For the Matlab implementation, all corrections from Section 4.3 are applied, and for our Python implementation, all adjustments from Section 4.2 are applied. The criterion for how well the implementations work is always how the computed cluster centers perform on the inertia. We always executed $5$ independent runs of the algorithms and calculated the average. Both implementations also require different input parameters. Our Python implementation requires as input: data $X$, number of clusters searched for $k$, privacy parameter $\epsilon$, failure probability $\delta$ and the range of the data $r$. The Matlab implementation requires as input: data $x\_data$, number of data points $n$, number of dimensions $d$, number of clusters searched for $k$, privacy parameter $\epsilon$ and the failure probability $\delta$. Furthermore, the global variables range of the data $range$ and the $side\_length$ must be specified. $side\_length$ is specified as $2$ times the infinity norm of the data space and the range is specified as the $l\_2$ radius of the data space.

#### Data Sets

- *MNIST Data Set* [Haf98]
  The MNIST data set is a database of images of the numbers from $0$ to $9$, which is often used to compare different machine learning algorithms. Each data point is a number encoded as a $28 \times 28$ pixel grayscale image. To use this data set in the algo-

rithms, these data points must be converted to data points with only one dimension, resulting in data points with $28 \cdot 28 = 784$ dimensions. The data set consists of $60.000$ data points as the training set and $10.000$ data points as the test set. We combined both data sets, and therefore when we refer to the MNIST data set, we are referring to the combined set of $70.000$ data points and $784$ dimensions.

For the Matlab and Python implementation, we also need to specify the range of the data set. In this case, the minimum value is $0$ (complete white) and the maximum value is $255$ (maximum black). Following this, the range of the data per dimension for our Python implementation is $255$. In the Matlab implementation, the range is the $l\_2$ radius of the data set, which is $255 \cdot \sqrt{784}$. Also, the $side\_length$ must be specified, which is at least $2$ times the infinity norm of the data space. Therefore, it is $2 \cdot 255 = 510$.

- *Synthetic Data Sets*
  When we refer to synthetic data sets, these data sets are made with the $make\_blobs$ function from the scikit-learn software library [PVG+11], which generates clusters of normally distributed data points. The number of data points, the number of dimensions, and the number of cluster centers created vary for each used data set. What is the same each time is the center box of $(0, 100)$ and the standard deviation of $1.0$. Following this, the data points have values between $0$ and $100$ plus or minus $3$ with high probability, since $3$ has already less than $0.1\%$ probability with the standard deviation of $1.0$. To account for other, even more unlikely events, we consider data points between $0$ and $100$ plus or minus $5$. Therefore, the range for our Python implementation is $|-5-105| = 110$ and the range for the Matlab implementation is $110 \cdot \sqrt{100}$. The $side\_length$ for the Matlab implementation is $2 \cdot 110 = 220$.

### 4.4.2 Results

The following figures show the averages and the standard deviations computed over 5 runs.

#### Privacy Utility Tradeoff

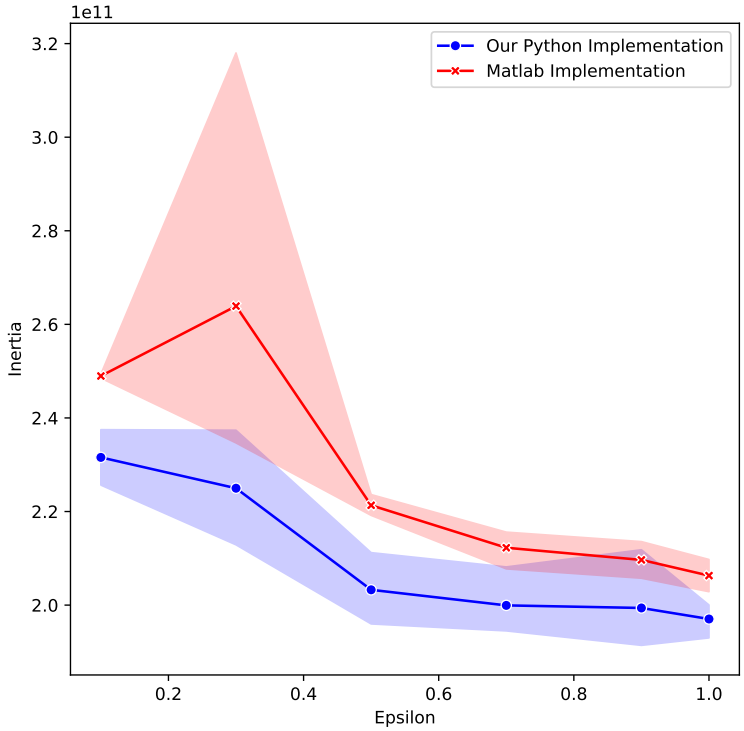


Figure 4.2: Comparison of the Matlab [Hon17b] and Python [Edl23] implementations in terms of the effect of the privacy budget $\epsilon$ on the inertia for the combined MNIST data set. Configurations for Python: data = MNIST data set, $k = 10, \epsilon = Epsilon, \delta = 0.1, r = 255$. Configurations for Matlab: data = MNIST data set, $n = 70000, d = 784, k = 10, \epsilon = Epsilon, delta = 0.1, range = 255 \cdot \sqrt{784}, side\_length = 510$.

### Effect of Number of Cluster searched for $k$

- MNIST data set

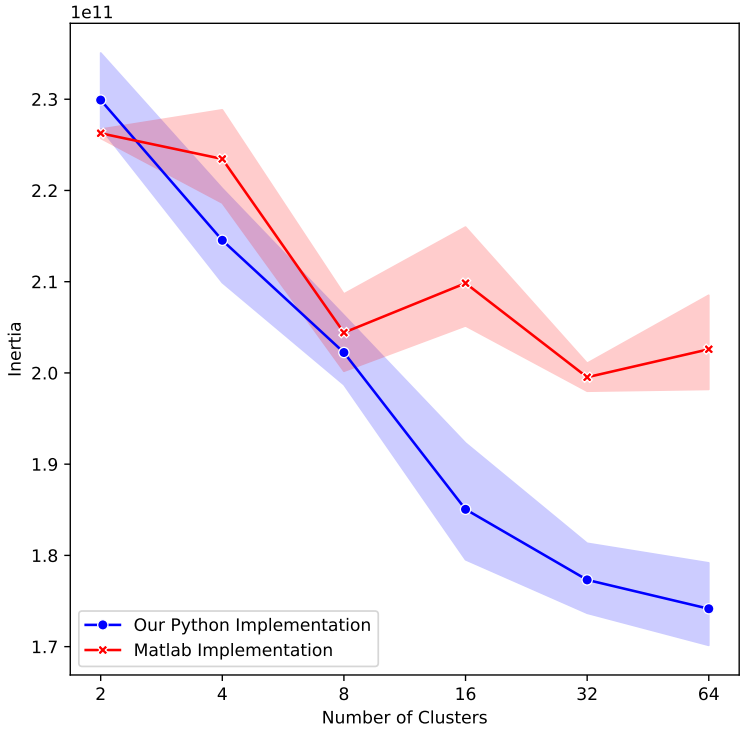


Figure 4.3: Comparison of the Matlab [Hon17b] and Python [Edl23] implementations in terms of the effect of the number of clusters searched for $k$ on the inertia for the combined MNIST data set. Configurations for Python: data = MNIST data set, $k$ = Number of Clusters, $\epsilon = 1.0, \delta = 0.1, r = 255$. Configurations for Matlab: data = MNIST data set, $n = 70000, d = 784, k$ = Number of Clusters, $\epsilon = 1.0, delta = 0.1, range = 255 \cdot \sqrt{784}, side\_length = 510$.

- Synthetic data set

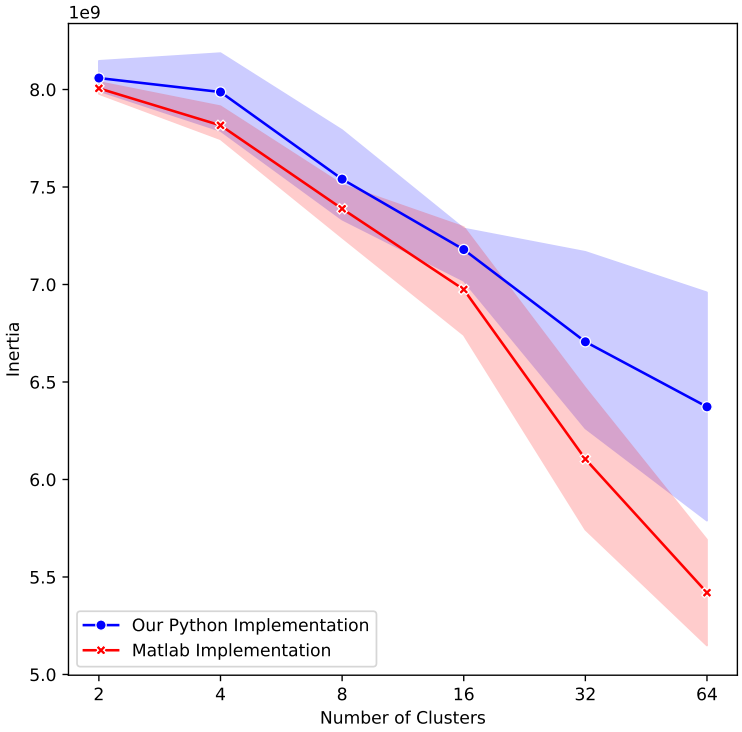


Figure 4.4: Comparison of the Matlab [Hon17b] and Python [Edl23] implementations in terms of the effect of the number of clusters searched for $k$ on the inertia for a synthetic data set. The synthetic data set contains $100000$ data points with $100$ dimensions each. Furthermore, $64$ clusters are created in the data set. Configurations for Python: data=Synthetic data set, $k$ = Number of Clusters, $\epsilon = 1.0, \delta = 0.1, r = 110$. Configurations for Matlab: data=Synthetic data set, $n = 100000, d = 100, k$ = Number of Clusters, $e = 1.0, delta = 0.1, range = 110 \cdot \sqrt{100}, side\_length = 220$.

### Effect of Dimension

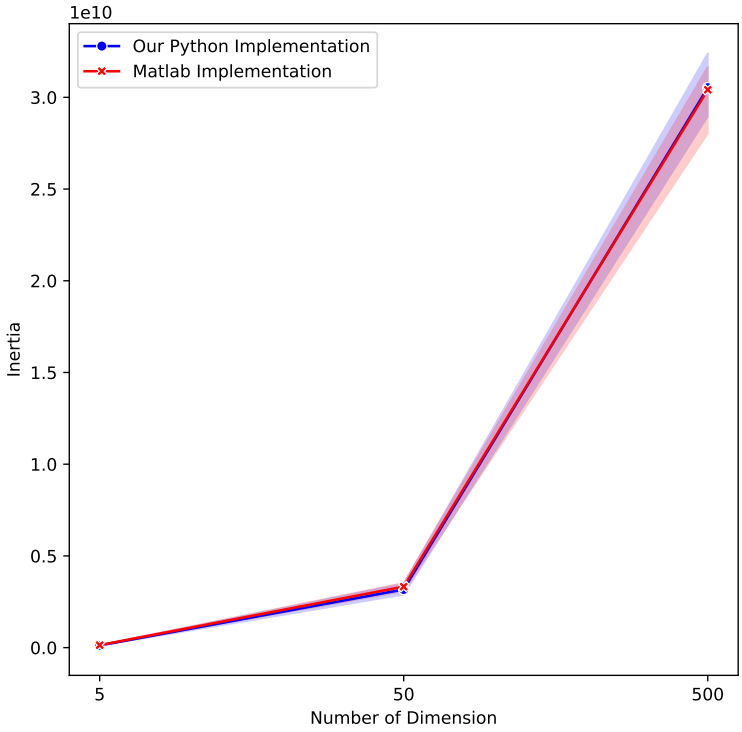


Figure 4.5: Comparison of the Matlab [Hon17b] and Python [Edl23] implementations in terms of the effect of dimensions in the synthetic data sets on the inertia. Therefore, 3 different data sets are created, all containing 100000 data points and 32 clusters. One data set has 5 dimensions, one 50 and one 500. Configurations for Python: data = Synthetic data set, $k = 32, \epsilon = 0.5, \delta = 0.1, r = 110$. Configurations for Matlab: data = Synthetic data set, $n = 100000, d =$ Number of Dimension, $k = 32, \epsilon = 0.5, delta = 0.1, range = 110 \cdot \sqrt{100}, side\_length = 220$.

### Effect of Number of Intrinsic Clusters

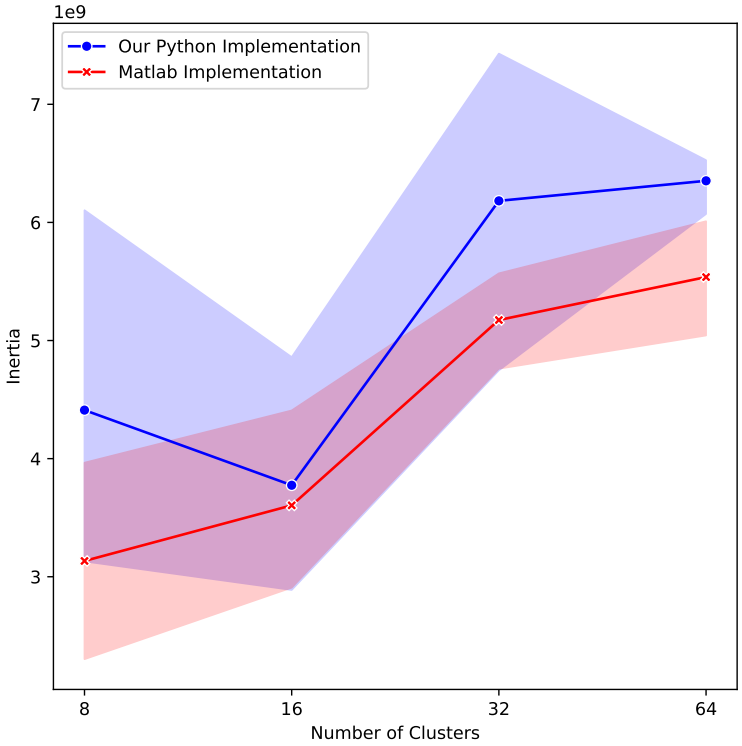


Figure 4.6: Comparison of the Matlab [Hon17b] and Python [Edl23] implementations in terms of the effect of intrinsic clusters in the data set on inertia. For each "Number of Clusters", a synthetic data set is created with that number of clusters. The algorithms are executed with that data sets and $k$ is set to the number of clusters contained in the currently used data set. Configurations for Python: data = Synthetic data set, $k$ = Number of Clusters, $\epsilon = 1.0, \delta = 0.1, r = 110$. Configurations for Matlab: data = Synthetic data set, $n = 100000, d = 100, k$ = Number of Clusters, $\epsilon = 1.0, delta = 0.1, range = 110 \cdot \sqrt{100}, side\_length = 220$.

The privacy utility tradeoff Figure 4.2 checks whether the algorithm behaves as expected. As the privacy budget parameter $\epsilon$ increases, the clustering results should also improve (inertia decreases), because the amount of noise that is added in the mechanisms decreases. As the Figure 4.2 shows, for both implementations, the inertia decreases as the $\epsilon$ increases. Furthermore, our implementation in Python consistently achieves about $10$ percent better results than the Matlab implementation.
Figures 4.3 and 4.4 examine the effect of a higher number of cluster centers searched for $k$. As expected, the more cluster center are searched for, the more cluster centers are found in average and thus the inertia decreases. When using the MNIST data set, both implementations produce almost the same results until the number of clusters searched for is set to $16$. Thereafter, our implementation achieves significantly better results. On the other hand, if the synthetic data set is used, both implementations produce almost the same results until the number of clusters searched for is set to $32$, after which the Matlab implementation achieves significantly better results.
Figure 4.5 shows the effect of increasing dimensions in synthetic data sets. As the number of dimensions increases, the inertia also increases, since each dimension introduces a distance between the cluster centers and the data points. Both implementations produce virtually the same results.
Figure 4.6 shows the effect of intrinsic clusters in synthetic data sets. As the number of intrinsic clusters increases, the inertia also increases because more clusters need to be found to achieve good clustering results, which is more difficult. Our Python implementation always performs slightly worse than the Matlab implementation.

As the results in the figures show, both implementations lead to similar results. Our Python implementation performs better on the MNIST data set, while the Matlab implementation performs better on synthetic data sets. Why the performances of the algorithms behave this way, we cannot explain. In summary, we can deduce that our implementation works the same as the Matlab implementation.

# 5 Experiments

In this chapter, we try to further improve the algorithm in terms of its performance measured by the inertia. We examine which algorithm parts have which effect on the clustering results and perform a hyperparameter search for the privacy budget distribution. We change many parameters, change and add other algorithms, and analyze the effect of these changes. We also identify practical flaws in the current implementation and seek to solve them. At last, we compare our improved version of the algorithm with other algorithms.

## 5.1 Effect of Additional Private k-means Iterations

When computing cluster centers without differential privacy, the additional input of already computed cluster centers into the k-means algorithm is always an improvement or at least not a degradation. However, when computing cluster centers in a differentially private way, it is not that simple, because the `private k-means` algorithm requires a part of the privacy budget. Therefore, it is a tradeoff between investing the privacy budget in the other parts of the algorithm or adding the `private k-means` algorithm, which also requires a portion of the privacy budget.
Originally, the algorithm proposed by Balcan et al. [Hon17a] did not append the `private k-means` algorithm 6 at the end. Yet, their Matlab implementation [Hon17b] appended 3 additional k-means iterations after applying their algorithm, and they did not correctly adjust the privacy budget as discussed in Section 4.3. Furthermore, they neither motivate its addition nor evaluate its impact. Therefore, we analyzed the effect of the addition and discussed whether it is worth investing a portion of the privacy budget in this addition. We examine here on different data sets how 3 iterations of the `private k-means` algorithm, as proposed in the Matlab implementation, affect the inertia. The question that arises is: Is it an improvement to input already computed cluster centers into the `private k-means` algorithm, considering the additional privacy budget that has to be spend?
In order to fairly compare the algorithms with and without the additional private k-means part, both algorithms should use the same privacy budget $\epsilon$. Therefore, we partitioned the privacy budget of the k-means algorithm ($\frac{1}{3}$) among all other sub-algorithms in the algorithm without the additional k-means steps. The algorithm without the additional k-means steps has the following privacy budget distribution:

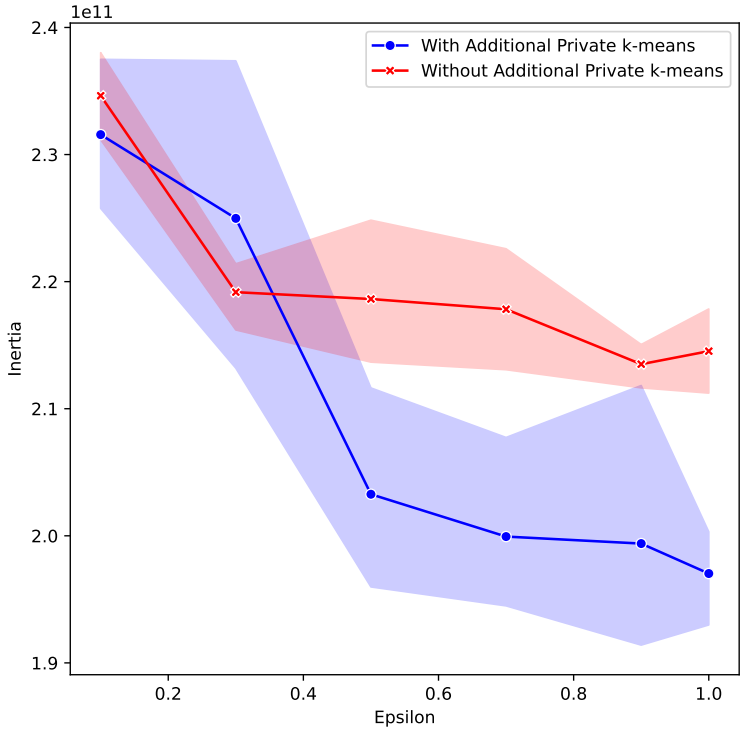


Figure 5.1: Comparison of Python implementation [Edl23] with and without the additional `private k-means` algorithm in terms of the effect of $\epsilon$ on the inertia for the combined MNIST data set. Configurations: data=MNIST data set, $k = 10, \epsilon = Epsilon, \delta = 0.1, r = 255$.

- `candidate`: $\frac{8}{18} \cdot \epsilon + \frac{8}{33} \cdot \epsilon = \frac{68}{99} \cdot \epsilon$
- `localswap`: $\frac{1}{18} \cdot \epsilon + \frac{1}{33} \cdot \epsilon = \frac{17}{198} \cdot \epsilon$
- `recover`: $\frac{2}{18} \cdot \epsilon + \frac{2}{33} \cdot \epsilon = \frac{17}{99} \cdot \epsilon$
- "cluster selection": $\frac{1}{18} \cdot \epsilon$

We always executed $5$ independent runs of the algorithms and calculated the average.

### Privacy Utility Tradeoff

For the comparison in Figure 5.1 we used the combined MNIST data set and analysed the privacy utility tradeoff again.
The implementation with the private k-means part achieved about $10\%$ better results. This is likely due to the characteristics of the MNIST data set: the data and clusters are widely distributed in the data space. Therefore, even if a cluster center is found before executing

the `private k-means` algorithm, the k-means step results in a better assignment of data points to cluster centers. So, in each run, the data points assigned to a cluster change, resulting in new, better cluster centers and the inertia decreases. For the MNIST data set, the use of the k-means algorithm is an improvement.

#### Effect of Number of Intrinsic Clusters

For this comparison in Figure 5.2, we used synthetic data sets, created by the $make\_blobs$ function from the scikit-learn software library [PVG+11]. The number of data points is set to $100000$, the number of dimensions to $100$, the bounding box to $(0, 100)$ and the standard deviation to $1.$ $4$ different synthetic data sets are created, one with $8$ clusters, one with $16$, one with $32$, and one with $64$ clusters. We then calculated the cluster centers, where the number of clusters searched for $k$ is exactly equal to the number of clusters created in the data set.
The algorithm with the additional private k-means part performs worse, which can be explained by the characteristics of the synthetic data sets. The synthetic data sets are created with clusters that are highly concentrated in small spaces. So, once the algorithm has found the clusters, before `private k-means` is executed, the clusters are represented fairly well by the cluster centers. Another calculation with the `private k-means` steps is not an improvement for the cluster centers because nothing change in the way the data points are assigned to which cluster center and only noise is added to the cluster centers. Therefore, it is more beneficial to invest the privacy budget of the private k-means part in the other sub-algorithms.

Since the algorithm performs much better on the more "realistic" MNIST data set, we leave the algorithm as suggested by the Matlab implementation of Balcan et al.

### 5.2 Hyperparameter Optimization of the Privacy Budget Distribution

We could further improve the algorithm by changing the privacy budget distribution and spending the privacy budget on the sub-algorithms that need it most to achieve the best clustering results. The algorithms `candidate` 2, `localswap` 4, `recover` 5, `private k-means` 6 and the "cluster selection" in the `Private Clustering` algorithm 1 use the privacy budget $\epsilon$, since all of them work with the data and need to add noise to make the computations $\epsilon$-differentially private. Therefore, all parts of the algorithm require a share of the $\epsilon$. The sum of all parts of the split $\epsilon$ summed must not exceed $1 \cdot \epsilon$ because of the sequential composition theorem. Then one would use more privacy budget than theoretically proven, and the algorithm would not be $\epsilon$-differentially private. The current

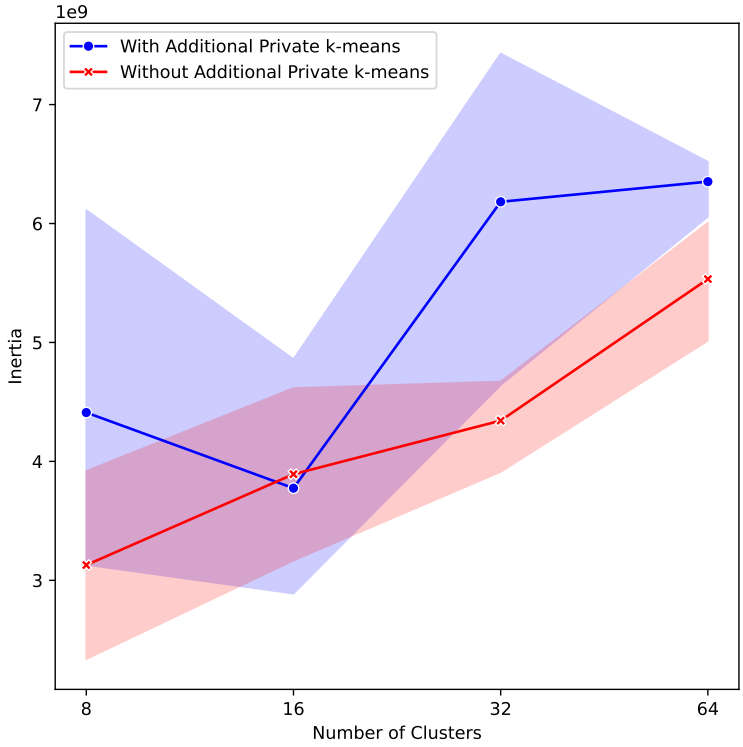


Figure 5.2: Comparison of Python implementation [Edl23] with and without the additional `private k-means` algorithm in terms of the effect of intrinsic clusters in the data set on inertia. For each "Number of Clusters", a synthetic data set is created with that number of clusters. The algorithms are executed with that data sets and $k$ is set to the number of clusters contained in the data set used. Configurations: data = Synthetic data set, $k$ = Number of Clusters, $\epsilon = 1.0, \delta = 0.1, r = 110$.

privacy budget distribution is as follows:

- `candidate`: $\frac{4}{9} \cdot \epsilon$
- `localswap`: $\frac{1}{18} \cdot \epsilon$
- `recover`: $\frac{1}{9} \cdot \epsilon$
- `private k-means`: $\frac{1}{3} \cdot \epsilon$
- "cluster selection": $\frac{1}{18} \cdot \epsilon$

We focus on the MNIST data set for the hyperparameter optimization. The current privacy budget distribution achieves an average inertia of approximately $1.97 \cdot 10^{11}$ in $5$ independent runs on the MNIST data set and the following configurations: data=MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$.
So the question is whether there is another privacy budget distribution that could achieve better results on average. Therefore, we performed a hyperparameter optimization with the following parameters: $\epsilon\_candidate, \epsilon\_localswap, \epsilon\_recover, \epsilon\_kmean, \epsilon\_cluster\_selection$. For each combination of hyperparameter values, we performed $5$ independent runs, and the average results of each combination can be found in Table 8 in the Appendix.
We do not use a hyperparameter optimization framework for this, as this is a special use case here. The parameters are interdependent, since in each run the sum of all parameters should be exactly $1$. In general, it would be more work to adjust our algorithm to a framework than to perform the hyperparameter optimization as described in this section. Our general approach in this section can be described as follows: We evaluate which sub-algorithm requires the largest part of the privacy budget to achieve the best clustering results, set the privacy budget value for that sub-algorithm fix, and repeatedly evaluate for the remaining sub-algorithms which sub-algorithm requires the next largest part, and set the values fix there until all privacy budget values for the sub-algorithms are set. In doing so, we set the privacy budget values for the sub-algorithms that require the most budget one after another, resulting in a privacy budget distribution that should yield the best clustering results on average.

At first, we examined whether there is an sub-algorithm that requires at least a bigger part of the privacy budget to achieve good clustering results. This ignores correlations between the sub-algorithms, but is necessary to get an idea which sub-algorithm requires much of the budget and therefore reducing the number of possible combinations. Therefore, we set the privacy budget high $(0.9)$ for one of the sub-algorithms while the others are low $(0.025)$, which leads to the following distributions and results in Table 5.1. The

| Privacy Budget Distribution ($\epsilon\_candidate, \epsilon\_localswap, \epsilon\_recover, \epsilon\_kmean, \epsilon\_cluster\_selection$) | inertia |
|---|---|
| (0.025, 0.025, 0.025, 0.025, 0.9) | 219589656245 |
| (0.025, 0.025, 0.025, 0.9, 0.025) | 231924610624 |
| (0.025, 0.025, 0.9, 0.025, 0.025) | 219796720934 |
| (0.025, 0.9, 0.025, 0.025, 0.025) | 225743678524 |
| (0.9, 0.025, 0.025, 0.025, 0.025) | 201976751799 |

Table 5.1: Results of inertia for different privacy budget distributions. Algorithm was executed with the following configurations: data = MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$.

results of Table 5.1 suggest that small shares of the privacy budget in the `candidate` algorithm 2 lead to poor performance. Therefore, the `candidate` algorithm should use the largest share of the overall budget.
To further evaluate this thesis, we tried some distributions for low values of the privacy budget in the `candidate` algorithm, leading to the distributions and results in Table 5.2.

| Privacy Budget Distribution ($\epsilon\_candidate, \epsilon\_localswap, \epsilon\_recover, \epsilon\_kmean, \epsilon\_cluster\_selection$) | inertia |
|---|---|
| (0.2, 0.15, 0.2, 0.4, 0.05) | 198577924821 |
| (0.2, 0.2, 0.15, 0.4, 0.05) | 204451801163 |
| (0.2, 0.2, 0.4, 0.15, 0.05) | 203827994802 |
| (0.3, 0.15, 0.2, 0.3, 0.05) | 197695862459 |
| (0.3, 0.2, 0.15, 0.3, 0.05) | 199555466986 |
| (0.3, 0.2, 0.3, 0.15, 0.05) | 202081990049 |

Table 5.2: Results of inertia for different privacy budget distributions. Algorithm was executed with the following configurations: data = MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$.

Since the current privacy budget, with $\epsilon\_candidate = \frac{4}{9} > 0.3$, used produce better results than when $\epsilon\_candidate = 0.2$ or $\epsilon\_candidate = 0.3$, we assume that at least a bigger part of the budget has to be invested in the `candidate` algorithm. For this reason, and also to keep the number of further distributions to be tried small, we do not further examine small $\epsilon$ values for the `candidate` algorithm. We set the following values for each part of the algorithm, compute all possible combinations that sum to 1, and execute the algorithm with these combinations of the privacy budget distributions.

- $\epsilon\_candidate$ = [0.4, 0.5, 0.6, 0.7, 0.8]
- $\epsilon\_localswap$ = [0.05, 0.1, 0.15, 0.2, 0.25, 0.3]
- $\epsilon\_recover$ = [0.05, 0.1, 0.15, 0.2, 0.25, 0.3]

- $\epsilon\_kmean$ = [0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.4]
- $\epsilon\_cluster\_selection$ = [0.05]

This results in 72 possible combinations that we tried and the results are included in Table 8 in the Appendix.
We also added some combinations when $\epsilon\_candidate$ is set to $0.7$ and $0.8$. We set $\epsilon\_cluster\_selection = 0.05$ because in the current implementation we only compute one cluster that can be selected. We first examine the average inertia when $\epsilon\_candidate$ is set to a fixed value, resulting in the following average values in Table 5.3. Table 5.3

| $\epsilon\_candidate$ | average inertia |
|---|---|
| 0.4 | 196181022615 |
| 0.5 | 195103649193 |
| 0.6 | 195434041992 |
| 0.7 | 193355074600 |
| 0.8 | 201386600601 |

Table 5.3: Results of average inertia at different values of $\epsilon\_candidate$. Algorithm was executed with the following configurations: data = MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$.

indicates that the `candidate` algorithm requires a large share of the privacy budget to achieve sufficient results until an upper limit is reached. Thus, the sweetspot for the privacy budget share for the `candidate` algorithm is $\epsilon\_candidate = 0.7$ and we fix this value for the next computations. When the $\epsilon$ share in other sub-algorithms is fixed at one value, the averages of these combinations do not show a noticeable pattern. The results for this can be found in the Appendix in Tables 9, 10, 11.

Now that we know where the largest part of the privacy budget should be spent, we further examine which sub-algorithm should receive the second largest part. Therefore, we added more combinations of privacy budget distributions where $\epsilon\_candidate = 0.7$.
In Table 5.4 are the averages of different privacy budget values for $\epsilon\_kmean$, when $\epsilon\_candidate$ is set to $0.7$. As Table 5.4 suggests, investing a larger part of the budget in $\epsilon\_kmean$ improves probably the results. So investing $0.2$ in $\epsilon\_kmean$ is probably the best solution.

The rest of the privacy budget is shared between `localswap` and `recover`. Therefore, we again added some privacy budget distributions where $\epsilon\_candidate = 0.7$ and $\epsilon\_kmean = 0.2$. The results indicate that it is better to invest a larger part of the privacy budget in `recover` than in `localswap`. Therefore, we chose $\epsilon\_recover = 0.04$ and

| $\epsilon\_kmean$ | average inertia |
|---|---|
| 0.025 | 197221349152 |
| 0.05 | 193711469626 |
| 0.0833 | 193523276360 |
| 0.1 | 191700195227 |
| 0.15 | 195511547391 |
| 0.2 | 190705907823 |
| 0.225 | 199292091853 |

Table 5.4: Results of average inertia at different values of $\epsilon\_kmean$ when $\epsilon\_candidate$ is set to $0.7$. Algorithm was executed with the following configurations: data = MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$

$\epsilon\_localswap = 0.01$. Our improved privacy budget distribution for the MNIST data set is:

- `candidate`: $0.7 \cdot \epsilon$
- `localswap`: $0.01 \cdot \epsilon$
- `recover`: $0.04 \cdot \epsilon$
- `private k-means`: $0.2 \cdot \epsilon$
- "cluster selection": $0.05 \cdot \epsilon$

We executed $5$ independent runs with this privacy budget distribution and the MNIST data set and obtained $1,92 \cdot 10^{11}$ as the inertia, which is an improvement on the previous inertia of $1,97 \cdot 10^{11}$.

This leads to the following conclusions. We identified that the `candidate` algorithm requires a large part of the privacy budget to compute a good set of possible cluster centers. If this set contains too few centers or too bad centers, no algorithm can significantly improve the cluster centers afterward. The swaps of cluster centers in `localswap` do not have an effect on the inertia and do not need much of the privacy budget. The recovery of the low-dimensional data to the high-dimensional data requires at least a small part of the privacy budget. The `private k-means` algorithm requires the second highest part of the privacy budget. If it is too low, too much noise will be added to the centers, resulting in inaccurate centers. Cluster selection is currently not important.

In fact, the distribution after the hyperparameter optimization is very similar to the distribution implemented by Balcan et al. in the Matlab implementation [Hon17b]. The `candidate.m` algorithm got by far the most budget, then the `private Lloyd`

algorithm, the `recover.m` algorithm, the `localsearch.m` algorithm and the cluster selection.

## 5.3 Effect of Higher Iterations

We already showed in Section 4.2 that the number of iterations $T$ in each sub-algorithm in the proposed algorithm of Balcan et al. [Hon17a] is too high, resulting in an inefficient runtime. So far, we have used the number of iterations $T$ of the Matlab implementation [Hon17b]. In this section, we analyze whether we can find a different number of iterations $T$ in each sub-algorithm that will improve the results and still have an acceptable runtime. Therefore, we need to examine the number of iterations $T$ for the `Private Clustering` algorithm 1, the `candidate` algorithm 2, the `localswap` algorithm 4 and the `private k-means` algorithm 6. During that, we also noticed that the swaps in the `localswap` algorithm are completely random in practice. For the following comparisons, we used the MNIST data set and made $5$ independent runs for each result and calculated the respective average.

### Effect of $T$ in `candidate`

Since the runtime of the `localswap` algorithm 4 and the `Private Clustering` algorithm 1 highly depends on the output and runtime of the `candidate` algorithm 2, we start by examining the iterations $T$ in the `candidate` algorithm. Currently, $T$ is set to $3$ in the `candidate` algorithm. So we measured the runtime and inertia while increasing the number of iterations $T$ in the `candidate` algorithm 2, and the results can be seen in Figure 5.3. As expected, the runtime increases with higher $T$ in the `candidate` algorithm. In fact, the runtime increases in a staircase fashion. This means that the runtime increases for some $T$, then decreases just a little bit for a increasing $T$, and then increases again. This behaviour can be observed because the setting $T$ in the `candidate` algorithm also affects the $\gamma$ value in the `private_partition` algorithm 3. The $\gamma$ value is basically a threshold for whether to add a new cube to the active cubes and thus a new cube center to the candidates. More precisely, $\gamma$ is compared to the number of data points that lay in a currently considered cube. If there are more data points than $\gamma$ in the current cube, the cube is added to the active cubes. $\gamma$ is affected by several parameters: the number of data points $n$, $\epsilon$ and $\delta$. Since in line 11 of the `candidate` algorithm 2 the $\epsilon$ used for $\gamma$ is divided by the number of iterations $T$, $T$ affects $\gamma$. In short, the behavior can be described such that when $T$ increases, $\gamma$ also increases. For this reason, in the `private_partition` algorithm, fewer new cubes are added to the active cubes as $T$ increases, resulting in a shorter runtime for each iteration $T$. Each time the runtime in Figure 5.3 gets a little shorter as

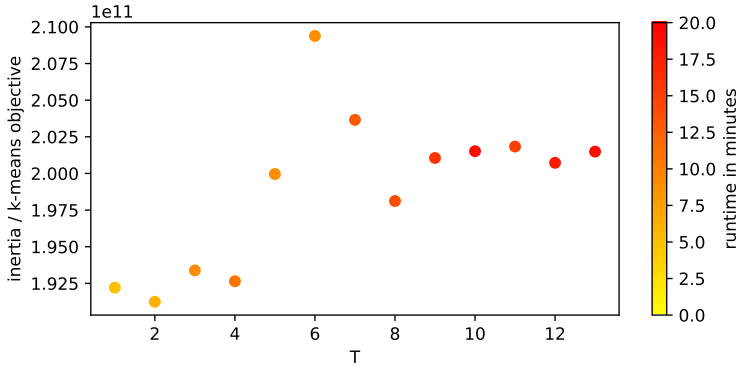


Figure 5.3: Effect of higher iterations $T$ in the `candidate` algorithm on the inertia and runtime. Configurations: data=MNIST data set, $\epsilon = 1.0, k = 10, \delta = 0.1, r = 255$. For each $T$ five independent runs were made and the respective average of inertia and runtime was calculated.

$T$ is increased, a threshold is reached for the number of data points in the spaces of the partitioned cubes, resulting in fewer and fewer new cubes being added. However, by increasing $T$, more candidate sets are computed, which leads to a longer runtime in general. This "staircase" behaviour for the runtime continues as $T$ increases until the threshold $\gamma$ is so high that no new cubes are added to the active cubes in the `private_partition` algorithm. Then the runtime will increase steadily as $T$ increases, since at each iteration of the `private_partition` algorithm the initial cube is not partitioned and only the center of that initial cube is returned as a candidate set.
In addition, we discovered a counterintuitive impact on the inertia when $T$ is increased. The inertia actually worsens as the number of iterations $T$ is increased, which is also related to the affect of $T$ on $\gamma$. So, as $T$ increases, leading to an increase in $\gamma$, less new cubes are added during partition. Therefore, the candidate set contains less centers of cubes and also worse centers, because the cubes that cover the cluster spaces more accurately are less likely to be added to the active cubes. Thus, increasing $T$ results in worse candidate sets issued by each `private_partition` call.
Since the runtime and the inertia increases as $T$ increases, the best solution is to choose a small $T$. The best option in Figure 5.3 is $T = 2$. Since we previously used $T = 3$ and to avoid the worst case that the initial cubes are so poorly chosen that there are not many data points in the initial cube, we stay with $T = 3$.

### Effect of $T$ in `localswap` and Effect of $\epsilon$ in `localswap`

Next, we examine the influence of $T$ on the `localswap` algorithm 4. The runtime increases steadily with each iteration $T$, since in each iteration $T$ all possible swaps are

considered and one is selected. The effect of increasing $T$ on the inertia is shown in Table 5.5. The results in Table 5.5 show that the number of swaps has no noticable effect on the

| $T$ in `localswap` | average inertia |
|---|---|
| 2 | 192913063310 |
| 4 | 194210141024 |
| 10 | 193384099925 |
| 20 | 194526067700 |

Table 5.5: Effect of increasing iterations $T$ in the `localswap` algorithm 4 on the inertia. Configurations: data = MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$.

outcome of the algorithm. The explanation for this behavior is that the probabilities for the different swaps are almost always the same, resulting in swaps that are completely random. More specifically, the calculation of the inputs to the exponential function in line $12$ in `localswap` 4 is always similar, resulting in similar outputs of the exponential function regardless of which swap is considered. Therefore, it does not matter how many swaps are performed, the result is the same as for the first $k$ randomly selected cluster centers. The probabilities for the swaps would differ more if the difference in the clustering loss increased or if a larger part of the privacy budget was spent in the `localswap` algorithm. We now examine how much privacy budget $\epsilon$ had to be spent on the `localswap` algorithm to achieve probabilities that actually affect the swap selection. Therefore, we chose realistic values for the clustering loss that we observed in the calculations with the MNIST data set. The clustering loss often had values between $4000000$ and $5000000$. For the experiment, we chose $4500000$ as the clustering loss before the swap and two different clustering losses after the swap, such that one swap increases the clustering loss and the other swap decreases the clustering loss. For the "bad" swap with the increasing clustering loss, we chose $5000000$ as the clustering loss, and for the "good" swap with the decreasing clustering loss, we chose $4000000$ as the clustering loss. The calculation for the probabilities ("prob") is done in line $12$ in the `localswap` algorithm 4: $\exp(-\epsilon \cdot \frac{loss\_after\_swap - loss\_before\_swap}{2 \cdot (T+1) r^2})$. The actual probabilities are these values in proportion to each other. So we compute the probabilities for only $2$ swaps and these swaps have a noteworthy effect on the clustering results. Furthermore, we leave the number of iterations as $T = k = 10$ as this also affects the probabilities and $r$ is set to $255$. As Table 5.6 shows, the probabilities for both swaps are the same when $\epsilon$ in `localswap` is set to $0.01$, as in the current privacy budget distribution. Even if we would spend the entire privacy budget $\epsilon = 1.0$ into the `localswap` algorithm, the probabilities for the swaps would still be the same for all possible swaps, given that the chosen clustering loss values are far apart. As Table 5.6 shows, the probability ratios become distinctive when $\epsilon$ is set

| $\epsilon$ for probability calculations in `localswap` | "good" swap $\approx$ "prob" | "bad" swap $\approx$ "prob" | probability ratio $\approx$ (good - bad) |
|---|---|---|---|
| 0.01 | 1.0035 | 0.99651 | 50% - 50% |
| 0.1 | 1.03557 | 0.9656 | 52% - 48% |
| 1.0 | 1.4184 | 0.705 | 66% - 34% |
| 2.0 | 2.012 | 0.497 | 80% - 20% |

Table 5.6: Effect of increasing $\epsilon$ on the probabilities for the swaps in the `localswap` algorithm 4.

to $2.0$. Then, the "good" swap actually had a higher probability than the "bad" swap, and the "good" swap is selected more often, which leads to better clustering results. If we set $\epsilon = 2.0$, the overall privacy budget would become too high for privacy considerations, since the algorithm should be $\epsilon = 1.0$-differentially private.
During this examination, we also found that the probability calculation for the cluster center set selection in line $16$ in the `localswap` algorithm 4 is also not effective. Again, the problem is that the probabilities are too similar, resulting in a completely random selection. We also examined how much of the privacy budget $\epsilon$ had to be spent in the `localswap` algorithm to achieve effective probabilities for the cluster center set selection. Therefore, from our values observed during the execution of the algorithm, we chose one "good" cluster center set with a clustering loss of $4000000$ and one "bad" cluster center set with a clustering loss of $5000000$. Then we changed the $\epsilon$ in the `localswap` algorithm and calculated the probabilities when only these two are available for selection. The calculation for the probabilities ("prob") is done in line $16$ in the `localswap` algorithm 4: $\exp(-\frac{\epsilon \cdot clustering\_loss}{2 \cdot (T+1) \cdot r^2})$. As the Table 5.7 shows, the same problem applies here as in the

| $\epsilon$ for probability calculations in `localswap` | "good" cluster center set $\approx$ "prob" | "bad" cluster center set $\approx$ "prob" | probability ratio $\approx$ (good - bad) |
|---|---|---|---|
| 0.01 | 0.9724 | 0.9657 | 50% - 50% |
| 0.1 | 0.756 | 0.705 | 52% - 48% |
| 1.0 | 0.061 | 0.03 | 67% - 33% |
| 2.0 | 0.003 | 0.00092 | 76% - 24% |

Table 5.7: Effect of increasing $\epsilon$ on the probabilities for the cluster selection in the `localswap` algorithm 4.

probability calculations of the swaps in the previous subsection. Even if we spent the entire privacy budget $\epsilon$ in `localswap` $\epsilon = 1.0$, the probabilities for the swaps would still be quite likely, given that the chosen clustering loss values are far apart.

In summary, the `localswap` algorithm has no effect on the clustering results and acts

like a random selection of cluster centers in the candidates set for reasonable privacy parameters. To achieve better runtime while producing the same results, we could just randomly select $k$ cluster centers from the candidates set. Maybe the swaps work better on other data sets, and since we present another improvement in Section 5.4, we simply set $T = k$ for now.

Note here that due to our bug in calculating the probabilities for the private selections described in Section 4.4, all probability calculations are wrong and too good. Therefore, it would require an even higher privacy budget $\epsilon$ invested in the algorithm to achieve distinct probability ratios. Also without the bug, the `localswap` algorithm would behave like a random selection of cluster centers in our practical setting.

#### Effect of Number of Iterations $iter$ in Private k-means

Now we examine the iterations $iter$ of the `private k-means` algorithm 6 in line $12 - 14$ in the `Private Clustering` algorithm 1. We increase the number of iterations $iter$, run the `Private Clustering` algorithm and calculate the respective average inertia. Table 5.8 shows that it makes no sense to change the current number of $iter = 3$ for the

| $iter$ for `private k-means` | average inertia |
|---|---|
| 1 | 199505463120 |
| 2 | 193758149394 |
| 3 | 192212684233 |
| 4 | 194465532039 |
| 5 | 193562194777 |
| 6 | 193931967432 |
| 8 | 194630942625 |
| 10 | 193358536310 |
| 20 | 205197742291 |

Table 5.8: Effect of increasing iterations $iter$ of the `private k-means` algorithm 6 in the `Private Clustering` algorithm 1 on the inertia. Algorithm was executed with the following configurations: data = MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$.

k-means algorithm. The current configuration of $iter = 3$ appears to be a sweet spot for the MNIST data set. If it is smaller than 3, the results are worse because the effect of the k-means algorithm, which improves the cluster by calculating the mean values of the data points assigned to a cluster as new cluster centers, is not working yet. The results also get worse if the number of iterations increases too much, since the noise that might be added increases as the number of iterations $iter$ increases due to the sequential composition the-

orem. Therefore, as before, we chose 3 as the number of iterations $iter$ for the `private k-means` algorithm.

#### Effect of $T$ in `Private Clustering`

Next, we examine the effect of an increasing number of iterations $T$ in the `Private Clustering` algorithm 1. Table 5.9 shows that as the number of iterations $T$ increases,

| $T$ in `Private Clustering` | average inertia |
|---|---|
| 1 | 192205648573 |
| 2 | 193893420435 |
| 3 | 203428190298 |
| 4 | 213852064420 |
| 5 | 209648299764 |

Table 5.9: Effect of increasing iterations $T$ in the `Private Clustering` algorithm 1 on the inertia. Algorithm was executed with the following configurations: data = MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$.

the inertia also increases. This is because the privacy budget $\epsilon$ is divided equally among each iteration $T$ due to the sequential composition theorem, resulting in more noise that might be added in one iteration. Therefore, every computation of a set of cluster centers achieves worse results than if only one set of cluster centers is calculated. We also stay with the current configuration of $T = 1$.

### 5.4 Further Improvements

In this section, we tested some adjustments to the `Private Clustering` algorithm 1 to further improve the clustering results on the MNIST data set.

#### Expansion of The Candidates Set

We already discussed in Section 4.2 the effect of $\gamma$ on the `private_partition` algorithm 3. It is a bound in the algorithm that decides whether a new cube is added to the active cubes and thus the center of the new cube to the candidates set. If the number of data points in a new cube is greater than the $\gamma$ bound, then the cube is added to the candidates set. Therefore, reducing this bound leads to more and more accurate candidates as the space of the cubes becomes more accurate to the clusters. We already reduced $\gamma$ in Section 4.2 from $\gamma = \frac{20}{\epsilon'} \cdot \log(\frac{n}{\delta})$ to $\gamma = \frac{2}{\epsilon'} \cdot \log(\frac{n}{delta})$. To maintain $\epsilon$-differential privacy, we can further reduce it to $\gamma = \frac{1}{\epsilon'} \cdot \log(\frac{n}{\delta})$. We extract from the differential privacy proofs of Balcan et al. [Hon17a] (Theorem 2) that just the calculation of $\epsilon' = \frac{\epsilon}{2 \cdot \log n}$ is relevant to

maintain differential privacy. The privacy budget $\epsilon$ is divided by $2$ because modifications to a single data point affect the number of data points contained in a cube of at most two active cubes. It is also divided by $\log n$ to distribute the privacy budget to each iteration of the while loop in the `private_partition` algorithm 3. Therefore, if we use $\gamma = \frac{1}{\epsilon'}$, this yields the bound to preserve differential privacy. Multiplying $\gamma$ by $20$ and $\log(\frac{n}{\delta})$ serves to make the `private_partition` algorithm computationally efficient, so that the bound is not so low that too many cubes are added to the active cubes.
We changed in the `private_partition` algorithm 3 line $4$ to $\gamma = \frac{1}{\epsilon'} \cdot \log(\frac{n}{delta})$. Then we executed the `Private Clustering` algorithm 1 algorithm with the following configurations: data = MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$. And obtained on the average of $5$ independent runs the inertia: $1.9 \cdot 10^{11}$. Before this reduction of $\gamma$, we obtained results around $1.92 \cdot 10^{11}$ on average. Therefore, this is an improvement and it only caused a little more runtime.

### Improve Probabilities for Private Selections

We already discussed in Section 5.3 that the probabilities for the private selections of swaps and cluster center sets in the `localswap` algorithm 4 are always the same for the MNIST data set and thereby offer no benefit on the performance of the clustering results. Since we also must invest a share of the privacy budget in the `localswap` algorithm, which thus cannot be invested in other sub-algorithms, the performance of the clustering results becomes even worse as more noise is added in the other sub-algorithms. In this subsection, we improve the probabilities for the private selections.
Currently, all private selections are performed using the exponential mechanism. The problem is that the mechanism is not good at selecting a candidate in a candidate set when there are only small changes in the scoring function. Thus, relatively small changes in the scoring functions do not affect the final probabilities as much. But in our case, considering these small changes would actually improve the cluster centers.
Therefore, we investigated whether there is already a solution to this problem. We found a differentially private mechanism called Permute-and-Flip [Dan20], which is an alternative to the exponential mechanism. The `Permute-and-Flip` algorithm 7 randomly selects one of the candidates and calculates a probability for this candidate based on the exponential mechanism. Then a coin is tossed to decide if that candidate should be returned. The probability of heads is the calculated probability for that candidate, and if the coin toss results in heads, that candidate is returned.
This encourages the mechanism to return candidates with a higher probability even more in a differentially private manner. McKenna and Sheldon [Dan20] have shown that the results of this mechanism are at least as good as those of the exponential mechanism.

**Algorithm 7:** Permute-and-Flip

**input** $\epsilon$, sensitivity $= \Delta$, utility scores $= U$, candidates $= R$
**output** candidate: $r$
**for** $r$ *in* $RandomPermutation(R)$ **do**
    $p = \exp(\frac{\epsilon \cdot U(r)}{2 \cdot \Delta})$
    **if** $Bernoulli(p)$ **then**
        **return** r

The Permute-and-Flip mechanism is already implemented in the differential privacy library in Python [Lev19], which makes its integration into our algorithms less complicated. The algorithm requires as input the privacy budget $\epsilon$, the sensitivity of the scoring function and the utility values of each candidate.
We first integrate the `Permute-and-Flip` mechanism into the swap selection in lines $12 - 14$ of the `localswap` algorithm 4. We can modify the `localswap` algorithm by replacing line 12 and 13 with a list that saves the loss difference for each swap. Afterwards we can replace line 14 with the `Permute-and-Flip` algorithm, which selects a swap. As input it receives the privacy budget $\frac{\epsilon}{2 \cdot T+1}$, sensitivity = $2 \cdot range\_data^2$ and as utility values the list with the calculated loss differences. Then the mechanism outputs a swap. Listing 5.1 contains the new code for lines $12 - 14$.

Listing 5.1: Modification for the localswap algorithm in lines 12-14

```
Save loss_after_swap − loss_before_swap in list util
Choose swap (x, y) with Permute-and-Flip(ϵ = ϵ/(2·(T+1)), sensitivity = 2 · r², utility = util)
```

For the cluster center selection in the `localswap` algorithm in line 16, we proceeded accordingly. There we also created a list containing the clustering loss for each cluster center set, which is the utility list input for the Permute-and-Flip algorithm. Again, we set $\frac{\epsilon}{2 \cdot T+1}$ as the privacy budget and as the sensitivity choose $2 \cdot range\_data^2$. Now we replace the exponential mechanism with the Permute-and-Flip mechanism in the `localswap` algorithm 4.

Listing 5.2: Modification for the localswap algorithm in line 16

```
Select a set of cluster centers list_Centers(t) from list_Centers using
Permute-and-Flip(ϵ = ϵ/(2·(T+1)), sensitivity = 2 · r²,
utility =clustering losses of each list_Centers(t))
```

We also further examined the effect of the Permute-and-Flip mechanism on the probabilities for the swap selection and the cluster selection, as we did with the exponential mechanism in Section 5.3. To calculate the probabilities for the swap selection, we again

choose the same observed clustering loss values for the "good" swap, which reduces the clustering loss, and a "bad" swap, which increases the clustering loss. We observe a difference, but the difference is so small that it does not affect the inertia. With the current privacy budget in `localswap` of $\epsilon = 0.01$ and a "bad" swap that increases clustering loss by $500.000$ and a "good" swap that decreases clustering loss by $500.000$, both swap still have the same probability. If we were to invest $\epsilon = 1.0$ and run the experiment, the probability of the "good" swap would be $58\%$ and the probability of the "bad" swap would be $42\%$. Compared to the old probabilities of $52\%$ for the "good" swap and $48\%$ for the "bad" swap, this is an improvement.
Also, the probabilities for selecting the cluster center sets were not significantly different from the probabilities when using the exponential mechanism. We leave it implemented because it provides at least a small improvement and cannot worsen the results, but it still does not improve the overall clustering results.
We also replaced the exponential mechanism in the `Private Clustering` algorithm 1 in line $15$ with the Permute-and-Flip mechanism. Since we only compute one cluster center set, this has no affect.
In summary, the Permute-and-Flip mechanism works better than the exponential mechanism in theory, but in our specific example, it has no effect on the results because the improvement is not sufficient.

Note here that we applied in our Python implementation [Edl23] the same bug to the Permute-and-Flip algorithm as to the exponential mechanism described in Section 4.4. Therefore, all probability calculations are wrong and too good in this subsection, and it would require an even higher privacy budget $\epsilon$ invested in the algorithm to achieve distinct probability ratios. Also without the bug, the Permute-and-Flip algorithm would not be an improvement in our practical setting.

#### Improvement of The Candidates Set

Since we did not find a solution to select the better cluster centers in the candidate set in the `localswap` algorithm 4, we try to improve the candidate set in the `private_partition` algorithm 3. The candidate set essentially contains the centers of the cubes that are partitioned. Our general idea is that cube centers from deeper depths are better representations of clusters than cube centers from higher depths because the cubes from deeper depths better cover the space where many data points are located. If there are more data points in an area, and therefore potentially a cluster, the algorithm partitions the cubes in that area more often, and each time this happens, the center of the cube becomes a better representation of that cluster. We select the centers of the cubes starting from the last depth

to the first depth until we receive at least $k$ candidates. Since the `private_partition` algorithm is differentially private and our candidate reduction is data independent, differential privacy is preserved.
To implement this, we modified the `private_partition` algorithm 3 in line 22.

Listing 5.3: Modification for the private partition algorithm in line 22

```
1 Assign C as the last cube centers that are added to C until the depth,
2 where the number of cube centers reaches k. Including the cube centers from
3 this depth.
```

This results in an inertia of $1.88 \cdot 10^{11}$, with the configurations: data=MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$, which is an improvement on the previous inertia of $1.9 \cdot 10^{11}$.
However, this reduction does not necessarily represent an improvement. If the `private_partition` algorithm finds a cluster at a relatively large depth and stops the partitioning of the cube there, and continuous partitioning other cubes at deeper depths, then that cluster cannot be found by the algorithm. Therefore, well-distributed data sets work better with this modification. Since it works in our case, we leave the implementation as described.

## 5.5 Comparison to Other Algorithms

After improving our implementation of the Balcan et al. [Hon17a] algorithm, we are now ready to compare our improved algorithm with other algorithms. Our algorithm [Edl23] contains the following modifications:

- Improved privacy budget distribution
- Reduction of $\gamma$ in the `private_partition` algorithm 3
- Replacing the exponential mechanisms with the Permute-and-Flip mechanism
- Reduction of the candidate set in the `private_partition` algorithm 3

The algorithm with all modifications can be found in the Appendix under Pseudocode.
We based our choice of algorithms for comparison on which algorithms Balcan et al. [Hon17a] used in their comparison. Therefore, we compared our improved algorithm to a non private k-means++ algorithm from the sklearn.cluster Python library [PVG+11], another differentially private k-means algorithm [Hol22] and the Matlab implementation [Hon17b] of the algorithm of Balcan et al. [Hon17a]. The other differentially private k-means algorithm [Hol22] essentially implements the SulQ k-means algorithm [Kob21] and

also includes parts of the k-means++ algorithm. An implementation for this is included in the differential privacy library for Python [Lev19].
The following data sets were used for comparison: the combined MNIST data set [Haf98] with $70.000$ data points and $784$ dimensions, and synthetic data sets, created using the $make\_blobs$ algorithm from the sklearn.datasets library [PVG+11]. The synthetic data sets are always created with $100000$ data points, a bounding box of $(0, 100)$, a standard deviation of $1$ and, unless otherwise specified, a dimensions of $100$.
Further information about the comparisons is in the captions of the following figures.

Note that our Python implementation [Edl23] contains the same bug in the Permute-and-Flip algorithm as in the exponential mechanism, as described in Section 4.4. As described in Section 5.3 and Section 5.4, this bug does not affect the performance of the algorithm.

### Privacy Utility Tradeoff

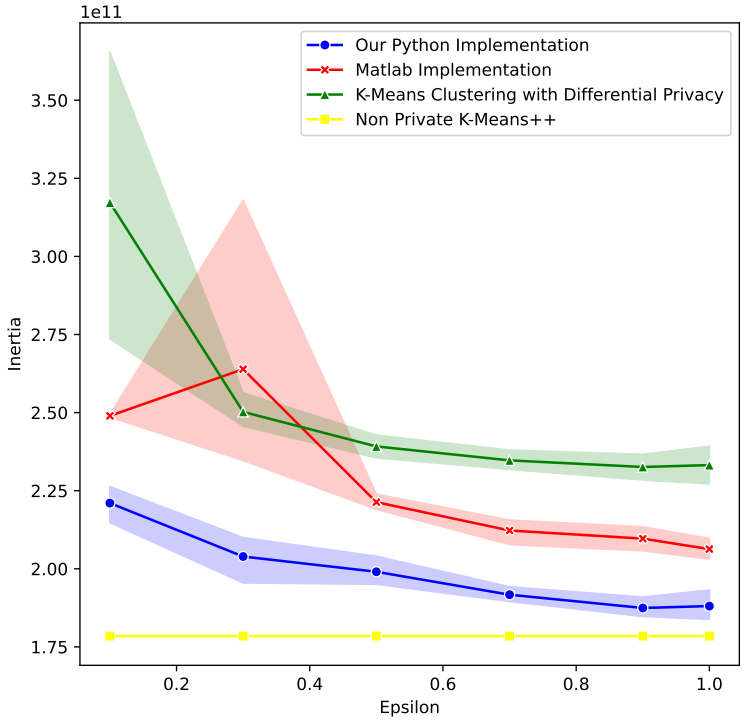


Figure 5.4: Comparison of the Matlab Implementation [Hon17b], our Python Implementation [Edl23], the k-means clustering with differential privacy algorithm [Hol22], and the non-private k-means++ algorithm [PVG+11] in terms of the effect of $\epsilon$ on inertia for the combined MNIST data set. Configurations for Python: data=MNIST data set, $k = 10, \epsilon = Epsilon, \delta = 0.1, r = 255$. Configurations for Matlab: data=MNIST data set, $n = 70000, d = 784, k = 10, \epsilon = Epsilon, delta = 0.1, range = 255 \cdot \sqrt{784}, side\_length = 510$. Configurations for the k-means clustering with differential privacy: X = MNIST data set, $n\_clusters = 10, bounds = (0, 255), \epsilon = Epsilon$. Configurations for the non private k-means++ algorithm: data = MNIST data set, $n\_clusters = 10, n\_init = 1$. For each Epsilon five independent runs were made and the respective average of inertia was calculated.

### Effect of Number of Cluster searched for $k$

- MNIST data set

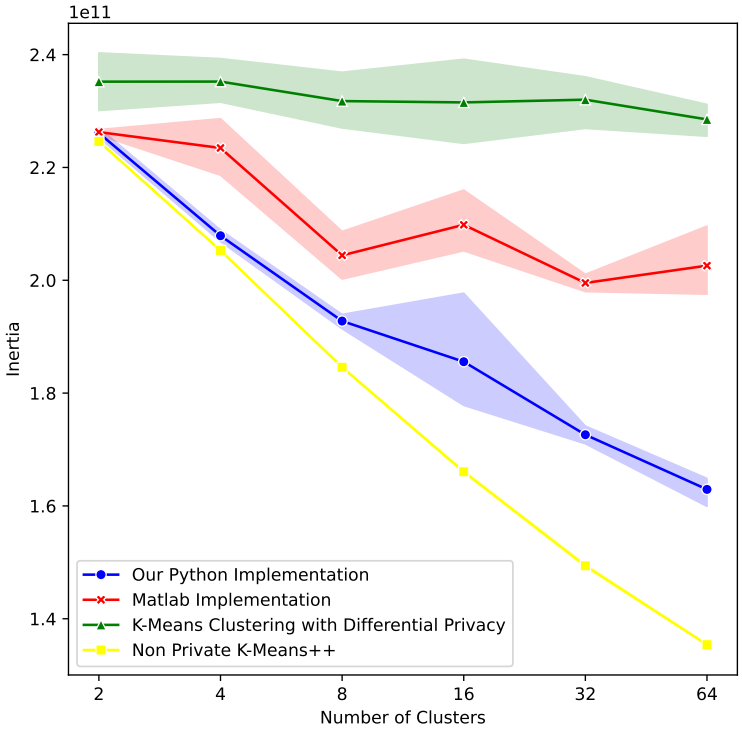


Figure 5.5: Comparison of the Matlab Implementation [Hon17b], our Python Implementation [Edl23], the k-means clustering with differential privacy algorithm [Hol22], and the non-private k-means++ algorithm [PVG+11] in terms of the effect of the Number of Clusters searched for on inertia for the combined MNIST data set. Configurations for Python: data=MNIST data set, $k$ = Number of Clusters, $\epsilon = 1.0, \delta = 0.1, r = 255$. Configurations for Matlab: data=MNIST data set, $n = 70000, d = 784, k$ = Number of Clusters, $\epsilon = 1.0, delta = 0.1, range = 255 \cdot \sqrt{784}, side\_length = 510$. Configurations for k-means clustering with differential privacy: X = MNIST data set, $n\_clusters$ = Number of Clusters , $bounds = (0, 255), \epsilon = 1.0$. Configurations for non private k-means++ algorithm: data = MNIST data set, $n\_clusters$ =Number of Clusters, $n\_init = 1$. For each Number of Cluster five independent runs were made and the respective average of inertia was calculated.

- Synthetic data set

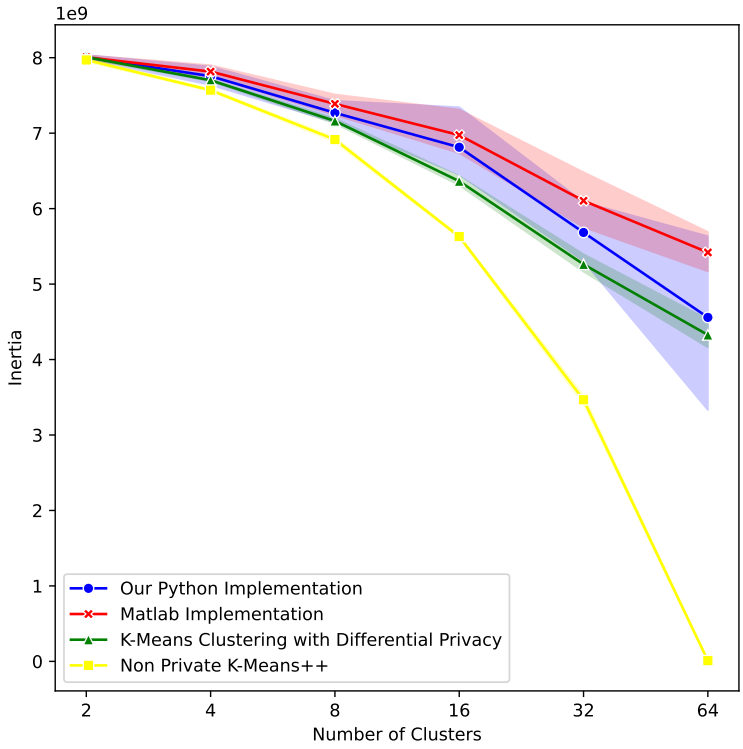


Figure 5.6: Comparison of the Matlab Implementation [Hon17b], our Python Implementation [Edl23], the k-means clustering with differential privacy algorithm [Hol22], and the non-private k-means++ algorithm [PVG+11] in terms of the effect of the Number of Clusters searched for on inertia for a synthetic data set. The synthetic data set is created with $64$ clusters. Configurations for Python: data=Synthetic data set, $k$ = Number of Clusters, $\epsilon = 1.0, \delta = 0.1, r = 110$. Configurations for Matlab: data=Synthetic data set, $n = 100000, d = 100, k =$ Number of Clusters, $\epsilon = 1.0, delta = 0.1, range = 110 \cdot \sqrt{100}, side\_length = 220$. Configurations for k-means clustering with differential privacy: X = Synthetic data set, $n\_clusters =$ Number of Clusters , $bounds = (-5, 105), \epsilon = 1.0$. Configurations for non private k-means++ algorithm: data = Synthetic data set, $n\_clusters =$ Number of Clusters, $n\_init = 1$. For each Number of Clusters five independent runs were made and the respective average of inertia was calculated.

### Effect of Dimension

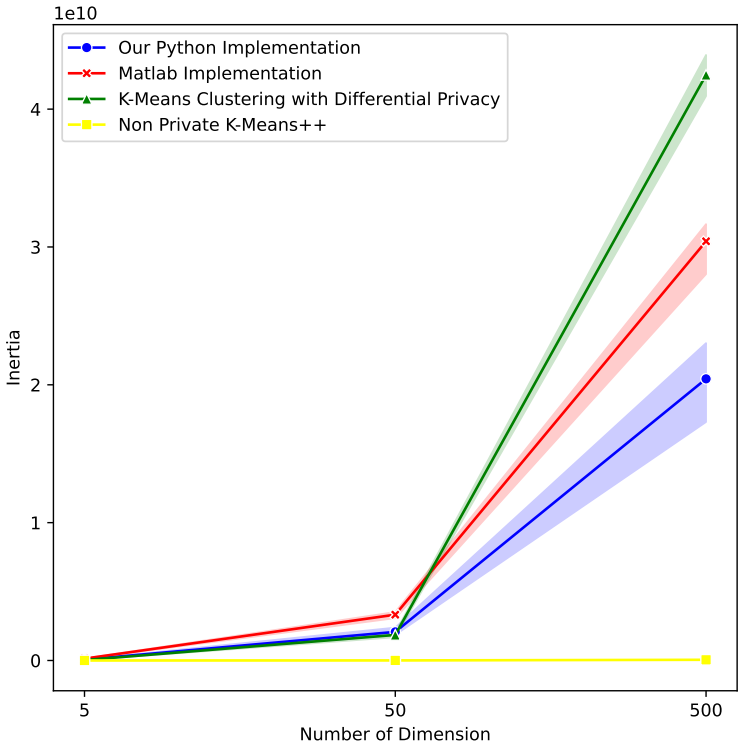


Figure 5.7: Comparison of the Matlab Implementation [Hon17b], our Python Implementation [Edl23], the k-means clustering with differential privacy algorithm [Hol22], and the non-private k-means++ algorithm [PVG+11] in terms of the effect of dimensions in the synthetic data set on inertia. For each $Dimension$, a synthetic data set is created with that number of dimensions and 32 clusters. Configurations for Python: data=synthetic data set with dimension $Dimension$, $k = 32, \epsilon = 0.5, \delta = 0.1, r = 110$. Configurations for Matlab: data=synthetic data set with dimension $Dimension$, $n = 100000, d = Dimension, k = 32, \epsilon = 0.5, delta = 0.1, range = 110 \cdot \sqrt{100}, side\_length = 220$. Configurations for k-means clustering with differential privacy: X = synthetic data set with dimension $Dimension$, $n\_clusters = 32, bounds = (-5, 105), \epsilon = 1.0$. Configurations for non private k-means++ algorithm: data = synthetic data set with dimension $Dimension$, $n\_clusters = 32, n\_init = 1$. For each synthetic data set with dimension $Dimension$, five independent runs were made and the respective average of inertia was calculated.

### Effect of Number of Intrinsic Clusters

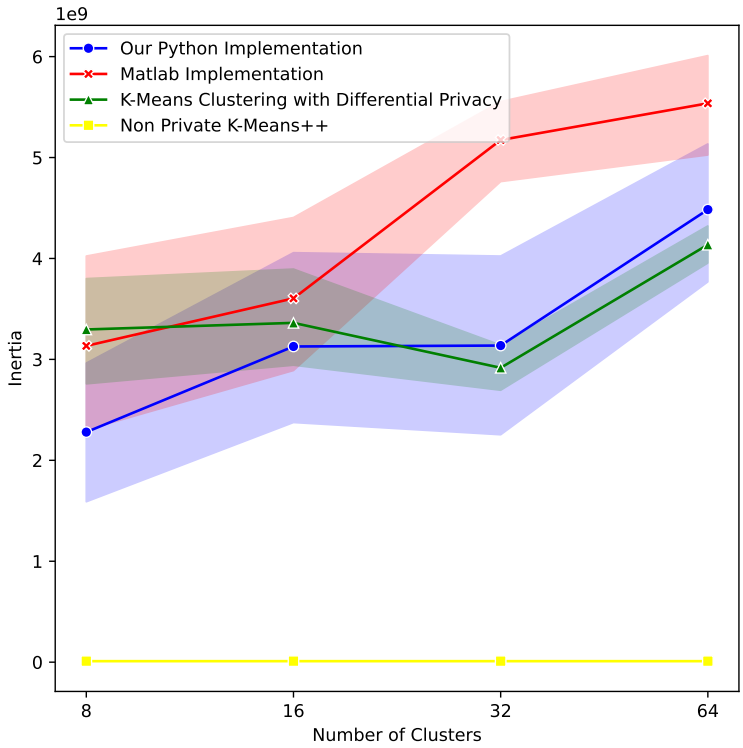


Figure 5.8: Comparison of the Matlab Implementation [Hon17b], our Python Implementation [Edl23], the k-means clustering with differential privacy algorithm [Hol22], and the non-private k-means++ algorithm [PVG+11] in terms of the effect of intrinsic clusters on inertia. For each Number of Clusters, a synthetic data set is created with that number of clusters. Afterwards, the algorithms are executed with these data sets, and $k$ is set to the number of clusters in the current data set. Configurations for Python: data = synthetic data set, $k$ = Number of Clusters,$\epsilon = 1.0, \delta = 0.1, r = 110$. Configurations for Matlab: data = synthetic data set, $n = 100000, d = 100, k$ =Number of Clusters, $\epsilon = 1.0, delta = 0.1, range = 110 \cdot \sqrt{100}, side\_length = 220$. Configurations for k-means clustering with differential privacy: X = synthetic data set, $n\_clusters$ = Number of Clusters, $bounds = (-5, 105), \epsilon = 1.0$. Configurations for non private k-means++ algorithm: data = synthetic data set, $n\_clusters$ =Number of Clusters, $n\_init = 1$. For each data set five independent runs were made and the respective average of inertia was calculated.

The non-private K-Means++ algorithm [Vas07] always outperforms all other algorithms, especially on synthetic data sets. Also as the number of intrinsic clusters increases, the non-private algorithm performs significantly better, suggesting that the other algorithms are not as good at finding many clusters. Furthermore, on the MNIST data set, our Python algorithm is not much worse than the non-private algorithm and outperforms the other two differentially private algorithms. On synthetic data sets, our algorithm and the differentially private k-means algorithm perform quite similar, although our algorithm handles high-dimensional data better. Moreover, our algorithm always outperforms the Matlab implementation, indicating that our modifications are indeed an improvement.

In summary, we were able to improve the performance of the algorithm. We left the private k-means part in the algorithm since it improves the performance of the algorithm, and we improved the algorithm by finding a better performing privacy budget distribution. The iteration parameters are already set for all algorithms to achieve the best performance, so we leave all iteration parameters as they were. We further improved the performance by adding more cube centers by reducing the $\gamma$ value in `private_partition`, and by improving the candidate set by selecting less but more accurate cube centers. Replacing the exponential mechanism with the Permute-and-Flip mechanism is an improvement in theory, but has no impact on the performance in practice.

# 6 Conclusion

In this work, we implemented a differentially private clustering algorithm that also works in high-dimensional Euclidean spaces in Python, based on the algorithm of Balcan et al. [Hon17a]. The algorithm works well in theory, but in practice many problems occur, so that the algorithm cannot be executed efficiently because the runtime is too high and the memory usage is too high. We lowered some parameters of the algorithm to achieve at least acceptable results, runtime and memory usage.
Furthermore, the implementation of differentially private clustering algorithms is hard and often leads to bugs. We located several bugs in the Matlab implementation of Balcan et al. [Hon17b] and corrected them in their implementation. We also made one bug in our implementation of the `localswap` algorithm, but it does not affect the performance of the algorithms. After applying our corrections to the Matlab implementation, we compared our Python implementation [Edl23] to the Matlab implementation and showed that they produce similar results on different data sets, indicating that they both work the same.
Our goal was then to further improve the clustering results by examining different modifications of the algorithm while preserving differential privacy and acceptable runtime and memory usage. We showed that investing a part of the privacy budget in the additional private k-means algorithm after the `recover` algorithm improves the results when using the MNIST data set, but this does not hold for synthetic data sets. We examined the effect of different iterations $T$ of the loops in the sub-algorithms and found that increasing or decreasing them does not improve the results. In fact, it causes more runtime and worsens the results when iterations are increased, since the privacy budget must be divided to each iteration. We found that the current implementation of the `localswap` algorithm does not affect the results when using the MNIST data set because the probabilities for selecting swaps and cluster center sets are too similar, resulting in random selections. Even if a large privacy budget is invested in `localswap`, resulting in an overall privacy budget that is too high due to privacy considerations, the algorithm would still behave like a random selection. To improve the `localswap` algorithm, we replaced the exponential mechanisms with the Permute-and-Flip mechanism [Dan20] in all private selections to achieve better probabilities for the private selections, which is an improvement in theory but did not affect the result in practice. The probabilities are still all the same, and thus the algorithm still behaves like a random selection, even if a larger privacy budget is invested, resulting in an overall privacy budget that is too high due to privacy considerations.

In addition, we performed a hyperparameter optimization of the privacy budget distribution to examine in which sub-algorithm how much of the privacy budget should be invested to achieve the best results when using the MNIST data set. This leads to a privacy budget distribution that improves the algorithm, and we found that the privacy budget distribution used by Balcan et al. is similar to the one we found to be best on average. Subsequently, we improve the algorithm by making two additional modifications that preserve differential privacy. We reduced the bound that determines whether a new cube center is included in the candidate set and a new cube is partitioned in the `private_partition` algorithm to obtain more and more accurate cube centers as possible cluster centers. In the `private_partition` algorithm, we reduced the candidate set only by the cube centers computed at greater depths to obtain only cube centers in the candidate set that are a more accurate representation of the clusters.
At last, we compared our improved Python implementation [Edl23] to a non-private k-means++ algorithm [Vas07], the Matlab implementation [Hon17b], and a differentially private k-means algorithm [Eri21]. Our algorithm always outperforms the Matlab implementation, indicating that the modifications are indeed an improvement. Using the MNIST data set, our algorithm performs better than the differentially private k-means algorithm, but on synthetic data sets, the differentially private k-means algorithm performs equally. The non-private k-means++ algorithm performs always better, leading to the conclusion that our improved algorithm still does not provide the best possible clustering results and could be further improved.

## 6.1 Future Work

Since our algorithm [Edl23] still does not perform as well as the non-private algorithm [Vas07], our algorithm could be further improved. In particular, private selections in the `localswap` algorithm could be further improved, as it still behaves like a random selection.
Furthermore, the runtime is still very high compared to the other algorithms used in the comparisons. Especially, the `localswap` algorithm has a long runtime because many different swaps need to be calculated. One solution would be to reduce the number of swaps to be calculated, e.g., by selecting a subset of the candidate set used for the possible swaps.
Another problem is that the range of the data space must be specified in the algorithm's input, which is often much too large to cover the worst case, resulting in too much noise being added. Finding privately the range of the data space is still an open problem in computer science.

# References


[Aar14] Cynthia Dwork, Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science Vol. 9, Nos. 3–4 (2014) 211–407*, 2014.

[Dan20] Ryan McKenna, Daniel Sheldon. Permute-and-flip: A new mechanism for differentially private selection. *34th Conference on Neural Information Processing Systems (NeurIPS 2020)*, 2020.

[Dwo06] C. Dwork. Differential privacy. *in Proceedings of the 33rd International Conference on Automata, Languages and Programming (ICALP)*, 2006.

[Edl23] Andre Edler. Implementation. *LINK*, 2023.

[Eri21] Huy L. Nguy ên, Anamay Chaturvedi, Eric Z Xu. Differentially private k-means via exponential mechanism and max cover. *https://ojs.aaai.org/index.php/AAAI/article/view/17099*, 2021.

[Haf98] LeCun Y, Bottou L, Bengio Y, Haffner P. Gradient based learning applied to document recognition. *In Proceedings of the IEEE*, 1998.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[Hol22] Naoise Holohan. Implementation. *https://github.com/IBM/differential-privacy-library/blob/main/diffprivlib/models/*$k_means.py, 2019 - 2022$.

[Hon17a] Maria-Florina Balcan, Travis Dick, Yingyu Liang, Wenlong Mou, Hongyang Zhang. Differentially Private Clustering in High-Dimensional Euclidean Spaces. *Proceedings of the 34th International Conference on Machine Learning, PMLR 70:322-331*, 2017.

[Hon17b] Maria-Florina Balcan, Travis Dick, Yingyu Liang, Wenlong Mou, Hongyang Zhang. Implementation. *https://github.com/mouwenlong/dp-clustering-icml17*, 2017.

[Jin10] Su Dong, Cao Jianneng, Li Ninghui, Bertino Elisa, Jin, Hongxia,. Differentially private combinatorial optimization. *In ACM-SIAM symposium on Discrete Algorithms, pp. 1106–1125*, 2010.

[Kap09] Feldman Dan, Fiat Amos, Kaplan Haim,Nissim Kobbi. Private coresets. *In ACM Symposium on Theory of Computing, pp. 361–370*, 2009.

[Kob21] Avrim Blum , Cynthia Dwork, Frank McSherry, Kobbi Nissim. Practical privacy: The sulq framework. *ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 128–138*, 2021.

[Lev19] Holohan Naoise, Braghin Stefano, Mac Aonghusa, Levacher Killian. Diffprivlib: the IBM differential privacy library. *ArXiv e-prints*, 1907.02444 [cs.CR], July 2019.

[Nie16] Nock Richard, Canyasse Rapha el, Boreli Roksana, Nielsen Frank. k-variates++: more pluses in the k-means++. *arXiv preprint arXiv:1602.01198*, 2016.

[PVG+11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[Sin15] Wang Yining, Wang Yu-Xiang, Singh Aarti,. Differentially private subspace clustering. *In Advances in Neural Information Processing Systems, pp. 1000–1008*, 2015.

[Smi07] Nissim Kobbi, Raskhodnikova Sofy, Boreli Roksana, Smith, Adam. Smooth sensitivity and sampling in private data analysis. *In ACM Symposium on Theory of Computing, pp. 75–84*, 2007.

[Tal16] Gupta Anupam, Ligett Katrina, McSherry Frank, Roth Aaron, Talwar Kunal. Differentially private combinatorial optimization. *In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, CODASPY '16*, 2016.

[Vas07] Arthur David, Vassilvitskii Sergei. k-means++: The advantages of careful seeding. *ACM-SIAM Symposium on Discrete Algorithms, pp. 1027–1035*, 2007.

# A Pseudocode

### All Modifications Applied

**Algorithm 8:** Private Clustering

**input** data: $X$ ($d \times n$-Matrix), number of clusters: $k$, privacy parameter: $\epsilon$, failure probability: $\delta$, range of data: $r$
**output** Cluster centers: $Z$
$low\_dim = \frac{\log n}{2}$
$T = 1$
$Cluster\_Centers = []$
**for** *t in range(T)* **do**
    $data\_low\_dim$ = JL-Transform($X$, $low\_dim$) $\cdot \frac{1}{\sqrt{d}}$
    $candidates$ = candidate($data\_low\_dim$, $k$, $\frac{\epsilon}{T} \cdot 0.7$, $\delta$, $r$)
    $centers\_low\_dim$ = localswap($candidates$, $data\_low\_dim$, $k$, $\frac{\epsilon}{T} \cdot 0.01$, $\delta$, $r \cdot \sqrt{d}$)
    $centers\_high\_dim$ = recover($centers\_low\_dim$, $data\_low\_dim$, $X$, $k$, $\frac{\epsilon}{T} \cdot 0.04$, $r$)
    $iter = 3$
    **for** *i in range(iter)* **do**
        $centers\_high\_dim$ = priv_kmean($centers\_high\_dim$, $X$, $k$, $\frac{\epsilon}{iter \cdot T} \cdot 0.2$, $r$)
    $Cluster\_Centers$.append( ($centers\_high\_dim$) )
Choose a set of cluster centers $Z$ from $Cluster\_Centers$ with the Permute and Flip mechanism ($epsilon = \frac{\epsilon}{T} \cdot 0.05, sensitivity = 2 \cdot (r \cdot \sqrt{d})^2, utility =$clustering losses from each $Cluster\_Centers(t)$)
**return** $Z$

**Algorithm 9:** Candidate

**input** low dimensional data: $Y$ ($p \times n$-Matrix), number of clusters: $k$, privacy parameter: $\epsilon$, failure probability: $\delta$, range of data: $r$
**output** Candidate Set: $Cand$
$Cand$ = []
$T = 3$
**for** *t in range(T)* **do**
    $cube\_initial$ = Cube()
    **for** *i in range(p)* **do**
        $random$ = random_uniform($-r, r$)
        $random\_bounds$ = [$-r$ + $random$, $r$ + $random$]
        $cube\_initial$.add_dimension($random\_bounds$)
    $C$ = private_partition($Y$, $k$, $\frac{\epsilon}{T}$, $\frac{\delta}{T}$, $cube\_initial$)
    $Cand$.append($C$)
**return** $Cand$

**Algorithm 10:** Private Partition

**input** low dimensional data: $Y$ ($p \times n$-Matrix), number of clusters: $k$, privacy parameter: $\epsilon$, failure probability: $\delta$, $cube\_initial$
**output** Cube Centers: $C$
$depth = 0$
$\epsilon' = \frac{\epsilon}{2 \cdot \log n}$
$\gamma = \frac{1}{\epsilon'} \cdot \log \frac{n}{\delta}$
$C = []$
$active\_cubes = [cube\_initial]$
**while** $depth \leq \log n$ *and* $active\_cubes \neq \emptyset$ **do**
    $depth = depth + 1$
    $Cubes\_Next\_Depth = []$
    **for** $cube$ *in* $active\_cubes$ **do**
        $C$.append($cube.center$)
        $active\_cubes$.remove($cube$)
        $new\_Cubes = cube$.partition()
        **for** $q$ *in* $new\_Cubes$ **do**
            $num\_data\_points = q$.getNumberOfDatapointsInCube()
            **if** $num\_data\_points \leq \gamma$ **then**
                $prob = \frac{1}{2} \cdot \exp(-\epsilon' \cdot (\gamma - num\_data\_points))$
            **else**
                $prob = 1 - \frac{1}{2} \cdot \exp(\epsilon' \cdot (\gamma - num\_data\_points))$
            Append $q$ with probability $prob$ to $Cubes\_Next\_Depth$
        $active\_cubes = Cubes\_Next\_Depth$
Assign $C'$ as the last cube centers appended to $C$ until the $depth$ where the number of cube centers in $C'$ reaches $k$, including the cube centers from that $depth$.
**return** $C'$

**Algorithm 11:** Localswap

**input** $candidates$, low dimensional data: $Y$ ($p \times n$-Matrix), number of clusters: $k$, privacy parameter: $\epsilon$, failure probability: $\delta$, range of data: $r$
**output** Cluster Centers (low dimensional): $centers\_low\_dim$
$Centers\_Init$ = uniformly choose $k$ centers from $candidates$
$list\_Centers$ = [ ($Centers\_Init$) ]
$T = k$
**if** $T > 20$ **then**
    $T = 20$
**for** $t$ *in range*$(1, T + 1)$ **do**
    $current\_Centers$ = $list\_Centers[t - 1]$
    $loss\_before\_swap$ = compute_loss($Y$, $current\_Centers$)
    **for** $x$ *in* $current\_Centers$ **do**
        **for** $y$ *in* $candidates \setminus current\_Centers$ **do**
            $loss\_after\_swap$ = compute_loss($Y$, $current\_Centers \cup x \setminus y$)
            Save $loss\_after\_swap - loss\_before\_swap$ in list $util$
    Select swap $(x, y)$ with the Permute and Flip mechanism $(epsilon = \frac{\epsilon}{2 \cdot (T+1)}, sensitivity = 2 \cdot r^2, utility = util)$
    $list\_Centers$.append($(current\_Centers \cup x \setminus y)$)
Select a set of cluster centers $list\_Centers(t)$ from $list\_Centers$ using the Permute-and-Flip mechanism $(epsilon = \frac{\epsilon}{2 \cdot (T+1)}, sensitivity = 2 \cdot r^2, utility =$clustering losses of each $list\_Centers(t))$
**return** $list\_Centers(t)$

**Algorithm 12:** Recover

**input** low dimensional cluster centers: $centers\_low\_dim$, low dimensional data: $Y$ ($p \times n$-Matrix), data: $X$ ($d \times n$-Matrix), number of clusters: $k$, privacy parameter: $\epsilon$, range of data: $r$
**output** Cluster Centers (high dimensional): $centers\_high\_dim$
$centers\_high\_dim$ = []
**for** $index\_of\_cluster$ *in range(k)* **do**
    $indexes\_data\_points\_assigned\_current\_cluster$ = indexes of low dimensional data points ($Y$) that have the closest distance to current cluster center: $centers\_low\_dim[index\_of\_cluster]$
    $numberDP$ = len($indexes\_data\_points\_assigned\_current\_cluster$)
    $numberDP\_noised$ = $numberDP$ + Laplace($\frac{2}{\epsilon}$)
    **if** $numberDP\_plus\_noise < 1$ **then**
        $numberDP\_plus\_noise = 1$
    $sum\_data\_points$ = sum_up_data_points($indexes\_data\_points\_assigned\_current\_cluster$, $X$)
    $noise\_center$ = []
    **for** $dim$ *in range(d)* **do**
        $noise\_center$.append_dimension(Laplace($\frac{2 \cdot r}{\epsilon \cdot numberDP\_noised}$))
    $center\_noised = \frac{1}{numberDP} \cdot sum\_data\_points + noise\_center$
    $centers\_high\_dim$.append( $center\_noised$ )
**return** $centers\_high\_dim$

**Algorithm 13:** Private_kMeans

**input** current cluster centers: $cur\_centers$, data: $X$ ($d \times n$-Matrix), number of clusters: $k$, privacy parameter: $\epsilon$, range of data: $r$
**output** Cluster Centers: $centers$
$centers$ = []
**for** $index\_of\_cluster$ *in range(k)* **do**
    $indexes\_data\_points\_assigned\_current\_cluster$ = indexes of data points ($X$) that have the closest distance to current cluster center: $cur\_centers$[$index\_of\_cluster$]
    $numberDP$ = len($indexes\_data\_points\_assigned\_current\_cluster$)
    $numberDP\_noised$ = $numberDP$ + Laplace($\frac{2}{\epsilon}$)
    **if** $numberDP\_plus\_noise < 1$ **then**
        $numberDP\_plus\_noise$ = 1
    $sum\_data\_points$ = sum_up_data_points($indexes\_data\_points\_assigned\_current\_cluster$, $X$)
    $noise\_center$ = []
    **for** $dim$ *in range(d)* **do**
        $noise\_center$.append_dimension(Laplace($\frac{2 \cdot r}{\epsilon \cdot numberDP\_noised}$))
    $center\_noised$ = $\frac{1}{numberDP} \cdot sum\_data\_points + noise\_center$
    $centers$.append( $center\_noised$ )
**return** $centers$

# B Figure Values

## 1 Values for Figures in Chapter 4

### Privacy Utility Tradeoff

| $\epsilon$ | Python | Matlab |
|---|---|---|
| 0.1 | 231568487650 | 248963173926 |
| 0.3 | 224978598581 | 263901498535 |
| 0.5 | 203270076643 | 221330835951 |
| 0.7 | 199940004255 | 212261355766 |
| 0.9 | 197388929880 | 209660015117 |
| 1.0 | 197031999345 | 206306096831 |

Table 1: Average inertia values for algorithms in Figure 4.2.

### Effect of k, MNIST

| $k$ | Python | Matlab |
|---|---|---|
| 2 | 229908752308 | 226267729190 |
| 4 | 214539179527 | 223453774582 |
| 8 | 202237972133 | 204418293874 |
| 16 | 185050998029 | 209836134612 |
| 32 | 177315813153 | 199528782432 |
| 64 | 174151779470 | 202593541793 |

Table 2: Average inertia values for algorithms in Figure 4.3.

### Effect of k, Synthetic

| $k$ | Python | Matlab |
|---|---|---|
| 2 | 8058815037 | 8005867623 |
| 4 | 7987195488 | 7816157600 |
| 8 | 7540396005 | 7388525276 |
| 16 | 7179246320 | 6974402996 |
| 32 | 6706233552 | 6104716131 |
| 64 | 6372741268 | 5419700408 |

Table 3: Average inertia values for algorithms in Figure 4.4.

### Effect of Dimension

| Dimension | Python | Matlab |
|---|---|---|
| 5 | 120987757 | 133377059 |
| 50 | 3169665090 | 3323998453 |
| 500 | 30528423526 | 30413081091 |

Table 4: Average inertia values for algorithms in Figure 4.5.

### Effect of Number of Intrinsic Cluster

| $k$ | Python | Matlab |
|---|---|---|
| 8 | 4410860023 | 3133818767 |
| 16 | 3774715341 | 3524399818 |
| 32 | 6182634341 | 5173658114 |
| 64 | 6352288372 | 5537152860 |

Table 5: Average inertia values for algorithms in Figure 4.6.

# 2 Values for Figures in Chapter 5

## 2.1 Effect of Private k-means

| $\epsilon$ | With Additional Private k-means | Without Additional Private k-means |
|---|---|---|
| 0.1 | 231568487650 | 234630257696 |
| 0.3 | 224978598581 | 219171509207 |
| 0.5 | 203270076643 | 218632552820 |
| 0.7 | 199940004255 | 217826886291 |
| 0.9 | 197388929880 | 213504087247 |
| 1.0 | 197031999345 | 214528653150 |

Table 6: Average inertia values for algorithms in Figure 5.1.

| $k$ | With Additional Private k-means | Without Additional Private k-means |
|---|---|---|
| 8 | 4410860023 | 3133818767 |
| 16 | 3774715341 | 3524399818 |
| 32 | 6182634341 | 5173658114 |
| 64 | 6352288372 | 5537152860 |

Table 7: Average inertia values for algorithms in Figure 5.2.

## 2.2 Hyperparameter Optimization

Table 8: Results of inertia with different privacy budget distributions. Algorithm was executed with the following configurations: data = MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$.

| Privacy Budget Distribution ($\epsilon\_candidates, \epsilon\_localswap, \epsilon\_recover, \epsilon\_kmean, \epsilon\_cluster\_selection$) | Average Inertia of 5 runs |
|---|---|
| (0.025, 0.025, 0.025, 0.025, 0.9) | 221494825919 |
| (0.025, 0.025, 0.025, 0.9, 0.025) | 220180054084 |
| (0.025, 0.025, 0.9, 0.025, 0.025) | 218939876144 |
| (0.025, 0.9, 0.025, 0.025, 0.025) | 218892144833 |
| (0.2, 0.15, 0.2, 0.4, 0.05) | 198577924821 |
| (0.2, 0.2, 0.15, 0.4, 0.05) | 204451801163 |
| (0.2, 0.2, 0.4, 0.15, 0.05) | 203827994802 |
| (0.3, 0.15, 0.2, 0.3, 0.05) | 197695862459 |
| (0.3, 0.2, 0.15, 0.3, 0.05) | 199555466986 |

| | |
|---|---|
| (0.3, 0.2, 0.3, 0.15, 0.05) | 202081990049 |
| (0.4, 0.05, 0.1, 0.4, 0.05) | 192778858843 |
| (0.4, 0.05, 0.2, 0.3, 0.05) | 195507959437 |
| (0.4, 0.05, 0.25, 0.25, 0.05) | 197435790690 |
| (0.4, 0.05, 0.3, 0.2, 0.05) | 195476219047 |
| (0.4, 0.1, 0.05, 0.4, 0.05) | 196073127166 |
| (0.4, 0.1, 0.15, 0.3, 0.05) | 199105227513 |
| (0.4, 0.1, 0.2, 0.25, 0.05) | 195780243586 |
| (0.4, 0.1, 0.25, 0.2, 0.05) | 194496966305 |
| (0.4, 0.1, 0.3, 0.15, 0.05) | 192680271849 |
| (0.4, 0.15, 0.1, 0.3, 0.05) | 197960104372 |
| (0.4, 0.15, 0.15, 0.25, 0.05) | 196024926608 |
| (0.4, 0.15, 0.2, 0.2, 0.05) | 197070767785 |
| (0.4, 0.15, 0.25, 0.15, 0.05) | 195506254901 |
| (0.4, 0.15, 0.3, 0.1, 0.05) | 196347787269 |
| (0.4, 0.2, 0.1, 0.25, 0.05) | 193410386123 |
| (0.4, 0.2, 0.2, 0.15, 0.05) | 199519716881 |
| (0.4, 0.2, 0.25, 0.1, 0.05) | 202515490660 |
| (0.4, 0.25, 0.05, 0.25, 0.05) | 195131010971 |
| (0.4, 0.25, 0.1, 0.2, 0.05) | 199558500194 |
| (0.4, 0.25, 0.15, 0.15, 0.05) | 196989171636 |
| (0.4, 0.25, 0.2, 0.1, 0.05) | 192554380834 |
| (0.4, 0.25, 0.25, 0.05, 0.05) | 193520722744 |
| (0.4, 0.3, 0.05, 0.2, 0.05) | 195125712712 |
| (0.4, 0.3, 0.1, 0.15, 0.05) | 195142570710 |
| (0.4, 0.3, 0.15, 0.1, 0.05) | 199051580127 |
| (0.4, 0.3, 0.2, 0.05, 0.05) | 195942839034 |
| (0.5, 0.05, 0.1, 0.3, 0.05) | 194229544430 |
| (0.5, 0.05, 0.15, 0.25, 0.05) | 194664810057 |
| (0.5, 0.05, 0.2, 0.2, 0.05) | 200190069185 |
| (0.5, 0.05, 0.25, 0.15, 0.05) | 193778596896 |
| (0.5, 0.05, 0.3, 0.1, 0.05) | 194482174341 |
| (0.5, 0.1, 0.05, 0.3, 0.05) | 199226219614 |
| (0.5, 0.1, 0.1, 0.25, 0.05) | 194760160715 |
| (0.5, 0.1, 0.15, 0.2, 0.05) | 195309784892 |
| (0.5, 0.1, 0.2, 0.15, 0.05) | 190613488357 |
| (0.5, 0.1, 0.25, 0.1, 0.05) | 196201690286 |

| | |
|---|---|
| (0.5, 0.1, 0.3, 0.05, 0.05) | 195599377809 |
| (0.5, 0.15, 0.05, 0.25, 0.05) | 197248207181 |
| (0.5, 0.15, 0.1, 0.2, 0.05) | 198357074260 |
| (0.5, 0.15, 0.15, 0.15, 0.05) | 192937371324 |
| (0.5, 0.15, 0.2, 0.1, 0.05) | 192137147519 |
| (0.5, 0.15, 0.25, 0.05, 0.05) | 194566789080 |
| (0.5, 0.2, 0.05, 0.2, 0.05) | 195711617466 |
| (0.5, 0.2, 0.1, 0.15, 0.05) | 194527524917 |
| (0.5, 0.2, 0.15, 0.1, 0.05) | 188960778185 |
| (0.5, 0.2, 0.2, 0.05, 0.05) | 196203915233 |
| (0.5, 0.25, 0.05, 0.15, 0.05) | 193639623674 |
| (0.5, 0.25, 0.1, 0.1, 0.05) | 194290531157 |
| (0.5, 0.25, 0.15, 0.05, 0.05) | 197530146762 |
| (0.5, 0.3, 0.05, 0.1, 0.05) | 193877199512 |
| (0.5, 0.3, 0.1, 0.05, 0.05) | 198547386974 |
| (0.6, 0.05, 0.05, 0.25, 0.05) | 194403421552 |
| (0.6, 0.05, 0.1, 0.2, 0.05) | 194355629782 |
| (0.6, 0.05, 0.15, 0.15, 0.05) | 199668653074 |
| (0.6, 0.05, 0.2, 0.1, 0.05) | 194439266492 |
| (0.6, 0.05, 0.25, 0.05, 0.05) | 196683014791 |
| (0.6, 0.1, 0.05, 0.2, 0.05) | 198158544133 |
| (0.6, 0.1, 0.1, 0.15, 0.05) | 190543168689 |
| (0.6, 0.1, 0.15, 0.1, 0.05) | 198349857974 |
| (0.6, 0.1, 0.2, 0.05, 0.05) | 192554078431 |
| (0.6, 0.15, 0.05, 0.15, 0.05) | 194644031977 |
| (0.6, 0.15, 0.1, 0.1, 0.05) | 200882189021 |
| (0.6, 0.15, 0.15, 0.05, 0.05) | 192180669492 |
| (0.6, 0.2, 0.05, 0.1, 0.05) | 195692992955 |
| (0.6, 0.2, 0.1, 0.05, 0.05) | 192583968130 |
| (0.6, 0.25, 0.05, 0.05, 0.05) | 196371143393 |
| (0.7, 0.001, 0.049, 0.2, 0.05) | 193613595055 |
| (0.7, 0.01, 0.04, 0.2, 0.05) | 194427446065 |
| (0.7, 0.0125, 0.225, 0.0125, 0.05) | 214171498090 |
| (0.7, 0.0125, 0.0125, 0.225, 0.05) | 199292091853 |
| (0.7, 0.0125, 0.0375, 0.2, 0.05) | 192488584530 |
| (0.7, 0.02, 0.03, 0.2, 0.05) | 194093797940 |
| (0.7, 0.025, 0.2, 0.025, 0.05) | 193994026113 |

| | |
|---|---|
| (0.7, 0.025, 0.025, 0.2, 0.05) | 190705907823 |
| (0.7, 0.03, 0.02, 0.2, 0.05) | 194498949759 |
| (0.7, 0.0375, 0.0125, 0.2, 0.05) | 201354280733 |
| (0.7, 0.04, 0.01, 0.2, 0.05) | 199900661121 |
| (0.7, 0.049, 0.001, 0.2, 0.05) | 211427964741 |
| (0.7, 0.05, 0.05, 0.15, 0.05) | 193043727579 |
| (0.7, 0.05, 0.1, 0.1, 0.05) | 191529358150 |
| (0.7, 0.05, 0.15, 0.05, 0.05) | 192881274591 |
| (0.7, $\frac{1}{12}$, $\frac{1}{12}$, $\frac{1}{12}$, 0.05) | 193523276360 |
| (0.7, 0.1, 0.05, 0.1, 0.05) | 193416906765 |
| (0.7, 0.1, 0.1, 0.05, 0.05) | 190454291557 |
| (0.7, 0.15, 0.05, 0.05, 0.05) | 200020493097 |
| (0.7, 0.2, 0.025, 0.025, 0.05) | 200448672192 |
| (0.7, 0.225, 0.0125, 0.0125, 0.05) | 215728287925 |
| (0.8, 0.0125, 0.125, 0.0125, 0.05) | 219478007478 |
| (0.8, 0.0125, 0.0125, 0.125, 0.05) | 198409087955 |
| (0.8, 0.025, 0.1, 0.025, 0.05) | 194081481067 |
| (0.8, 0.025, 0.025, 0.1, 0.05) | 195853488658 |
| (0.8, 0.1, 0.025, 0.025, 0.05) | 201672918696 |
| (0.8, 0.125, 0.0125, 0.0125, 0.05) | 207799282728 |
| (0.9, 0.025, 0.025, 0.025, 0.025) | 191873433481 |

| $\epsilon\_localswap$ | average inertia |
|---|---|
| 0.05 | 195032256996 |
| 0.1 | 195277573574 |
| 0.15 | 196134558134 |
| 0.2 | 195458487838 |
| 0.25 | 195509470151 |
| 0.3 | 196281214844 |

Table 9: Results of average inertia at different values of $\epsilon\_localswap$. Algorithm was executed with the following configurations: data = MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$.

| $\epsilon\_recover$ | average inertia |
|---|---|
| 0.05 | 195736498734 |
| 0.1 | 194888484949 |
| 0.15 | 195665711710 |
| 0.2 | 195209489397 |
| 0.25 | 196078368483 |
| 0.3 | 194917166063 |

Table 10: Results of average inertia at different values of $\epsilon\_recover$. Algorithm was executed with the following configurations: data = MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$.

| $\epsilon\_kmeans$ | average inertia |
|---|---|
| 0.05 | 195042674074 |
| 0.1 | 195328401170 |
| 0.15 | 194516726604 |
| 0.2 | 196710080523 |
| 0.25 | 195428773053 |
| 0.3 | 197205811073 |
| 0.4 | 194425993004 |

Table 11: Results of average inertia at different values of $\epsilon\_kmeans$. Algorithm was executed with the following configurations: data = MNIST data set, $k = 10, \epsilon = 1.0, \delta = 0.1, r = 255$.

### 2.3 Effect Higher $T$ in Candidate

Effect Higher T

| $T$ | average runtime in minutes | average inertia |
|---|---|---|
| 1 | 5.11 | 192208459625 |
| 2 | 6.86 | 191244311011 |
| 3 | 9.05 | 193384099925 |
| 4 | 11.84 | 192645002489 |
| 5 | 9.48 | 199962775079 |
| 6 | 9.73 | 209381147269 |
| 7 | 13.82 | 203654871763 |
| 8 | 14.95 | 198118186953 |
| 9 | 16.5 | 201056109812 |
| 10 | 19.04 | 201516542621 |
| 11 | 15.78 | 201834249369 |
| 12 | 18.59 | 200724551641 |
| 13 | 19.59 | 201488407549 |

Table 12: Average inertia and average runtime values for the algorithm in Figure 5.3.

### 2.4 Comparison To Other Algorithms

#### Privacy Utility Tradeoff

| $\epsilon$ | Python | Matlab | K-Means Clustering with DP | Non Private K-Means++ |
|---|---|---|---|---|
| 0.1 | 221037065061 | 248963173926 | 317215779099 | 178463293723 |
| 0.3 | 203925721070 | 263901498535 | 250231903382 | 178463293723 |
| 0.5 | 199048050348 | 221330835951 | 239169249654 | 178463293723 |
| 0.7 | 191706524737 | 212261355766 | 234709443271 | 178463293723 |
| 0.9 | 187457411535 | 209660015117 | 232589281559 | 178463293723 |
| 1.0 | 188063471126 | 206306096831 | 233192953324 | 178463293723 |

Table 13: Average inertia values for algorithms in Figure 5.4.

### Effect of k, MNIST

| $k$ | Python | Matlab | K-Means Clustering with DP | Non Private K-Means++ |
|---|---|---|---|---|
| 2 | 226099236236 | 226267729190 | 235210868892 | 224609799117 |
| 4 | 207895447679 | 223453774582 | 235221879849 | 205281882588 |
| 8 | 192771949453 | 204418293874 | 231765723939 | 184584690624 |
| 16 | 185561768924 | 209836134612 | 231530780146 | 166055532292 |
| 32 | 172620576451 | 199528782432 | 232016313229 | 149388540718 |
| 64 | 162929878236 | 202593541793 | 228517503111 | 135406472376 |

Table 14: Average inertia values for algorithms in Figure 5.5.

### Effect of k, Synthetic

| $k$ | Python | Matlab | K-Means Clustering with DP | Non Private K-Means++ |
|---|---|---|---|---|
| 2 | 8006593279 | 8005867623 | 8008008879 | 7970197410 |
| 4 | 7755163296 | 7816157600 | 7701395825 | 7569614479 |
| 8 | 7269464091 | 7388525276 | 7161761324 | 6915096690 |
| 16 | 6813257933 | 6974402996 | 6363514041 | 5629634742 |
| 32 | 5683927702 | 6104716131 | 5261762330 | 3467080527 |
| 64 | 4558354663 | 5419700408 | 4326059414 | 9988959 |

Table 15: Average inertia values for algorithms in Figure 5.6.

### Effect of Dimension

| Dimension | Python | Matlab | K-Means Clustering with DP | Non Private K-Means++ |
|---|---|---|---|---|
| 5 | 62619249 | 133377059 | 33242334 | 499224 |
| 50 | 2075071639 | 3323998453 | 1854134759 | 4994987 |
| 500 | 20433064121 | 30413081091 | 42481044641 | 49963357 |

Table 16: Average inertia values for algorithms in Figure 5.7.

### Effect of Number of Intrinsic Cluster

| $k$ | Python | Matlab | K-Means Clustering with DP | Non Private K-Means++ |
|---|---|---|---|---|
| 8 | 2279480211 | 3133818767 | 3296334163 | 9994533 |
| 16 | 3127961609 | 3524399818 | 3361823173 | 9993763 |
| 32 | 3137007212 | 5173658114 | 2916590268 | 9992256 |
| 64 | 4484112697 | 5537152860 | 4136291870 | 9988959 |

Table 17: Average inertia values for algorithms in Figure 5.8.