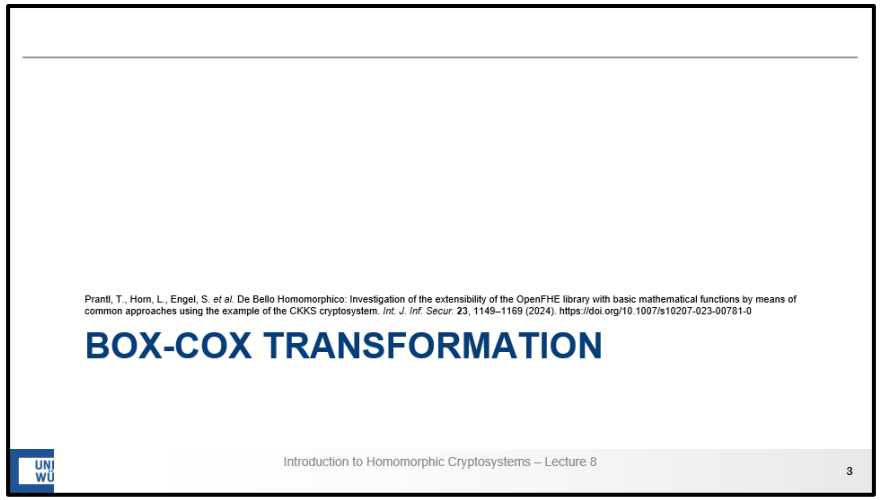
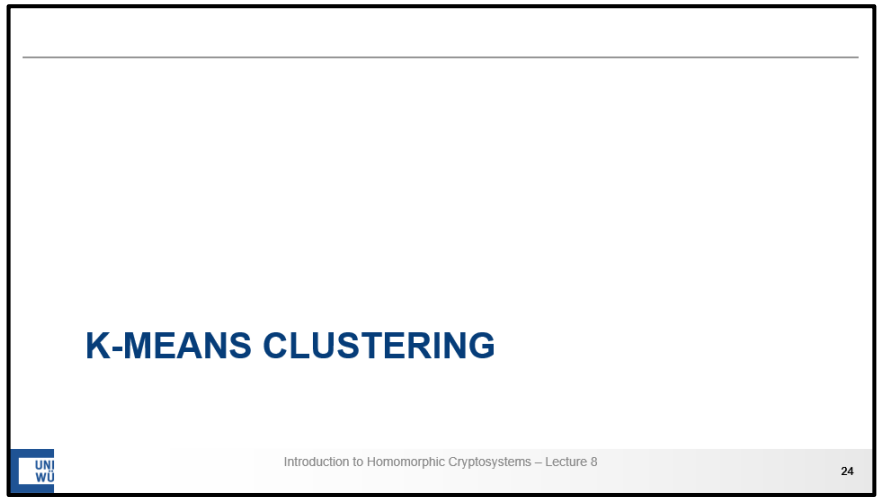
Homomorphic Applications

Introduction to Homomorphic Cryptosystems - Lecture 8

Advanced Homomorphic Applications



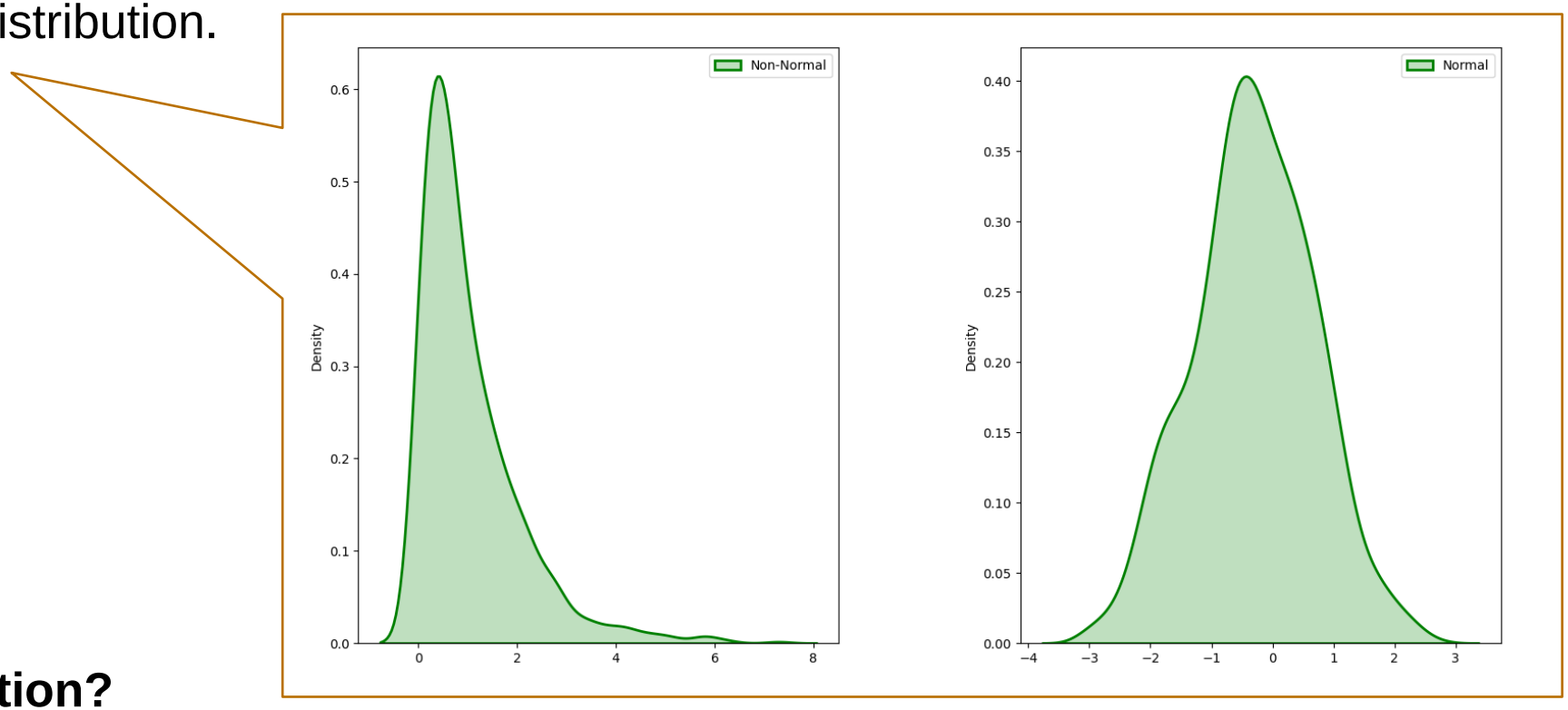


Prantl, T., Horn, L., Engel, S. *et al.* De Bello Homomorphico: Investigation of the extensibility of the OpenFHE library with basic mathematical functions by means of common approaches using the example of the CKKS cryptosystem. *Int. J. Inf. Secur.* **23**, 1149–1169 (2024). https://doi.org/10.1007/s10207-023-00781-0

BOX-COX TRANSFORMATION

Box-Cox Transformation

The **Box-Cox Transformation** is a statistics tool, that transform a non-normal distribution to a normal distribution.



Why do the transformation?

In many real-world scenarios, the data is not normally distributed. However, many statistical techniques assume a normal distribution. Transforming the data beforehand can improve the accuracy of other statistical analysis.

Box-Cox Transformation

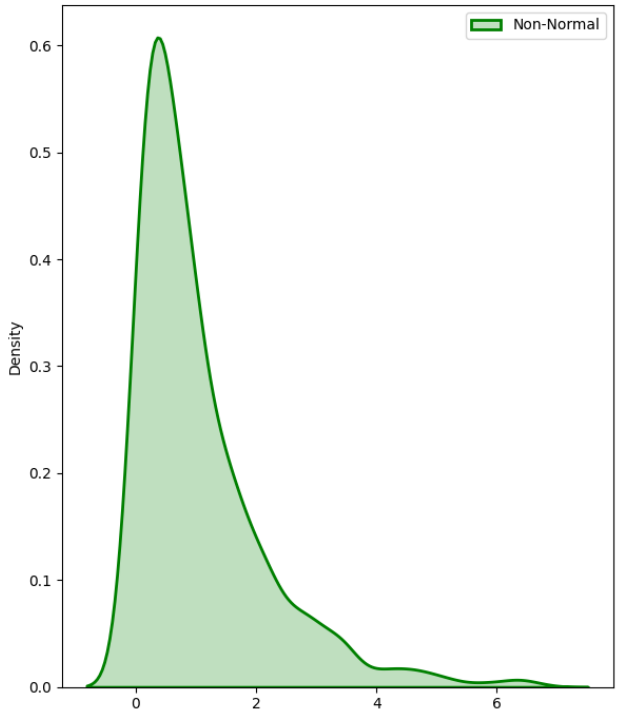
Mathematical Implementation
To transform the original data, we apply

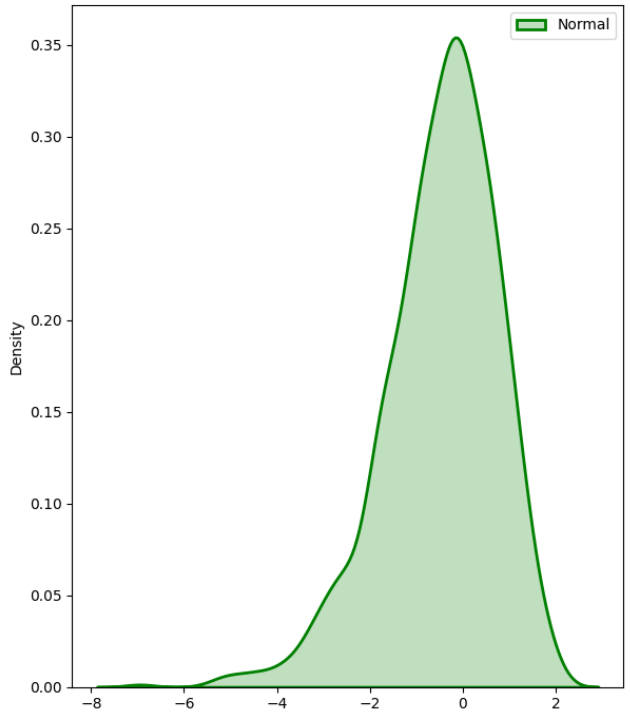
$$Y = \begin{cases} \frac{y^{\lambda} - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \log(y) & \text{else} \end{cases}$$

to the original data.

This seems to be straightforward (even homomorphically), but how about λ?
Choosing the right lambda is very important.

Using the wrong λ (0.05 in this case) does not yield a normal distribution:





Box-Cox Transformation

Finding the right λ

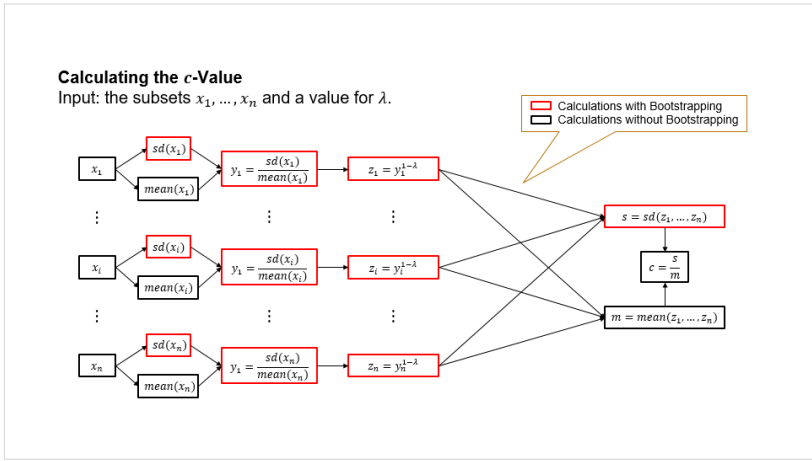
Guerrero's method suggests testing different values for λ and choose the value that has the lowest coefficient of variation c for the subsets of the original data.

Steps

1. Split the data into different subsets

$$x_1, \dots, x_n$$

2. Calculate the c-value for different λ-values
3. Choose the λ with the lowest c

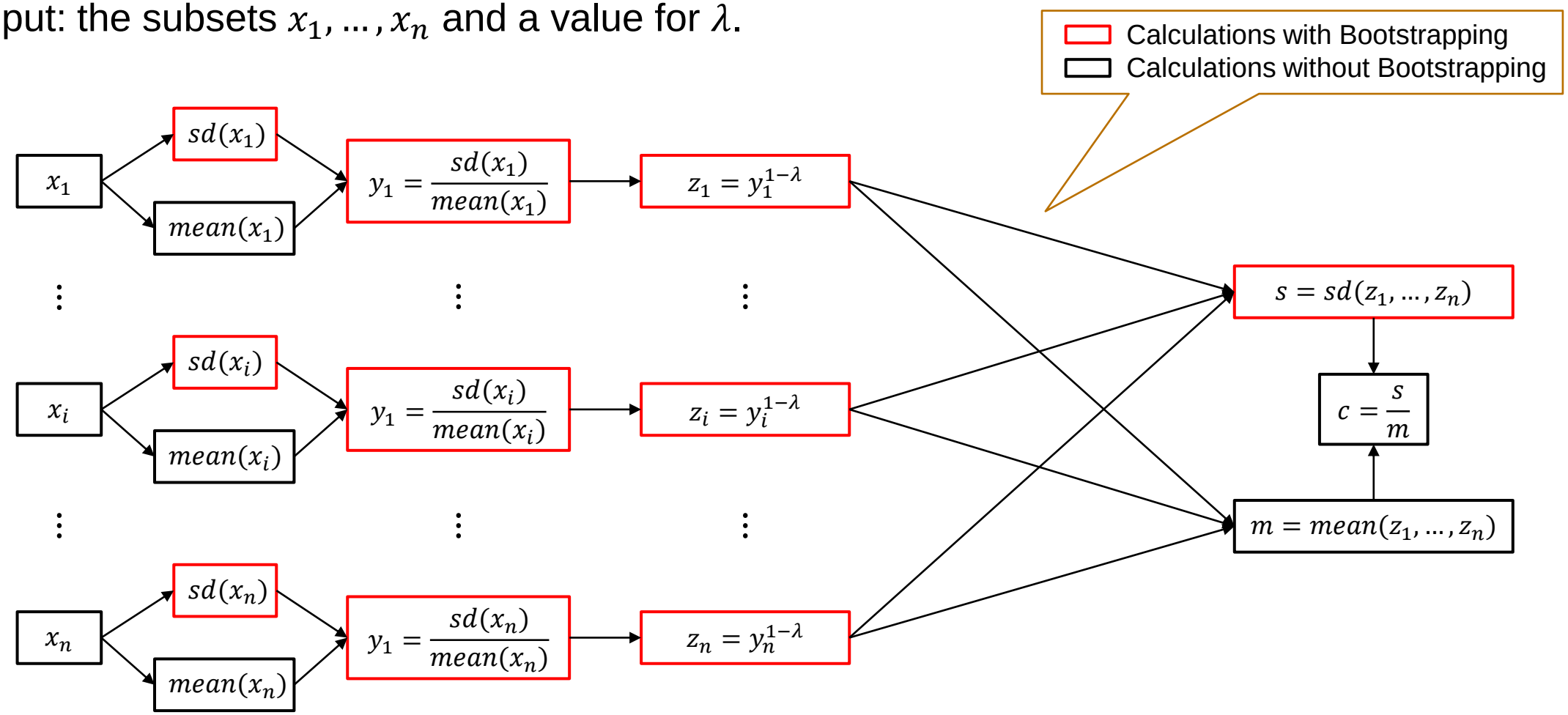


We already saw this last lecture (and in the exercise):

$$\lambda_{min} = 2 * \sum_{i=1}^{n} BS(c_{min} - c_i) * \lambda_i$$

Calculating the c-Value

Input: the subsets $x_1, \dots, x_n$ and a value for λ.



Box-Cox Transformation

Homomorphic Realization
We can do all the necessary calculations for the Box-Cox transformation homomorphically using the BS function and the other extend mathematical functions. However, we observe a big performance overhead (as expected).

Example
For this example, a dataset containing 168 values was chosen. The data represent the birthrates in New York per month. The authors implemented these steps and measured the performance:

1. The dataset was split into subsets of 12 values (representing one year) and encrypted.
2. Four different λ-values were chosen and the corresponding c-values c_1, c_2, c_3, c_4 were calculated
3. Using the BS function, the best λ was determined.
4. The Box-Cox transformation was applied to the original dataset

Box-Cox Transformation

Evaluation – Calculation Time

Step	Required time in seconds
c_1 Calculation	1806.48 ± 14.07
c_2 Calculation	1863.51 ± 17.97
c_3 Calculation	1570.91 ± 14.56
c_4 Calculation	1818.01 ± 24.97
λ Selection	6832.48 ± 9.23
Transformation Calculation	828.64 ± 2.41

Most expensive calculation
The most expensive step is selecting the λ. This is probably the easiest step in the unencrypted case.

Large calculation times
The whole operation needs 4.09 ± 0.01 hours when done homomorphically (non parallelized). Calculating the transformation the usual way, we only need 3.4 ± 0.01 milliseconds.

Box-Cox Transformation

Evaluation – Accuracy

λ	Non-homomorphically computed c-value	Homomorphically computed c-value
-1	0.16610	0.16379
0	0.14184	0.14068
1	0.15569	0.15566
2	0.19852	0.19841

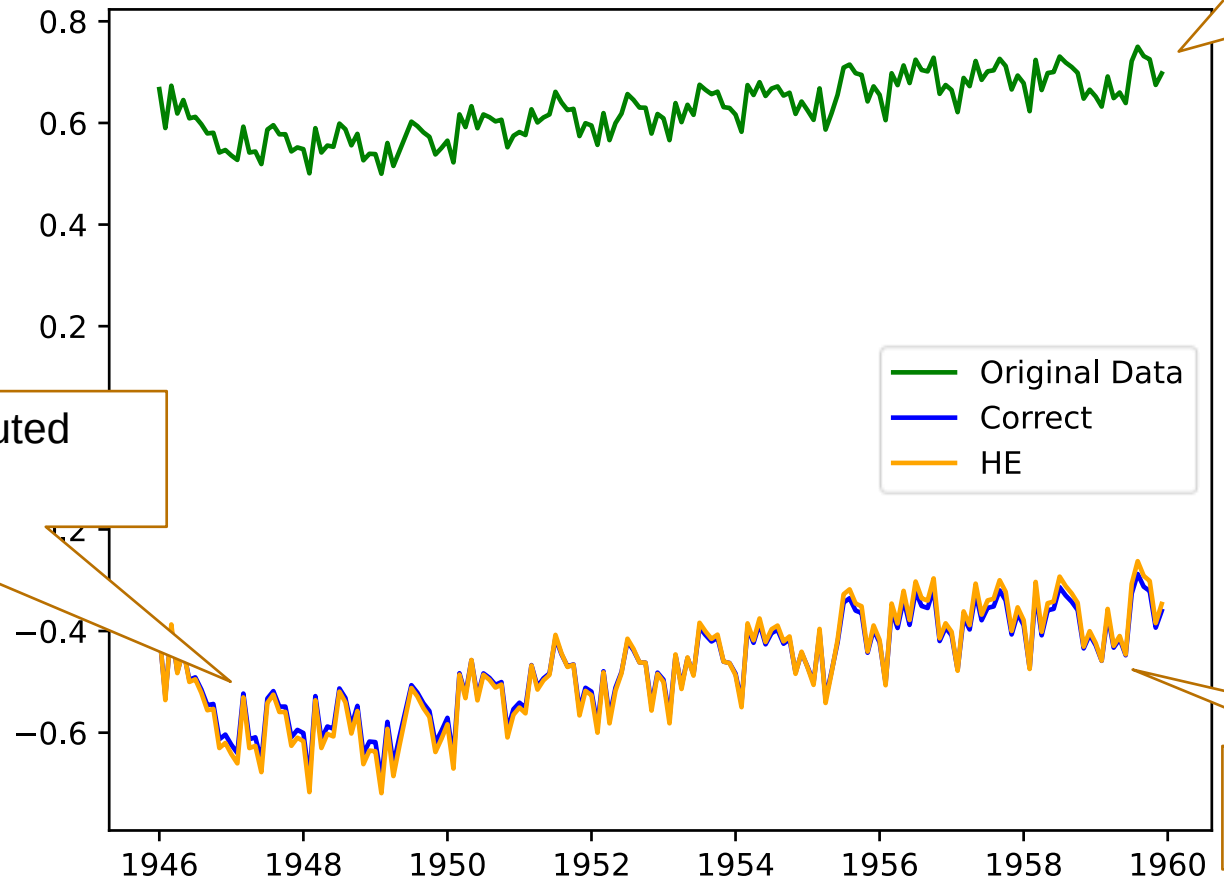
High Accuracy
Doing the operation homomorphically yields vary accurate c-values

$\lambda = 0$ should be the correct choice for the transformation.
But: When applying our formula for finding λ_{min} we get $\lambda_{min} = 0.1382$

Box-Cox Transformation

Evaluation – Results

Original data before transformation



Correct result ($\lambda = 0$), computed non-homomorphically

Result of the homomorphic computation.

Prantl, T. *et al.* (2023). Performance Impact Analysis of Homomorphic Encryption: A Case Study Using Linear Regression as an Example. In: Meng, W., Yan, Z., Piuri, V. (eds) Information Security Practice and Experience. ISPEC 2023. Lecture Notes in Computer Science, vol 14341. Springer, Singapore. https://doi.org/10.1007/978-981-99-7032-2_17

LINEAR REGRESSION

Linear Regression

Linear regression describes the problem to find a linear representation, that describes the relationship of a depended variable, and a set of independent variables (features).

$x_j = (\tilde{x}_1, \dots, \tilde{x}_m)^\perp$

We get a set of datapoints:

$$D = \{(y_1, x_1), (y_2, x_2), \dots, (y_n, x_n)\}$$

And we search for the best weight vector $\theta = (\theta_1, \dots, \theta_m)^\perp$, so that for every datapoint $(y_j, x_j) \in D$ this equation is optimized:

$$y_j = \langle \theta, x_j \rangle = \sum_{i=1}^{m} \theta_i \tilde{x}_i$$

But how do we do that?

Linear Regression

Gradient Descent

This is one method to determine θ.

It depends on a cost function (or loss function), which could be like this:

$$J(\theta) = \frac{1}{2}\left(\langle \theta, x_j \rangle - y_j\right)^2$$

This cost function describes (for a specific datapoint $(y_j, x_j) \in D$) how applying a concrete assignment of θ deviates from y.

We start with an initial random assignment of θ and then change the coefficients by using the derivation of the cost function:

$$\theta_j := \theta_j - \frac{\partial J(\theta)}{\partial \theta_j}$$

$$\frac{\partial J(\theta)}{\partial \theta_j} = \left(\langle \theta, x_j \rangle - y_j\right) * x_j$$

$$\theta_j := \theta_j - \left(\langle \theta, x_j \rangle - y_j\right) * x_j$$

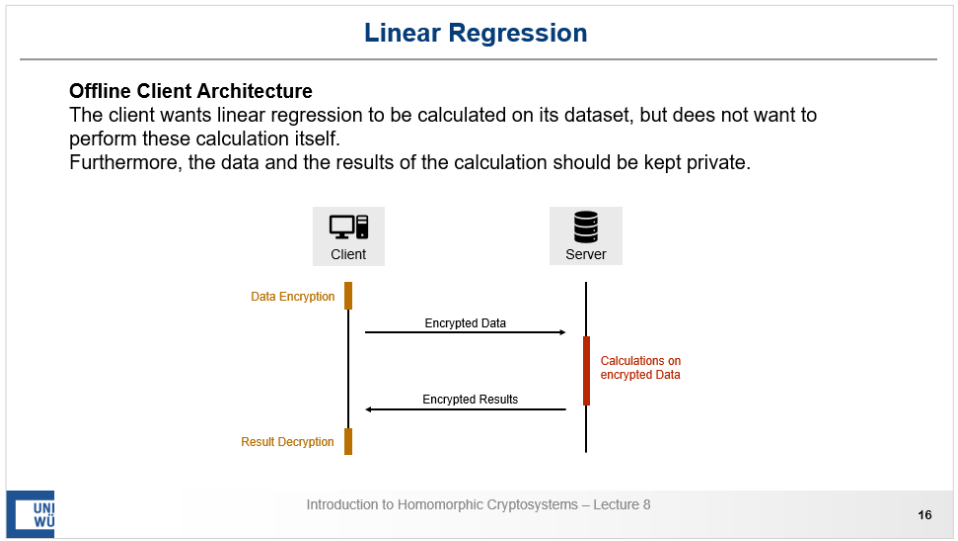
We can imagine this process as finding the sink in a complex graph. We utilize the gradient on our current position to find the right direction to walk to.

Linear Regression

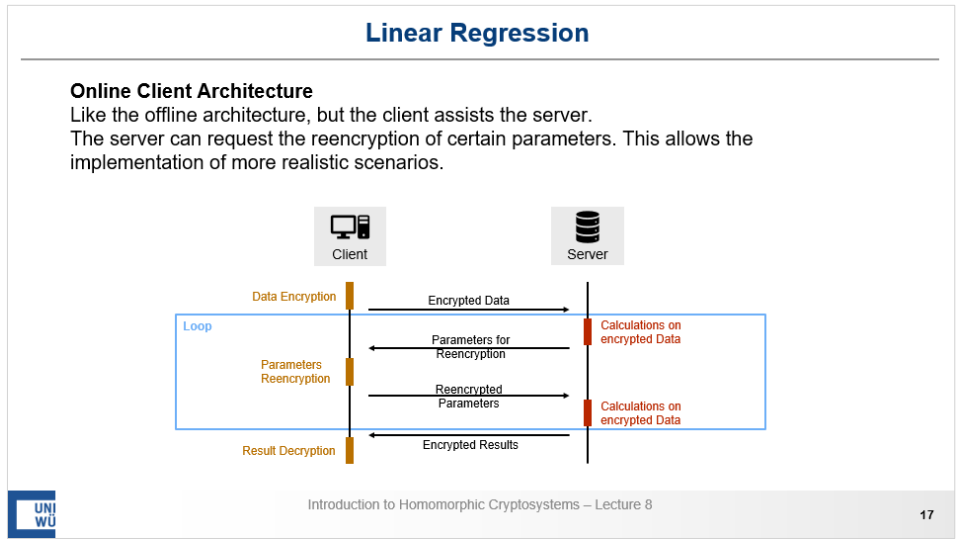
Homomorphic Realization

We can do all the necessary calculations for the Linear Regression homomorphically.
As linear regression with gradient descent is an iterative process with a high multiplicative depth, there are two interesting concepts for the homomorphic realization:

Offline Client Architecture



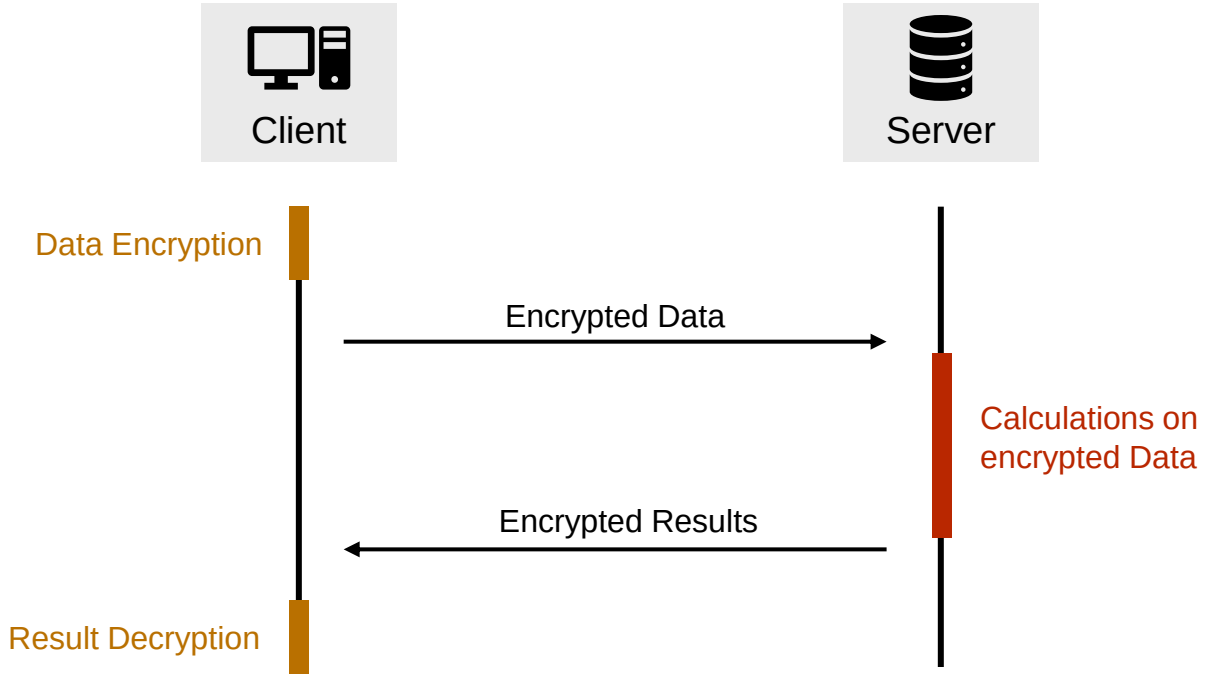
Online Client Architecture



Linear Regression

Offline Client Architecture

The client wants linear regression to be calculated on its dataset, but dees not want to perform these calculation itself.
Furthermore, the data and the results of the calculation should be kept private.

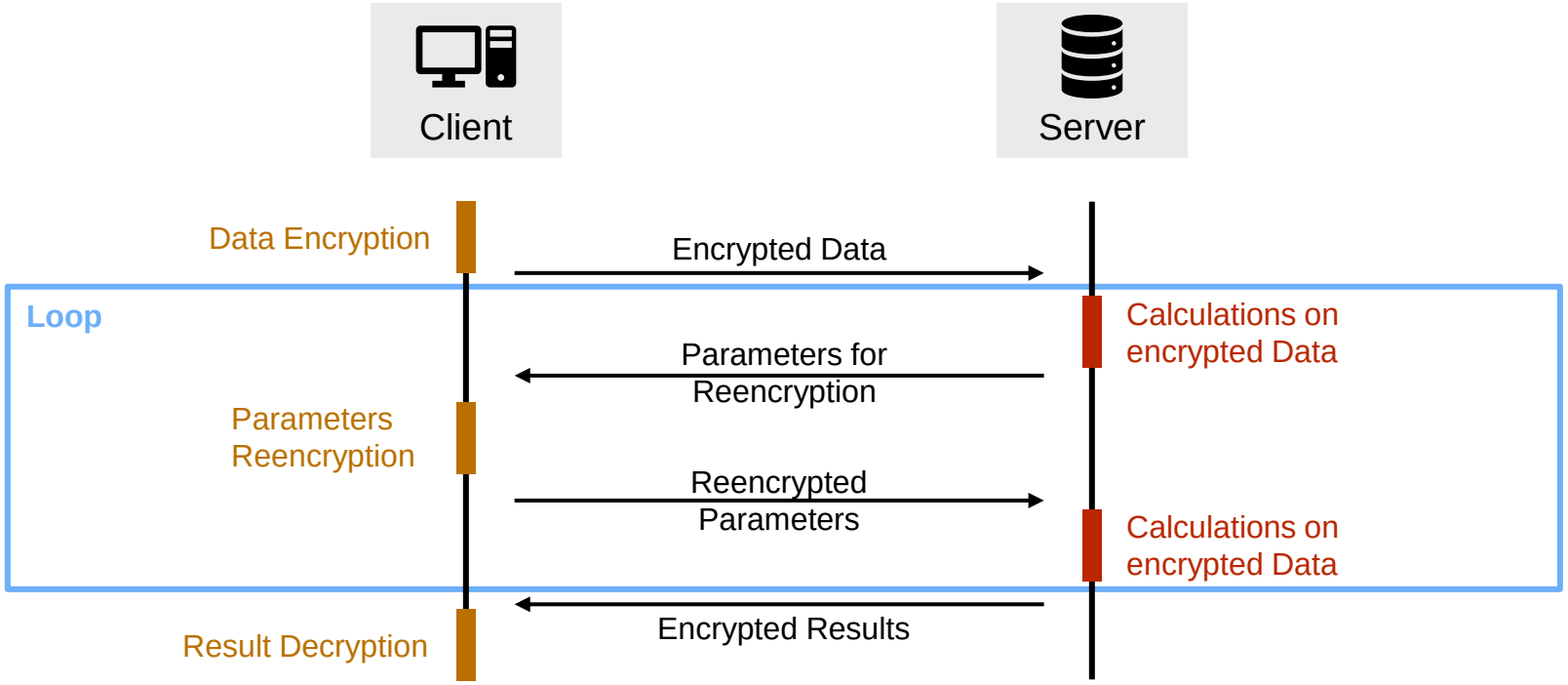


Linear Regression

Online Client Architecture

Like the offline architecture, but the client assists the server.

The server can request the reencryption of certain parameters. This allows the implementation of more realistic scenarios.

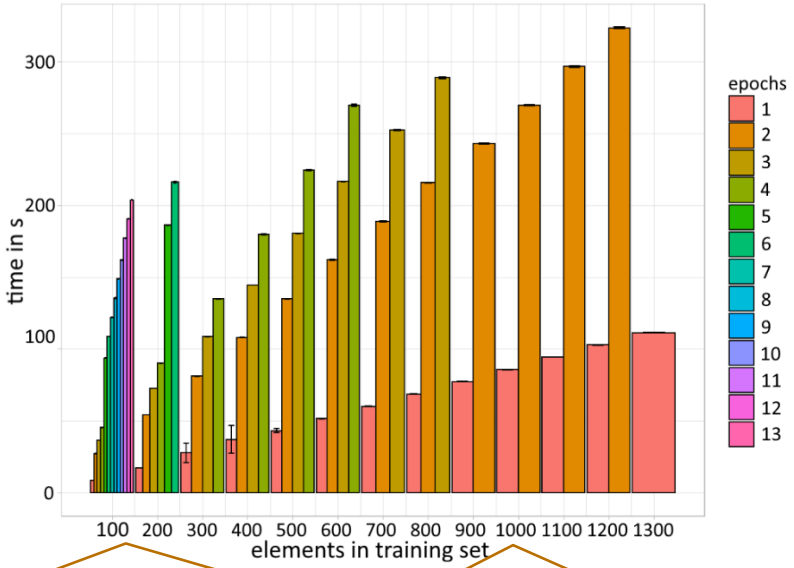


Linear Regression

Evaluation – Offline Architecture

Dataset used for evaluation:
https://www.kaggle.com/datasets/vjchoudhary7/customer-segmentation-tutorial-in-python/data

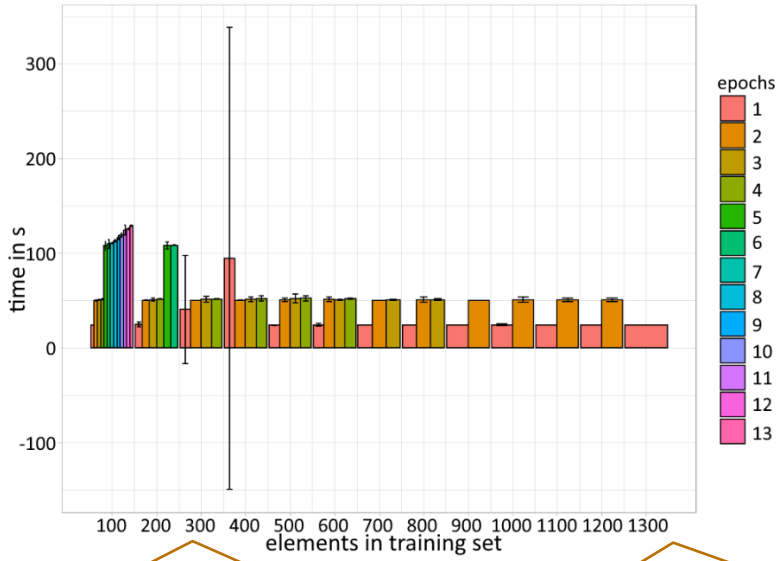
Data Encryption



The encryption time does not only depend on the number of elements, but also on the number of epochs. More epochs mean a higher multiplicative depth and thus a more complex encryption.

High RAM demands (the test system had 32GB RAM) limit the number of epochs for a larger number of elements.

Result Decryption



As the number of elements which have to be decrypted is always the same, decryption is more constant

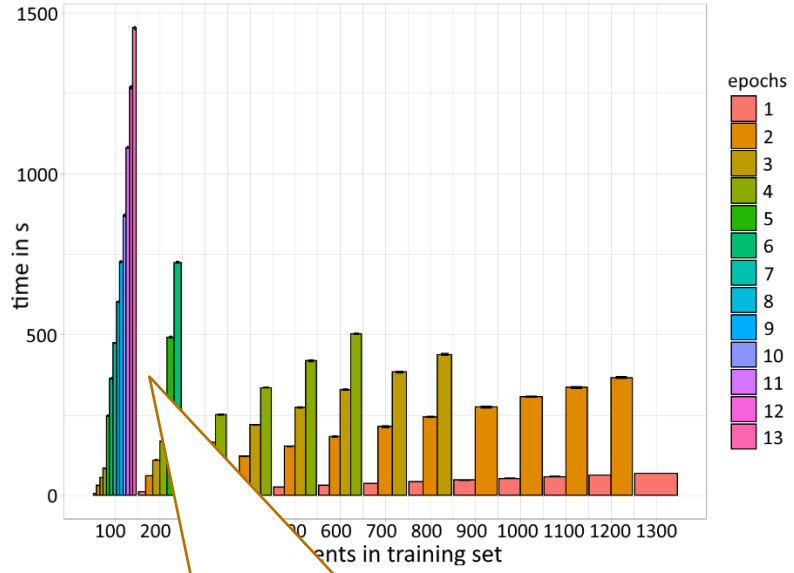
Encryption and decryption may take some minutes, but not hours.

Linear Regression

Evaluation – Offline Architecture

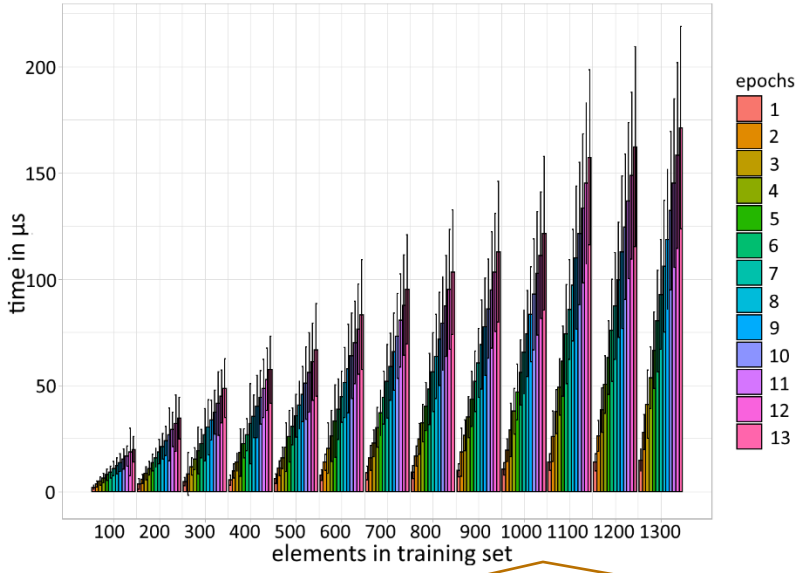
Dataset used for evaluation:
https://www.kaggle.com/datasets/vjchoudhary7/customer-segmentation-tutorial-in-python/data

Homomorphic Calculation



Homomorphic calculation is (unsurprisingly) much slower.

Non-Homomorphic Calculation



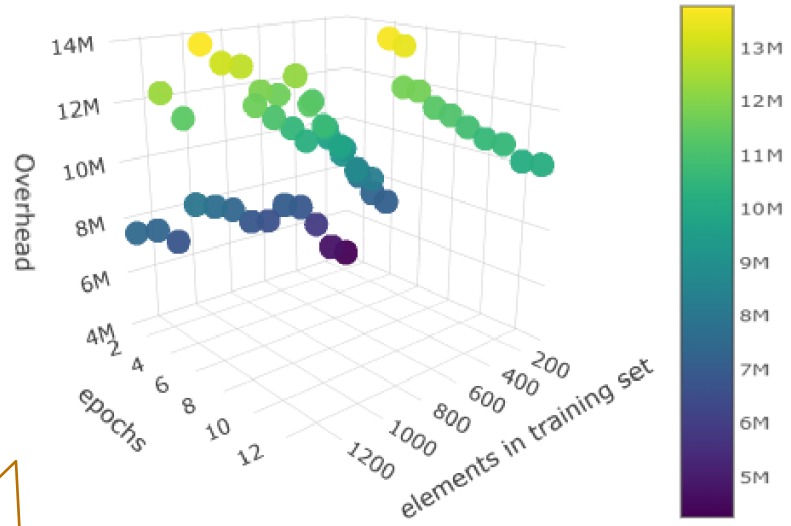
The non-homomorphic calculation does not suffer from high RAM demands. Therefore, every combination of epochs and number of elements is possible.

Linear Regression

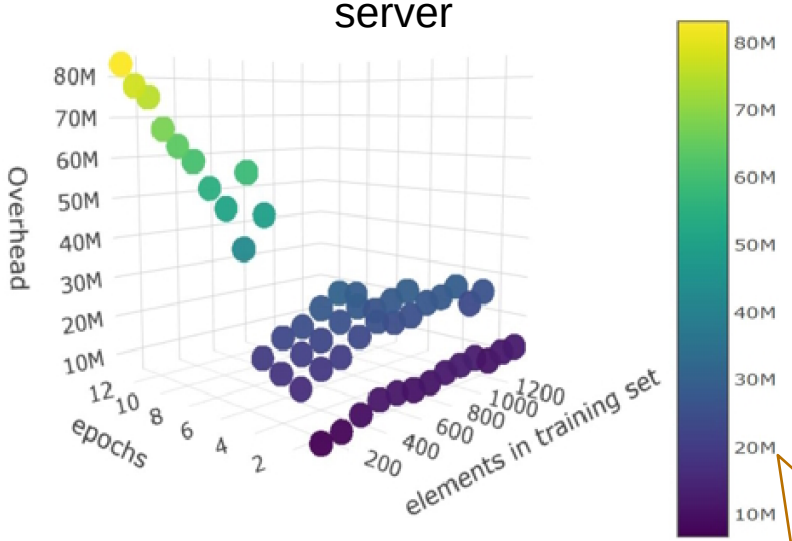
Evaluation – Offline Architecture

Dataset used for evaluation:
https://www.kaggle.com/datasets/vjchoudhary7/customer-segmentation-tutorial-in-python/data

Time overhead of the client



Time overhead of the client and server



Time overhead from client perspective is at best 4 million and at worst 14 million.

In general, time overhead increases with the number of elements and epochs.

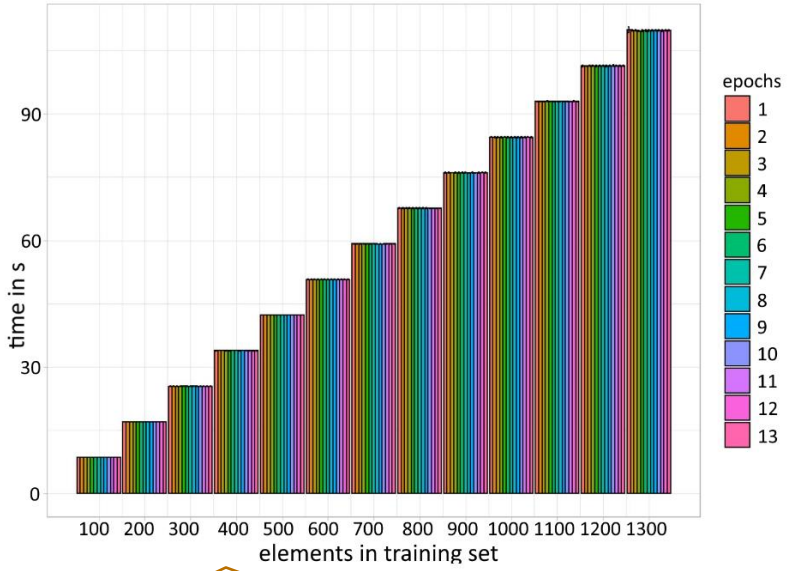
Total time overhead (this includes also the calculations on the server) ranges from 10 to 80 million.

Linear Regression

Evaluation – Online Architecture

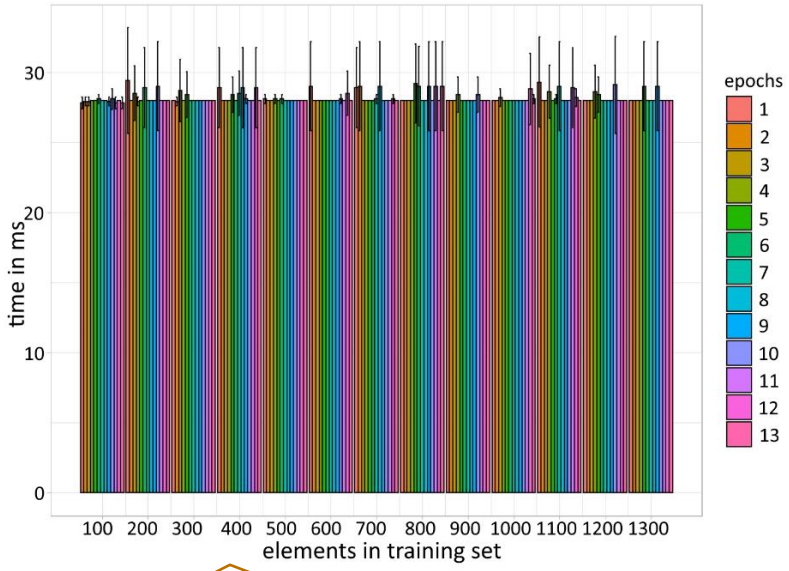
Dataset used for evaluation:
https://www.kaggle.com/datasets/vjchoudhary7/customer-segmentation-tutorial-in-python/data

Data Encryption



As we are able to reencrypt, data encryption and result decryption times do not depend on the number of epochs (multiplicative depth does not play a role during this step).

Result Decryption



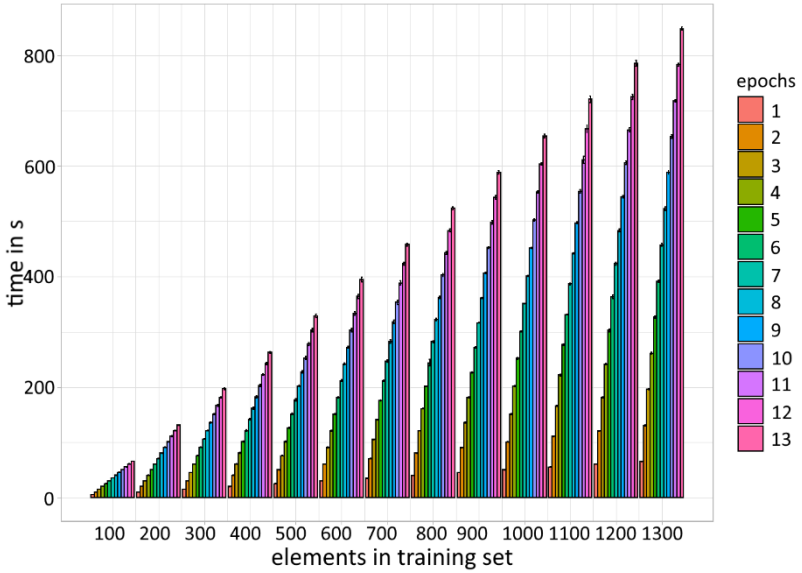
Computation times are also better than in the offline architecture due to the shallower multiplicative depth.

Linear Regression

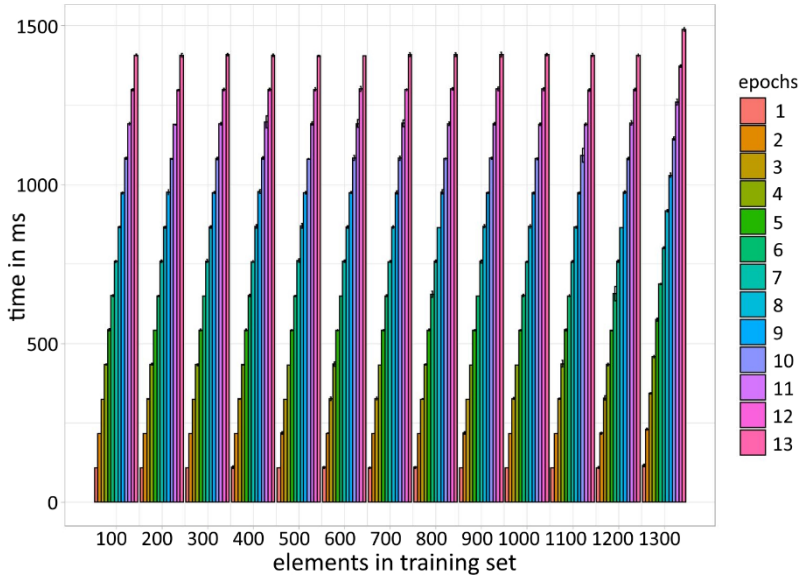
Evaluation – Online Architecture

Dataset used for evaluation:
https://www.kaggle.com/datasets/vjchoudhary7/customer-segmentation-tutorial-in-python/data

Homomorphic Calculation (Server)



Parameters Reencryption (Client)



Better calculation times as in the offline architecture (even for 13 epochs and 1300 elements!).

In general, this architecture provides a significant performance boost (but involves the client). Furthermore, it's obviously also much slower than the non-homomorphic solution.

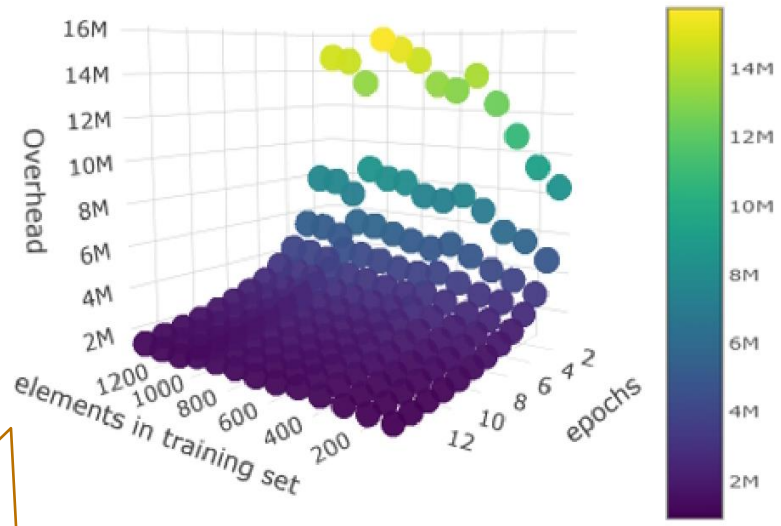
The additional effort for the client is always below 1.5 seconds.

Linear Regression

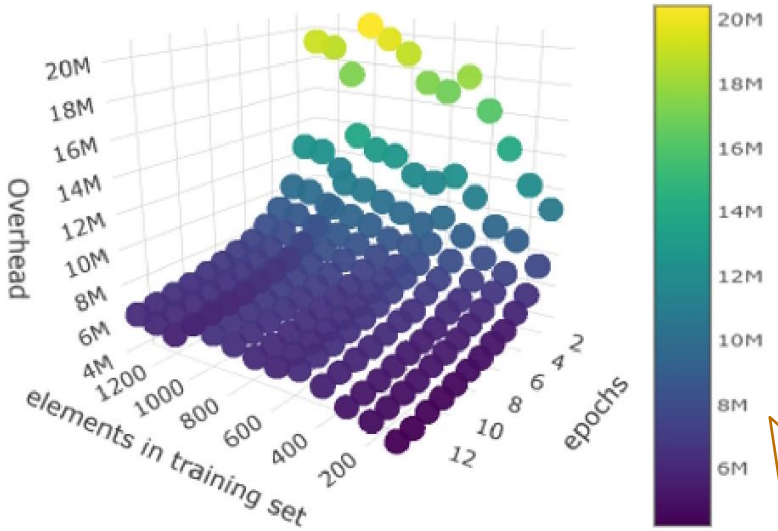
Evaluation – Online Architecture

Dataset used for evaluation:
https://www.kaggle.com/datasets/vjchoudhary7/customer-segmentation-tutorial-in-python/data

Time overhead of the client



Time overhead of the client and server



Time overhead from client perspective is like in the offline architecture.

Total time overhead is way better than in the offline architecture (but also in the range of millions).

K-MEANS CLUSTERING

K-Means Clustering

***k*-means clustering** is the task to segment a given set of datapoints $\{d_1, d_2, \dots d_n\}$ into k clusters $(S_1, \dots S_k)$.
Mathematically this is the problem of optimizing the function

$$J = \sum_{i=1}^{k} \sum_{d_j \in S_i} \|d_j - z_i\|^2$$

$d_j = (\tilde{d}_1, \dots, \tilde{d}_m)^\perp$

$\|x\|$ is usually the Euclidean norm

z_i is the cluster head of cluster S_i and the mean of all datapoints in the cluster:

$$z_i = \frac{1}{|S_i|} \sum_{d_j \in S_i} d_j$$

But how do we find good clusters?

Application
The result of k-means is a data segmentation, that describes the structure of the dataset.
Usually, the algorithm is applied for different values of k to find related datapoints.

K-Means Clustering

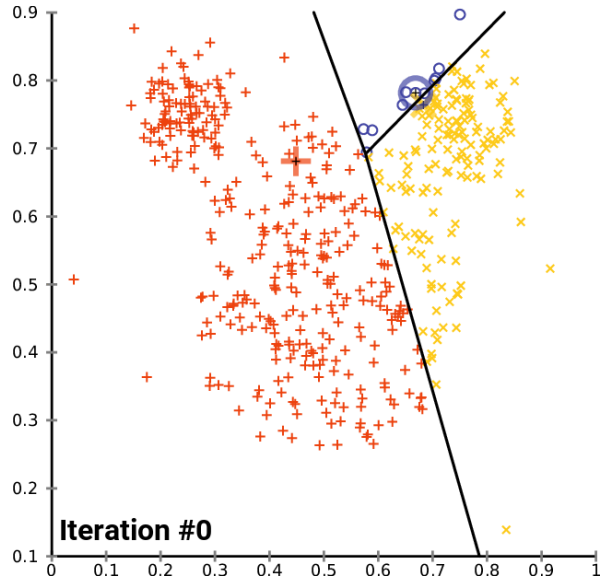
The **Lloyd-algorithm** is the most common way to find a solution for k-means.
However, the problem is NP-hard, which is why the algorithm only gives an approximation for the best result.

Steps
Input: Dataset $\{d_1, \dots d_n\}$ and k

1. **Initialization**
   Choose k random cluster heads:
   $$\left(z_1^{(1)}, \dots, z_k^{(1)}\right), z_i^{(1)} \in \{d_1, \dots d_n\}$$
2. **Assignment**
   For each datapoint, find the closest cluster head and group all points of one cluster head in a cluster:
   $$S_i^{(t)} = \left\{d_j : \left\|d_j - z_i^{(1)}\right\|^2 \leq \left\|d_j - z_{i^*}^{(1)}\right\|^2 \forall i^* = 1, \dots k\right\}$$
3. **Update**
   Recalculate all cluster heads:
   $$z_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{d_j \in S_i^{(t)}} d_j$$

Steps 2-3 are done iteratively until the cluster assignment does not change anymore.

But how can we do that homomorphically?



By Chire - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=59409335

Steps

Input: Dataset $\{d_1, \dots d_n\}$ and k

1. **Initialization**

   Choose k random cluster heads:

   $$\left(z_1^{(1)}, \dots, z_k^{(1)}\right), z_i^{(1)} \in \{d_1, \dots d_n\}$$

2. **Assignment**

   For each datapoint, find the closest cluster head and group all points of one cluster head in a cluster:

   $$S_i^{(t)} = \left\{d_j : \left\|d_j - z_i^{(1)}\right\|^2 \leq \left\|d_j - z_{i^*}^{(1)}\right\|^2 \forall i^* = 1, \dots k\right\}$$

3. **Update**

   Recalculate all cluster heads:

   $$z_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{d_j \in S_i^{(t)}} d_j$$

K-Means Clustering

The **homomorphic realization** of the Llyod-algorithm requires a smart combination of tools we already know. We need to introduce some extra steps to the original setup:

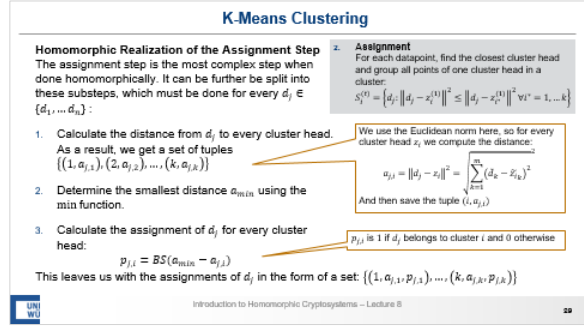
Steps

1. **Initialization**
   Choose k random cluster heads:

2. **Assignment**
   For each datapoint, find the closest cluster head and group all points of one cluster head in a cluster.

3. **Update**
   Recalculate all cluster heads.

The first step can also be done homomorphically with no problem.


K-Means Clustering

Homomorphic Realization of the Assignment Step
The assignment step is the most complex step when done homomorphically. It can be further be split into these substeps, which must be done for every $d_j \in \{d_1, \dots d_n\}$:

1. Calculate the distance from d_j to every cluster head. As a result, we get a set of tuples $\{(1, a_{j,1}), (2, a_{j,2}), \dots, (k, a_{j,k})\}$
2. Determine the smallest distance a_{min} using the min function.
3. Calculate the assignment of d_j for every cluster head:
   $p_{j,i} = BS(a_{min} - a_{j,i})$

This leaves us with the assignments of d_j in the form of a set: $\{(1, a_{j,1}, p_{j,1}), \dots, (k, a_{j,k}, p_{j,k})\}$

2. **Assignment**
For each datapoint, find the closest cluster head and group all points of one cluster head in a cluster:
$S_i^{(t)} = \left\{d_j : \left\|d_j - z_i^{(t)}\right\|^2 \leq \left\|d_j - z_{i^*}^{(t)}\right\|^2 \forall i^* = 1, \dots k\right\}$

We use the Euclidean norm here, so for every cluster head z_i we compute the distance:
$a_{j,i} = \|d_j - z_i\|^2 = \sqrt{\sum_{k=1}^{m} (d_k - z_{ik})^2}$
And then save the tuple $(i, a_{j,i})$

$p_{j,i}$ is 1 if d_j belongs to cluster i and 0 otherwise

Introduction to Homomorphic Cryptosystems – Lecture 8 29

With our results from the assignment step we can update the cluster head z_i like this:

$$z_i^{(t+1)} = \frac{1}{\sum_{j=1}^{n} p_{j,i}} \sum_{j=1}^{n} d_j * p_{j,i}$$

What about termination?
We cannot know after which iteration the clustering is constant.
Which is why we stop after a predefined number of iterations.

K-Means Clustering

Homomorphic Realization of the Assignment Step

The assignment step is the most complex step when done homomorphically. It can be further be split into these substeps, which must be done for every $d_j \in \{d_1, \dots d_n\}$:

1. Calculate the distance from d_j to every cluster head. As a result, we get a set of tuples $\{(1, a_{j,1}), (2, a_{j,2}), \dots, (k, a_{j,k})\}$
2. Determine the smallest distance a_{min} using the $\min$ function.
3. Calculate the assignment of d_j for every cluster head:

$$p_{j,i} = BS(a_{min} - a_{j,i})$$

This leaves us with the assignments of d_j in the form of a set: $\{(1, a_{j,1}, p_{j,1}), \dots, (k, a_{j,k}, p_{j,k})\}$

2. **Assignment**
For each datapoint, find the closest cluster head and group all points of one cluster head in a cluster:

$$S_i^{(t)} = \left\{ d_j : \left\| d_j - z_i^{(1)} \right\|^2 \leq \left\| d_j - z_{i^*}^{(1)} \right\|^2 \forall i^* = 1, \dots k \right\}$$

We use the Euclidean norm here, so for every cluster head z_i we compute the distance:

$$a_{j,i} = \left\| d_j - z_i \right\|^2 = \sqrt{\sum_{k=1}^{m} \left(\tilde{d}_k - \tilde{z}_{i_k} \right)^2}^2$$

And then save the tuple $(i, a_{j,i})$

$p_{j,i}$ is 1 if d_j belongs to cluster i and 0 otherwise

K-Means Clustering

Evaluation
Similar to the linear regression we can implement the calculation of the k-means algorithm in an offline and online architecture.
At this point only the results of the online architecture are ready to present.

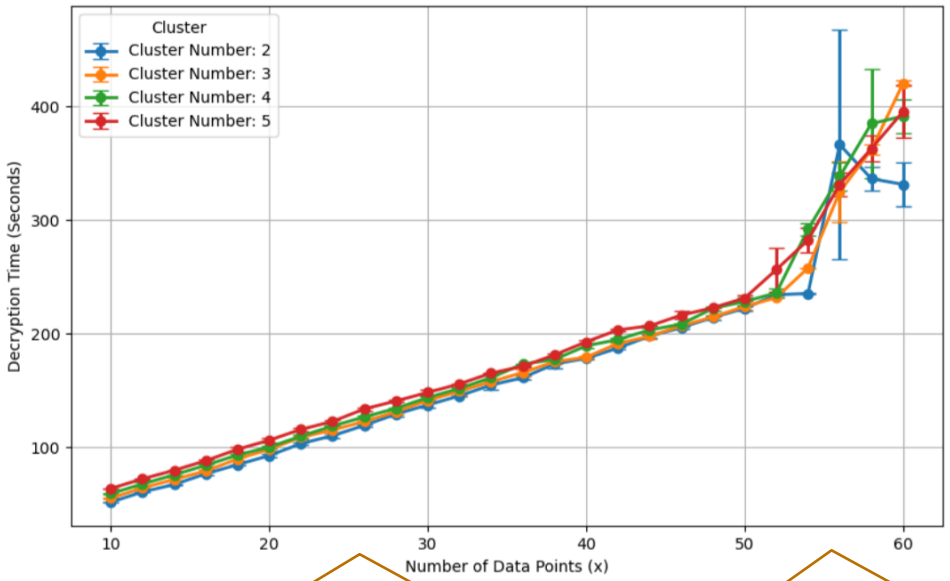
Furthermore, **one iteration** of the Llyod algorithm was executed for every combination of number of datapoints and cluster size.

K-Means Clustering

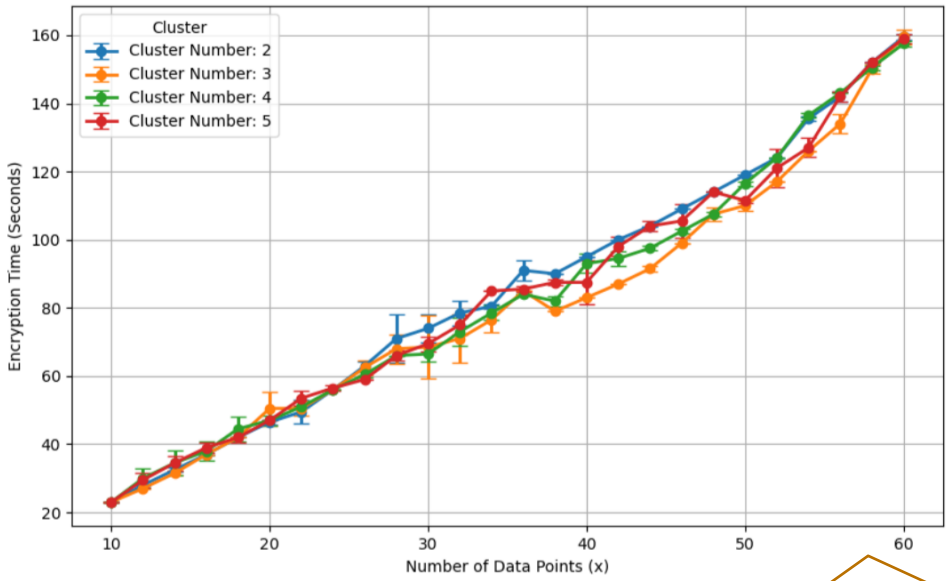
Evaluation – Online Architecture

Dataset used for evaluation:
https://www.kaggle.com/datasets/vjchoudhary7/customer-segmentation-tutorial-in-python/data

Result Decryption



Data Encryption



Encryption and decryption seem to be independent from the number of clusters (which is plausible).

Larger amounts of datapoints result in higher encryption/decryption time (also plausible).

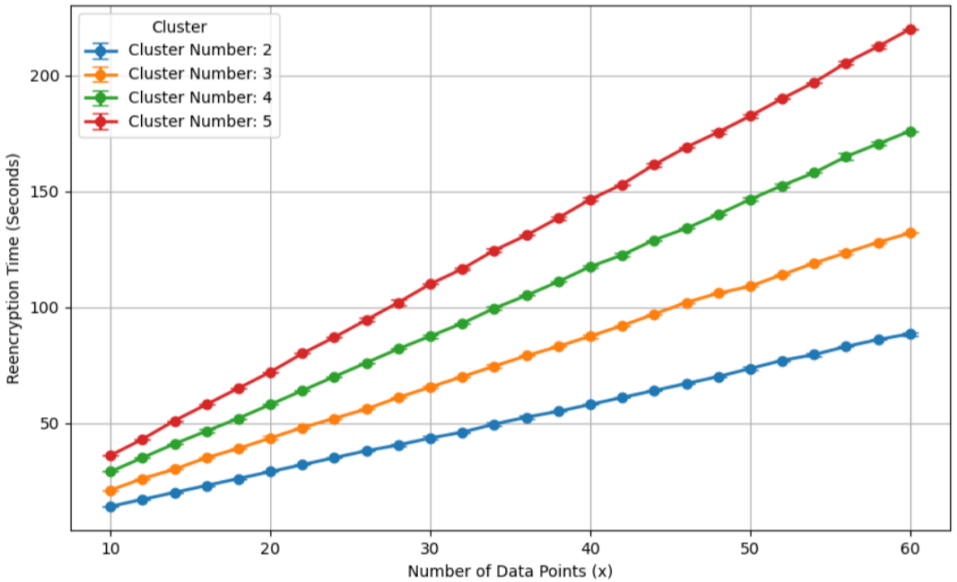
Encryption is faster than decryption.

K-Means Clustering

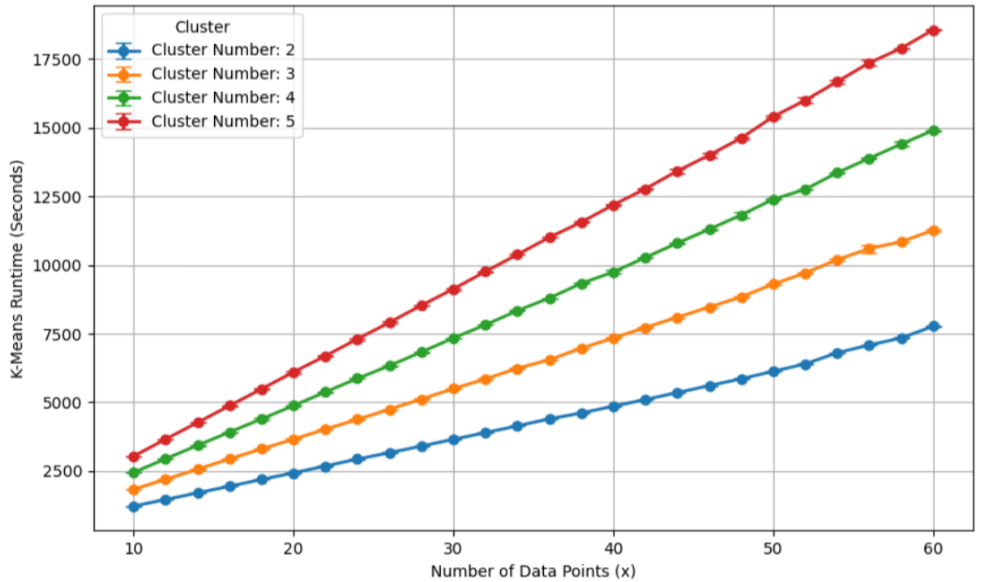
Evaluation – Online Architecture

Dataset used for evaluation:
https://www.kaggle.com/datasets/vjchoudhary7/customer-segmentation-tutorial-in-python/data

Parameters Reencryption (Client)



Homomorphic Calculation (Server)



Client calculations are similar to encryption and decryption times.

In general, more clusters result in higher computation times

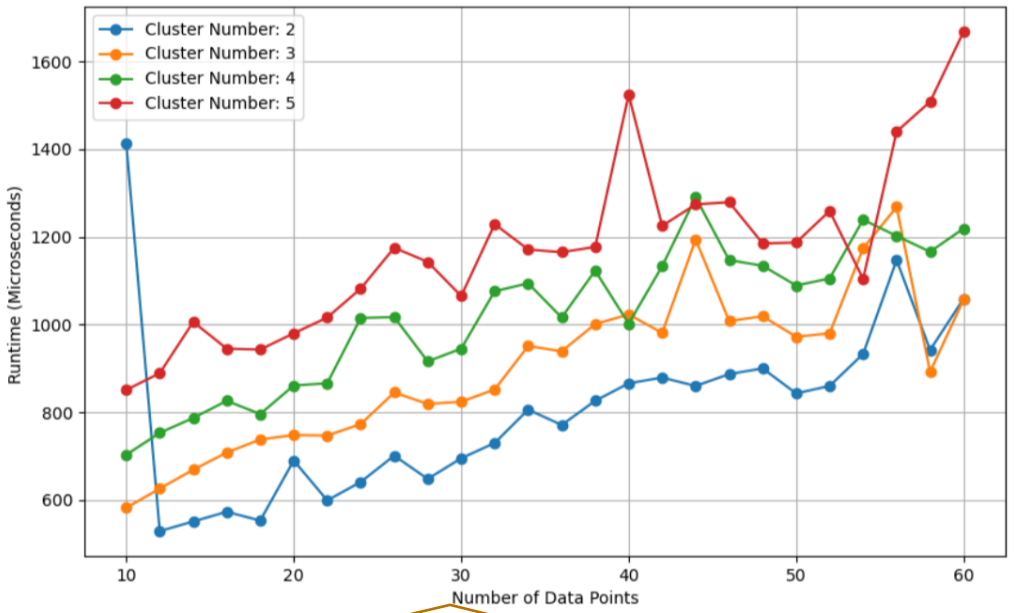
Server calculations are way more time consuming

K-Means Clustering

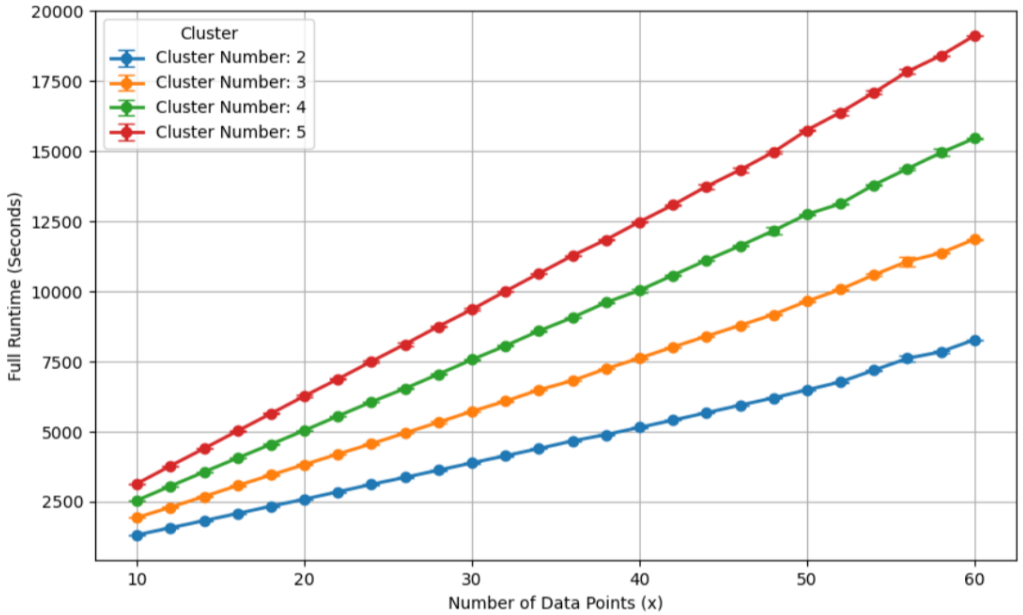
Evaluation – Online Architecture

Dataset used for evaluation:
https://www.kaggle.com/datasets/vjchoudhary7/customer-segmentation-tutorial-in-python/data

Non Homomorphic Runtime



Total (Client and Server) Homomorphic Runtime



These fluctuations may be due to other influences (system processes etc.)

As in the other applications, the homomorphic computation introduces a large performance overhead

Summary – What did we learn today?

Homomorphic implementation more advanced applications

Box-Cox Transformation
A function, that allows arbitrary data to be transformed from a non-normal distribution to a normal distribution.

Linear Regression with Gradient Descent
Tuning feature weights to find a good representation of the dataset in one linear relation.

k-Means Clustering
Data segmentation into k different groups. Each group should contain ‘similar’ data.

All the applications can be done homomorphically by applying our previously implemented extension of the CKKS cryptosystem. They all can do their respective tasks with good accuracy but introduce a large performance overhead.