A CURE clustering algorithm implementation in Scala with Spark*

Mining of Massive Datasets, MSc Data & Web Science

Vasileios Moschopoulos
MSc Data & Web Science
AUTH
Thessaloniki, Greece
vmoschop@csd.auth.gr

Georgios Michoulis
MSc Data & Web Science
AUTH
Thessaloniki, Greece
gmichoul@ csd.auth.gr

1 Introduction

Traditional clustering algorithms either favor clusters with spherical shapes and similar sizes or are very fragile in the presence of outliers. CURE clustering algorithm [1] is more robust to outliers and identifies clusters having non-spherical shapes and wide variances in size.

CURE (Clustering Using Representatives) is a distributed hierarchical clustering algorithm that can capture complex cluster shapes using several representatives instead of a centroid. It uses a sample of the data to do that, its power lies in its scalability and reduced running time due to sampling.

Having more than one representative point per cluster allows CURE to adjust well to the geometry of non-spherical shapes and the shrinking helps to dampen the effects of outliers.

In this work, we created an implementation of CURE, as well as a K-Means & Agglomerative implementation for comparison. We also devised a method for handling outliers. We discuss the correctness of our implementation, scalability, compare against our K-Means & Agglomerative solution, and present results on handling outliers.

The remaining of this work is structured as follows: In section 2 we present the dataset we used, the preprocessing procedure, and devise a method for inserting outliers. In section 3 we briefly discuss the CURE architecture and present our implementation. In section 4 we present our K-Means & Agglomerative implementation. In section 5 we present results from our experiments. And finally, in section 6 we discuss our conclusions.

2 Dataset and Preprocessing

This section is dedicated to the dataset that was used and the preprocessing procedure that was applied to it to become useful for our experiments.

The section is structured as follows: **(i)** firstly, we discuss the characteristics of the dataset that we were provided, including file structure, cardinality, and shape, and **(ii)** secondly, we outline the preprocessing procedure. Preprocessing included increasing the cardinality of the dataset while maintaining its shape and inserting outliers.

Increasing the cardinality of the dataset was crucial to our goals since, to measure the scalability of our implementation, a high number of points was necessary. On the other hand, outlier insertion in the dataset was necessary to measure our proposed method for handling outliers.

2.1 Dataset

Out initial dataset consisted of 2 dataset folders containing 100 duplicate files with several two-dimensional points. There were 5725 points in each file of the first folder and 16924 points in each file of the second folder. Points in both folders were shaped in the form of 5 pentagons, 4 large ones and a smaller one, with the pentagon's getting denser towards the mean of the points. Figure 1 accurately depicts the shape of the data in the first folder. It is worth mentioning that points in both folders presented an identical distribution and shape. A distribution plot for both dimensions is visible in Figure 2.

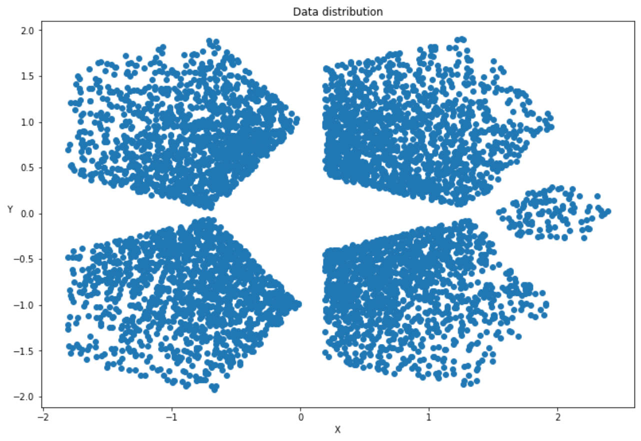


Figure 1: The shape of the points in the data. The points are shaped in the form of 5 pentagons, with 4 large pentagons and 1 small one.

In general, points in both dataset folders were center around (0, 0), presented standard deviation of 0.94 and 0.91, for dimensions 0 and 1 respectively, and covariances of ~0.85 regarding their same dimension, and ~-0.32 regarding the alternate dimension. All statistics can be seen in detail in Table 1.

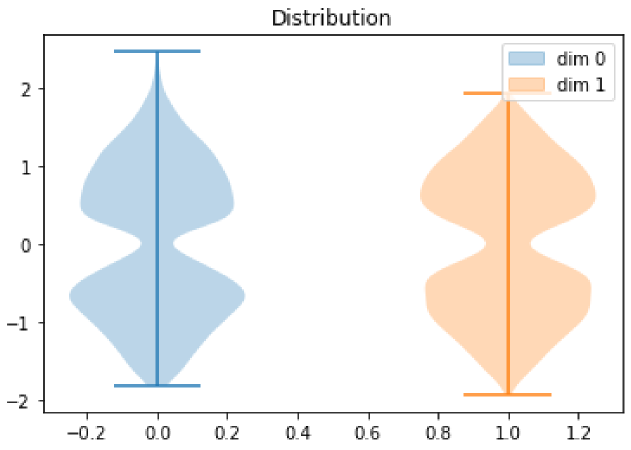


Figure 2: The distribution of each dimension in the data.

	Mean	*Standard deviation*	*Covariance same dim.*	*Covariance other dim.*
Dim. 0	0	0.94	0.89	-0.32
Dim. 1	0	0.91	-0.32	0.83

Table 1: Dataset statistics.

2.2 Dataset preprocessing

As discussed earlier, preprocessing the data was crucial in order to test the correctness, scalability, and outlier handling methodology of our implementation.

2.2.*1* Increasing cardinality.

Since our implementation cannot handle duplicate data, we had to find a way to test the scalability without using the duplicate points found in the 100 files of each dataset folder. In order to maintain the shape of the data, a solution that would qualify needed to be found.

Our solution eventually consisted of increasing the cardinality of both datasets by factors of 10, 30, and 100. To do that, we duplicated the points by the respective factor and added an infinitesimally small amount of noise, using a uniform distribution ranging between 0.04 and -0.04.

2.2.*2* Inserting outliers.

Outliers were added in all flavors of the dataset, i.e., different cardinalities, in the form of a circle perimeter around the mean of the points, and a radius of 2.9. To avoid creating outliers in the form of a perfect circle, random noise was added to each point, in the form of a scalar, drawn from a uniform distribution, ranging from 0.1 * *radius* to - 0.1 * *radius*. An example in the form of the original dataset with cardinality 16924, along with the added outliers, can be seen in Figure 3.

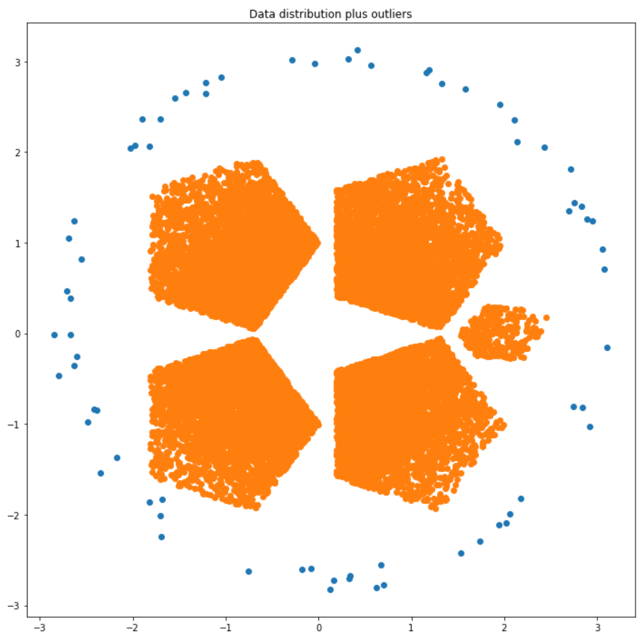


Figure 3: Original dataset of cardinality 16924 with outliers added in the form of a circular perimeter.

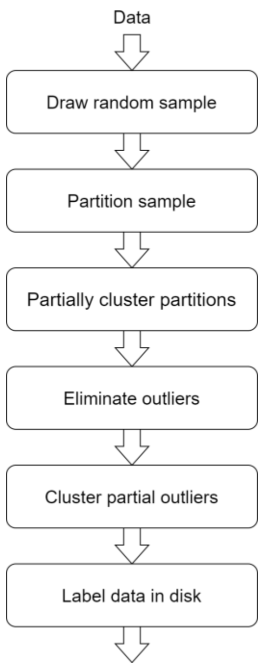


Figure 4: Overview of CURE.

3 The CURE implementation

This section is dedicated to discussing the CURE algorithm. The section is structured as follows: **(i)** firstly, we present an overview of the CURE algorithm in general, **(ii)** secondly, we discuss data structures that CURE uses in order to perform fast range queries among representative points and retrieve clusters having the smallest distance to their closest clusters, and **(iii)** finally, we present our implementation of the CURE algorithm.

3.1 Overview

CURE is a very powerful and scalable hierarchical clustering algorithm that can be used for very large datasets. The algorithm utilizes several representative points, through which it can discover complex cluster shapes. For every pass of the algorithm, the representatives are selected each time as the *m* farthest points in the cluster.

One of the main advantages of CURE is that it inherently combats outliers. The main mechanism that allows it to do so is that representative points are shrunk towards the mean of the cluster at every pass, using a shrinking factor *α*. The shrinking factor can range between 0 (original points contained in the dataset) and 1 (centroid mean).

Also, an important aspect of the algorithm is that it does not use all of the data to be trained on, but instead picks only a small representative sample. The distributed nature of CURE lies over the fact that the selected sample can be further partitioned to be processed individually.

The process can be briefly described as follows:

- Each point in the dataset starts as a separate cluster.
- At each pass, and for every partition, the cluster having one representative point closest to a representative point of another cluster is merged with its closest cluster. This involves combing the cluster points into a new cluster and identifying the new representative points.
- Outliers can be further combated by partially clustering the data and removing clusters with less than *m* representative points, right in the middle of the clustering process. The clustering process then continues with the partially clustered data until *k* clusters emerge.
- At the end of the clustering process, all clusters are collected from each partition and merged, with clusters having less than m representative points being merged with other clusters.
- After the algorithm has been trained, all points in the original dataset are classified into their respective clusters, based on their distance from the representative points.

An overview of CURE can be seen in Figure 4.

3.2 The CURE data structures

CURE utilizes two data structures: **(i)** a MinHeap to accelerate merging clusters and **(ii)** a KD-Tree for performing range queries to find the representatives with the closest distance.

3.2.*1* The MinHeap.

The MinHeap is a data structure with a list that is internally implemented in the form of a binary tree. The first element in the list, and the leftmost top node in the tree, is the element that has the smallest property value. In this case, the property is the minimum distance between a cluster and its nearest cluster.

Elements of the MinHeap comprise cluster object that contains:

- A list of all points
- A list of representatives
- The centroid of all points within the cluster
- A pointer to the nearest cluster
- The squared Euclidean distance from the nearest cluster
- And finally, an identifier number.

If a cluster node is stored at index k, then the left cluster child is at index 2k + 1, while the right cluster child is at index 2k + 2. Additionally, the parent of every cluster node is at index k / 2.

Getting the cluster with the minimum nearest-cluster distance (min cluster) requires an operation of O(1) time, extracting the min cluster requires an operation of O(log N) time, while inserting a new cluster requires an operation of O(log N) time.

3.2.*2* The KD-Tree.

The KD-Tree is a binary tree structure that can deal with elements containing multiple dimensions instead of one (K-Dimensional Tree). At each level l of the tree, nodes are sorted into left and right using the l mod d dimension, where d is the number of dimensions of the nodes.

The KD-Tree in CURE is used for performing fast range queries between representative points, in order to find the closest representative of another cluster, for a given representative.

Nodes of the KD-Tree comprise objects containing:

- The representative point
- A pointer to the left node
- A pointer to the right node
- And an additional boolean property, which we discuss in paragraphs to come.

Representative points comprise objects containing:

- The coordinates of the point
- And the cluster to which the point belongs.

Regarding a regular KD-Tree, constructing the tree requires an operation of O(log N) time, adding a representative requires an operation of O(log N) time, removing a representative requires an operation of O(log N) time, while querying for the closest representative requires an operation of $O(n^{1-1/d} + m)$ time, where m is the number of reported representatives and d is the dimension of the tree.

However, our implementation was slightly different. In order to reduce the running times of the algorithm, at the expense of additional memory space, instead of removing representatives, we added a boolean property to every node object. When a representative was to be removed, the boolean property would be set to false, thereby reducing the time that would be needed to remove the representative to O(1). Given a representative is reintroduced at some point in time, the boolean of the according to node object would be set to true and the cluster of the representations contained in the node would be set to the new cluster.

For this exact reason, we cannot handle duplicate points in the dataset, and which is why for increasing the cardinality of the datasets we had to introduce a small random noise.

3.3 Our CURE implementation

Our implementation of CURE was done in Scala using Spark.

The algorithm was split into three major methods:

- The *main* method

```
function MAIN
    Read the data points
    Pick a sample and repartition it
    Map each sample point to a cluster
    Map each sample point to a cluster
    Create the clusters ateachpartition : Cluster() function
    Collect the clusters of all partitions
    Retrieve the representatives of all collected clusters
    Merge collected clusters and clusters short of m representative points
    Label all data points using the representatives of the final clusters
    Return the labeled points, the final representatives, and the final sample cluster means
```

Figure 7: The CURE main method

```
function CLUSTER
    If the partition contains less than m points, return a new cluster with all of the points
in the partition as points and representatives, and the corresponding mean
    Create the KD − Tree using all points as representatives
    Create the MinHeap containing all clusters, using all points as separate clusters
    If removeOutliers is true partially cluster points until there 2 ∗ k clusters :
ComputePartitionClusters() function
    Continue / Fully cluster all clusters : ComputePartitionClusters() function
    Return all clusters from the MinHeap
```

Figure 6: The CURE cluster method

```
function COMPUTEPARTITIONCLUSTERS
    while MinHeap.size > k or MinHeap.size > 2 * k do
        Merge the nearest clusters
        Remove their representatives from the KD − Tree
        Remove them from the MinHeap
        Insert the new merged cluster
```

Figure 5: The CURE computePartitionClusters method

- The *cluster* method
- And the *computePartitionClusters* method

The CURE algorithm accepts the following parameters:

- Number of clusters: The number of representatives must be larger than 1
- Number of representatives: The number of clusters must be larger than 0
- Shrinking factor-alpha: The shrinking factor needs to be between 0 and 1
- Number of partitions: The number of partitions allowed is between 1 and 99
- Input file/folder: The input must be a valid path
- Sampling ratio: The sample ratio must be larger than 0 and smaller or equal to 1
- removeOutliers: This is a boolean indicating whether outlier must be filtered

After the algorithm has run, the data points along with their predicted clusters, the representative points, and the means of the sample clusters are stored in a new folder with the current timestamp.

The flow of the *main* method is implemented as follows:

- The data points are read and loaded in the form of a Spark RDD
- A sample is picked in the form of an RDD and the RDD is repartitioned
- Each sample point is mapped to a new cluster
- The clusters are created at each partition using the *cluster* function
- The clusters of all partitions are collected

- The representatives of all collected clusters are retrieved
- The collected clusters and the clusters short of *m* representative points are merged
- All data points are labeled using the representative points
- The labeled points, the final representatives, and the final sample cluster means are returned

The flow of the *cluster* method is implemented as follows:

- If the partition contains less than m points, a new cluster with all of the points in the partition as points and representatives, and the corresponding mean is returned
- The KD-Tree is created using all points as representatives
- The MinHeap containing all clusters is created, using all points as separate clusters
- If removeOutliers is true, points are partially clustered until there 2 * k clusters, using the *computePartitionClusters* function
- The process continues / fully clusters all clusters using the *computePartitionClusters* function
- All clusters from the MinHeap are returned

Finally, the flow of the *computePartitionClusters* function is implemented as follows:

- While the heap size is larger than k (or 2 * k in the case of outlier filtering):
    - Merge the nearest clusters
    - Remove their representatives from the KD-Tree
    - Remove them from the MinHeap
    - And insert the new merged cluster to the MinHeap

4 The K-Means & Agglomerative algorithm implementation

As a comparison, we also created a conjoined implementation, using the K-Means algorithm and then applying a simple average-linkage Agglomerative clustering algorithm on the predicted K-Means clusters. This implementation was also done in Scala[1].

The algorithm begins with a higher number of *initialClusters* clusters for the K-Means algorithm and then agglomerates the predicted K-Means clusters into *finalClusters* clusters using the Agglomerative algorithm.

4.1 K-Means

K-means is one of the most commonly used clustering algorithms that cluster the data points into a predefined number of clusters.

The implementation was done using the K-Means algorithm provided by Spark MLLib.

For K-Means we set the following parameters:

- A number of max iterations *maxIterations* equal to 100.
- Initialization mode to 'k-means||'. The "k-means||" initialization mode is a scalable implementation of the K-Means algorithm, and it is similar to "k-means++"[2].

4.1 Agglomerative clustering

The Agglomerative algorithm, also known as AGNES (Agglomerative Nesting), is a very powerful clustering algorithm, albeit often having a highly prohibitive cost for applying to very large data.

There are several different flavors of the Agglomerative algorithm, of which we selected the average-linkage flavor due to its simplicity.

Our implementation was a very simple one. By providing the initial clusters to the algorithm, the algorithm would calculate the means of each cluster, calculate pairwise distances using the means, and then would join at each step the clusters having the closest distance.

5 Results

5.1 Evaluation Metrics

In this study, we evaluate some metrics in order to understand how well the clustering algorithms perform. For that reason, we need to find a base of metrics that can show the clustering performance. The most popular metrics and the best so far are the Silhouette and WSSSE (Within Set Sum Squared Error) metrics.

5.1.*1* Silhouette

Silhouette refers to a method of interpretation and validation of consistency within clusters of data. The technique provides a succinct graphical representation of how well each object has been classified. The silhouette value is a measure of how similar an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette ranges from −1 to +1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters. If most objects have a high value, then the clustering configuration is appropriate. If many points have a low or negative value, then the clustering configuration may have too many or too few clusters. The silhouette can be calculated with any distance metric, such as the Euclidean distance or the Manhattan distance.

Silhouette Coefficient = (x-y)/ max(x,y)

5.1.*2* WSSSE

Error Sum of Squares (SSE) is the sum of the squared differences between each observation and its group's mean. It can be used as a measure of variation within a cluster. If all cases within a cluster are identical the SSE would then be equal to 0.

All it is, for the WSSSE score we look at the distance from each point to its centroid, the final centroid in each cluster, take the square of that error, and sum it up for the entire Dataset. It's just a measure of how far apart each point is from its centroid. Obviously, if there's a lot of error in our model then they will tend

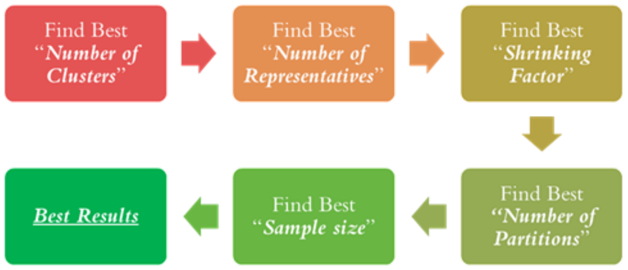
[1] https://spark.apache.org/docs/latest/mllib-clustering.html#k-means

[2] k-means++ - Wikipedia

to be far apart from the centroids that might apply, so for that we need a higher value of K, for example.

5.2 Correctness

In order to find the best variables formula for the clustering on our dataset by the CURE algorithm, we have to find to test all the variables and based on some metrics and visualization to choose the optional variables. As shown in Figure 8, there are 5 steps. In each step, we randomly pick some numbers for our variables and then evaluate them by visualizing the results and comparing the Silhouette and WSSSE metrics.



Find Best "Number of Clusters"
Find Best "Number of Representatives"
Find Best "Shrinking Factor"
Find Best "Number of Partitions"
Find Best "Sample size"
Best Results

Figure 8: Steps to check correctness.

5.2.*1* Number of clusters.

In the first step, we wanted to find the best variable for the number of clusters that the algorithm will produce as a result of its procedure. In order to do so, we executed many instances and examples. Moreover, we set the other variable randomly as such:

- 4 representatives
- Shrinking factor 0.5
- 5 partitions
- Dataset cardinality 169240
- Sample size 13773 – sampling ratio 0.08

For our experiments, we examine a number of 10, 8, 5, 3, and 2 clusters. For each number of clusters, we ran the CURE algorithm and stored its results. What we found was insightful. Silhouette score gradually increases up to 5 clusters and then gradually decreases as the number of clusters increase. So, from the first metric, we can conclude that 5 clusters are the optimal variable, but we have to further verify our results with the next metric.

The Within Sum of Squared Error (WSSE) steadily decreases as the number of clusters increase. The observation here is that with a number of 5 clusters WSSE decreases significantly with a large slope, and after that point, it decreases steadily. This means that the number of 5 clusters is verified by the WSSE. This is natural due to smaller distances of points from their means since clusters get smaller as the number of the predicted clusters increase. We also observe a few peculiarities where clusters can overlap, especially in lower numbers of clusters, which we visualize in Figure 11. We must highlight here, that for Figure 11 and all following Figures, representatives are represented with circles, sample centroids are represented with crosses, while actual data centroids are represented with stars.

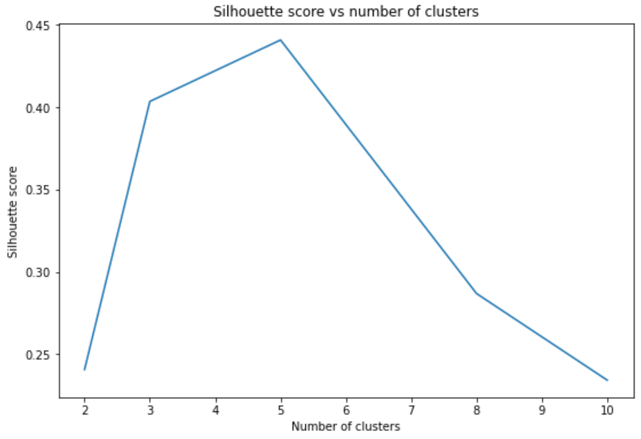


Figure 9: Silhouette score for different numbers of clusters.

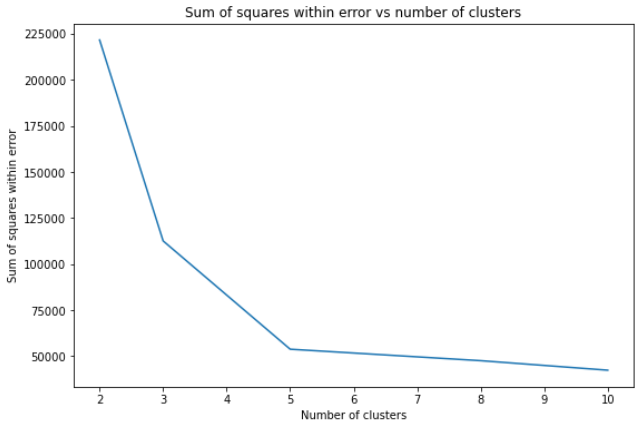


Figure 10: WSSSE for different numbers of clusters.

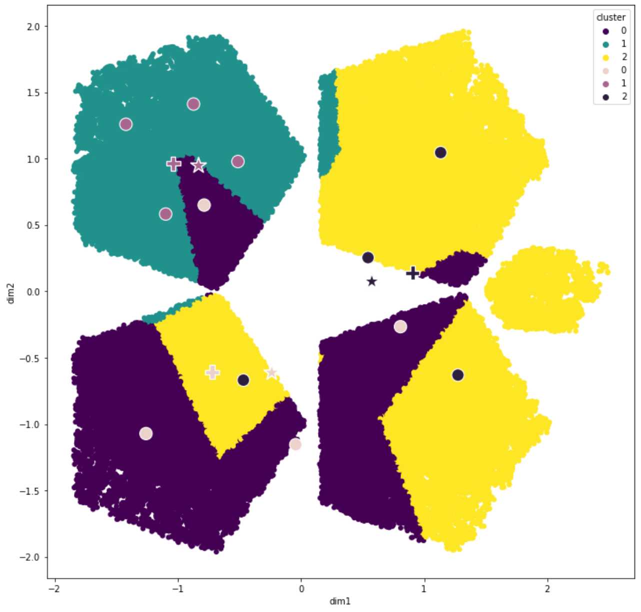


Figure 11: In low numbers of clusters, clusters can overlap. Here we see results for 3 clusters.

5.2.*2* Number of representatives.

The results of the first step teach us that the optimal number of clusters is 5, so we are going to use it from now on. In the second step, we will find the number of representatives that is optimal to create for the clusters. So, we start again executing the algorithm in a loop, trying to find the best representatives' number. To do so, we initialize randomly the other variables, except for the number of clusters:

- 5 clusters
- Shrinking factor 0.5.
- 5 partitions
- Dataset cardinality 169240
- Sample size 13773 – sampling ratio 0.08

We executed the algorithm 6 times with different representative values each time. We tried once each time for 1, 2, 3, 4, 5, and 7 representatives.

Clustering quality heavily improved with higher numbers of representatives with higher silhouette scores and a lower sum of squares within errors. We also observed some cases when a cluster can be lost due to having less than 4 representatives. Despite defining 5 clusters, we retrieve back 4. Similarly, with the last step, the Silhouette and the WSSSE show that the optimal number of representatives is 4. To verify it again, we visualized the results in Figure 14, with the circles to be the representatives and the crosses to be the mean of each cluster.

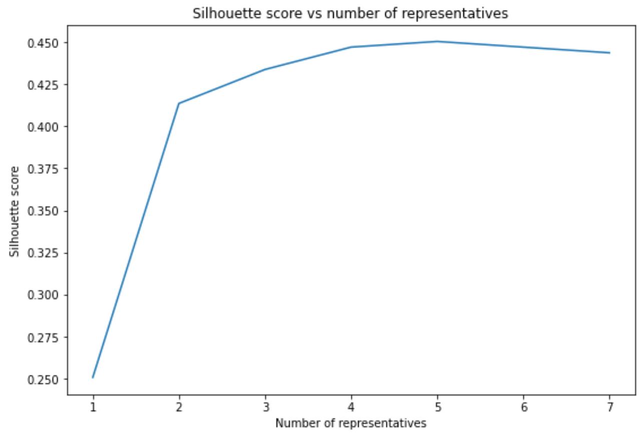


Figure 12: Silhouette score for different numbers of representatives.

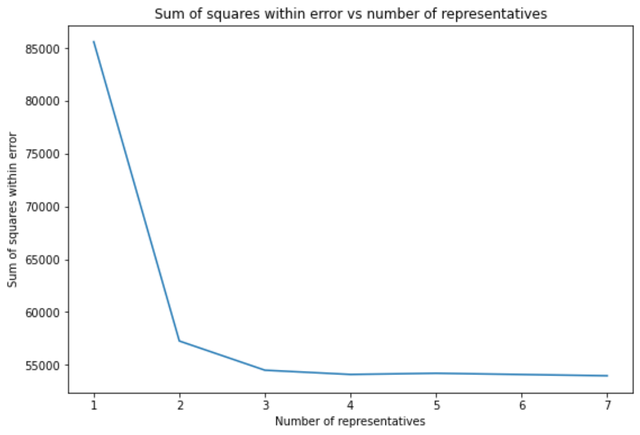


Figure 13: WSSSE for different numbers of representatives.

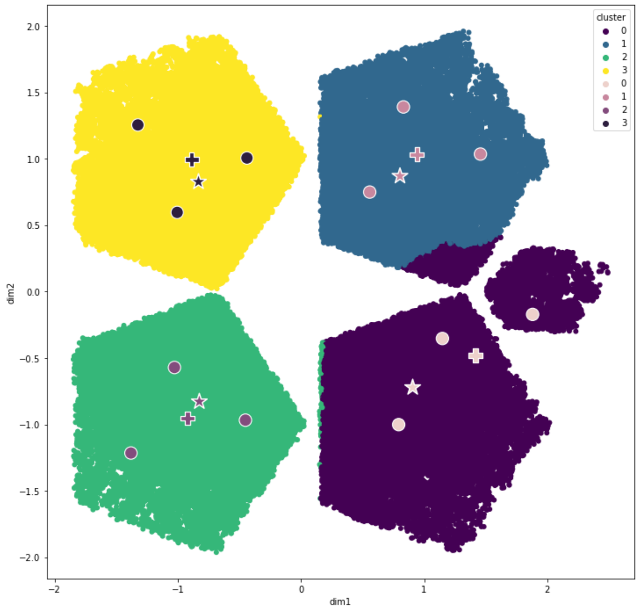


Figure 14: When defining less than 4 representatives a cluster can be occasionally lost. Here we defined 5 clusters, but we got back only 4.

5.2.*3* Shrinking factor.

At this point, one may realize that testing each variable and examining the results can take a lot of time. Eventually, however, we reach our best results. This time, we examine the optimal shrinking factor. Once again, we randomize the other variables, setting the number of clusters to 5 and the representative points to 4.

- 5 clusters
- 4 representatives
- 5 partitions
- Dataset cardinality 169240
- Sample size 13773 – sampling ratio 0.08

We tried the following experiments with 0.25, 0.35, 0.5, 0.7, and 0.9 numbers for the shrinking factor. What we found was interesting.

We observe that low shrinking factors displayed worse silhouette scores and the sum of squares within errors. Figures 15 and 16 depict that after 0.5 shrinking metrics improve marginally. Therefore, we select 0.5 as the optimal shrinking factor.

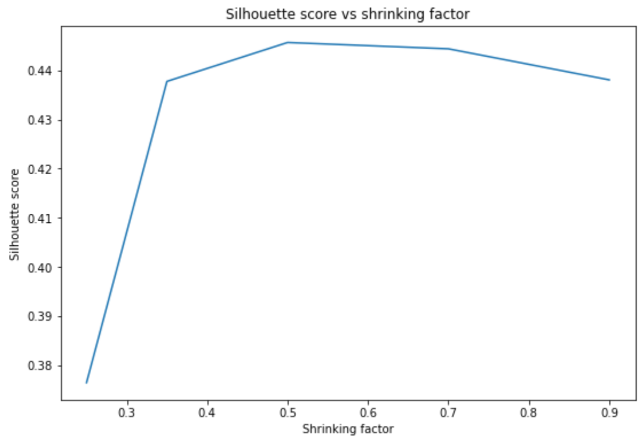


Figure 15: Silhouette score for different shrinking factors values.

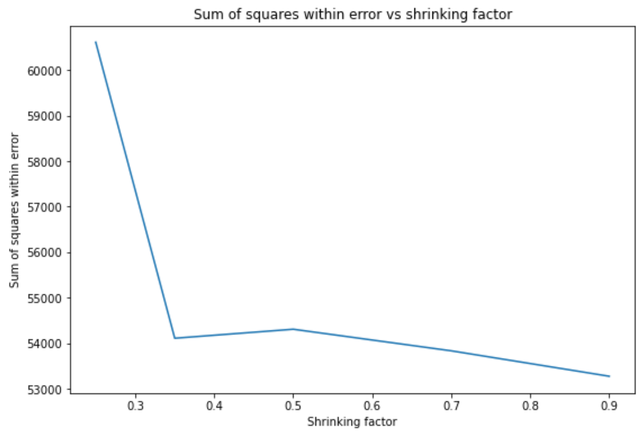


Figure 16: WSSE for different shrinking factor values.

5.2.*4* Number of partitions.

In this step, our goal was to find the optimal number of partitions. As we saw in previous steps, the optimal values for the number of clusters, representatives, and shrinking factor are 5, 4, and 0.5, respectively, which we set is what we set them to for this experiment.

For this step, we tested for 1, 2, 3, 4, 5, 7, and 10 partitions.

From our findings, we observed that clustering quality was not affected, and was very good. We also observed however that for lower numbers of partitions a cluster can be occasionally lost. In Figure 17 we see that despite the clustering quality being excellent, for 5 clusters, one cluster is missing.

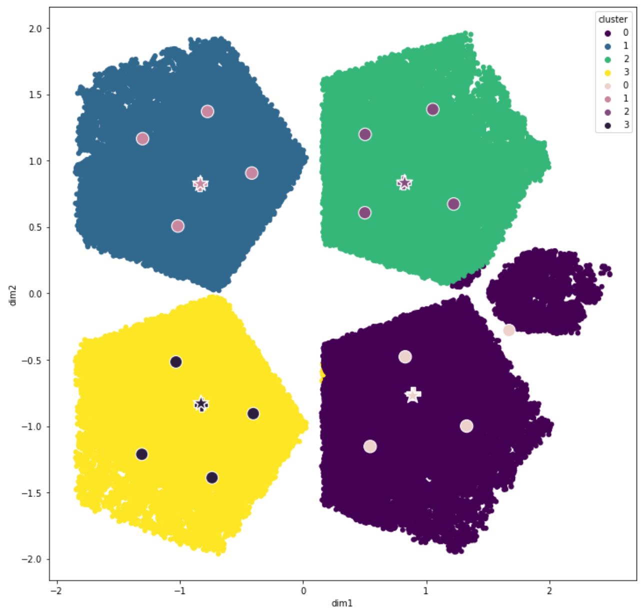


Figure 17: Lower number of partitions can cause a cluster to be lost. In this case, we see that for 5 clusters, one cluster missing.

5.2.*5* Sample size.

In the fifth and final step, our goal was to discover the optimal sample size.

Based on previous steps, we set other hyperparameters as follows:

- 5 clusters
- 4 representatives
- 0.5 shrinking factor
- 5 partitions
- 169240 total points - a dataset with 16924 points with x10 cardinality

For our experiments, we tried sample sizes of 1000, 2000, 3500, 5000, 8000, 13000, 20000, and 50000. We found that clustering quality was very overall, but we observed a cluster being occasionally lost for sample sizes 1000, 2000, as well as 13000.

Our best results were obtained for the sample size of 8000. The results are displayed in Figure 18, and as we can observe, very few points are eventually misclassified in the dark green cluster.

Silhouette score and WSSSE were 0.455 and 54501 respectively. Despite this being our best results yet, silhouette score and WSSSE were not the highest and lowest, respectively, among tests with 5 clusters, which we attribute to the fact that sometimes was misclassifying points, especially in the center-right small cluster, can produce smaller clusters on average, and therefore better metric results.

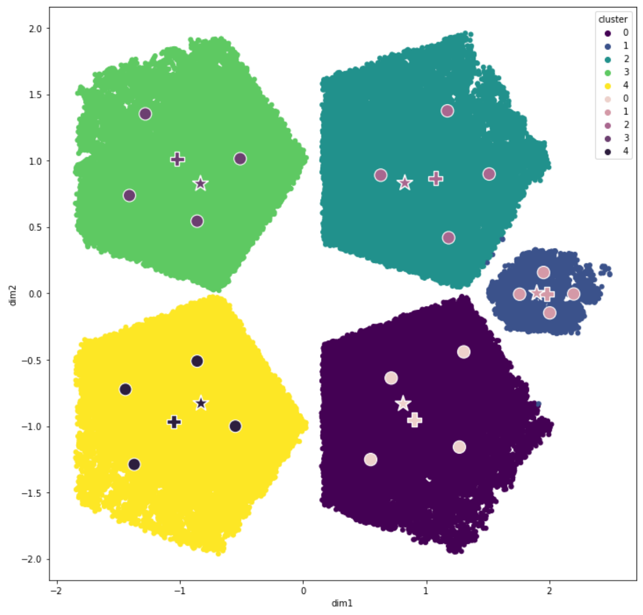


Figure 18: Best clustering quality results.

5.3 Scalability

So far, we execute the algorithm and retrieved a really good result. But, in the Big Data era, one of the most important features for the algorithms is scalability. Till now we tested our algorithm regarding the correctness, but we did not try to scale it. For this reason, we ran an experiment with different sample sizes and the number of partitions. Moreover, we set the remaining variables as follow:

- 5 clusters.
- 4 representatives.
- 0.5 shrinking factor.
- 1692400 total points - a dataset with 16924 points with x100 cardinality.

The tests have executed a machine that had the following specifications:

1. 24 Gb RAM
2. Ryzen 5 1600X with 6 cores and 12 threads

Because our experiments were tested on a 12-thread machine, we expected running times to get smaller as we approached the number of partitions closer to 12. We tested for a variety of partitions such as 1, 2, 4, 6, and 12 partitions, as well as sample sizes of 50000, 100000, and 200000 points.
From our experiments, using only 1 partition we had a much worse performance than using 2 partitions or more. Our best results were obtained using 12 partitions, albeit not much better than when using 6 partitions. The full work of our experiments is shown in Figure 19 and Table 2.

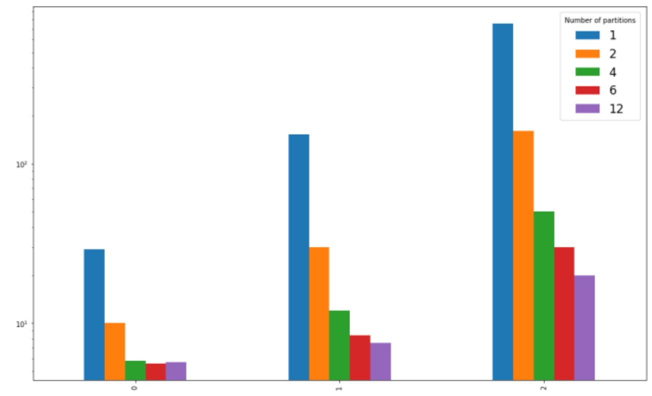


Figure 19: Scalability – Results for different sample sizes and partitions. Logarithmic scale.

Sample size	1 partition	2 partitions	4 partitions	6 partitions	12 partitions
50000	29 sec	10 sec	5.8 sec	5.6 sec	5.7 sec
100000	153 sec	30 sec	12 sec	8.4 sec	7.5 sec
200000	759 sec	160 sec	50 sec	30 sec	20 sec

Table 2: Scalability - Execution times for different sample sizes and a number of partitions.

5.4 K-Means & Agglomerative

One of the requirements of our project was to compare the CURE algorithm with a K-Means which after uses a hierarchical algorithm. The algorithm procedure was introduced in section 3, now we will experiment with this algorithm and we will compare it with the CURE.

First, we have to find the optional variables for the K-means. For that reason, we experiment with a different number of initial clusters and a different number of final clusters. Therefore, we set our initial variables as follow:

- We used from the initial 169240 total points only a dataset with 16924 points and x10 cardinality.
- We finalized the maximum K-Means iterations up to 100.

For our experiments, we used each time a different number of a 5, 20, 50, 100 as the “initial clusters” variable and 2, 5, and 10 for the “final clusters” variable.
The performance of the agglomerative algorithm on top of the K-Means predicted clusters was very good, and better than the performance of the CURE algorithm, which we attribute to the power of agglomeration.
Consequently, silhouette scores and the sum of squares within errors were very good, and for initial clusters higher than 20 and final clusters equal to 5 we observed a perfect fit to the 5 clusters of the data, as depicted in Figures 20 and 21.

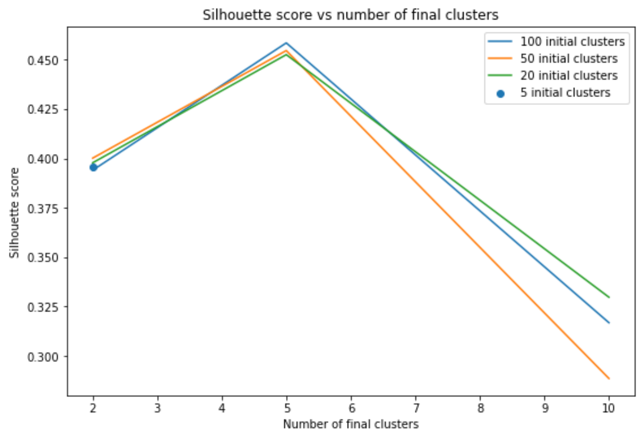


Figure 20: Silhouette score for Agglomerative K-Means with a different number of final clusters.

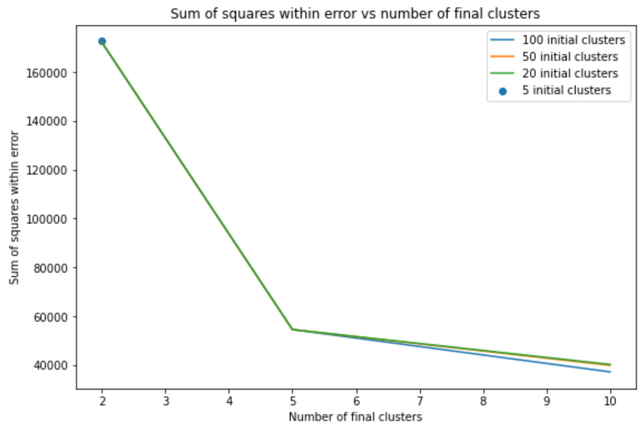


Figure 21: WSSSE for Agglomerative K-Means with a different number of final clusters.

What we found was very impressive. Using the Agglomerative algorithm on 50 K-Means predicted clusters, for 5 final clusters, can perfectly classify all points in the data. This means that it outperforms even the CURE algorithm we described previously. The 5 perfects clusters are described in Figure 22.

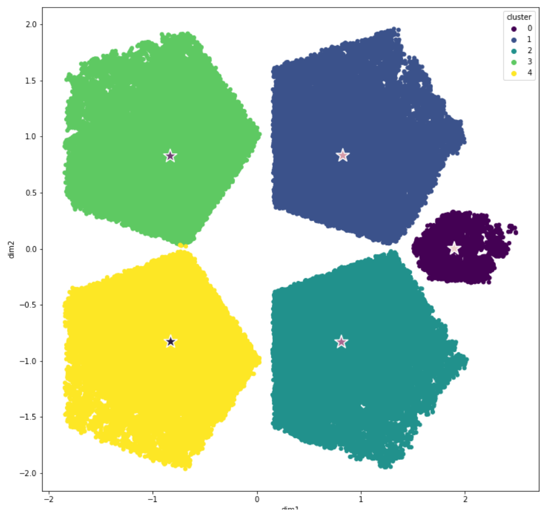


Figure 22: Clustering results with Agglomerative K-means.

5.5 CURE compared with Agglomerative K-means.

We implemented the Agglomerative K-means in order to compare it with the CURE algorithm. The question is, how do we compare 2 algorithms? In what kind of aspects can they be compared? We will compare their scalability, their time to compute, their correctness, and their metrics, i.e., Silhouette and WSSSE.

Firstly, it is easier to compute the algorithms total time execution and through that, their scalability. In Figure 23, we describe the experiments we executed using the CURE algorithm and Agglomerative K-Means on two different datasets and with two different numbers of threads. The cores we describe in the fires are the threads we let the spark use. The experiments were to let the algorithms use 4 threads or only 1 thread, in each experiment. The first dataset has more than 5.000 points and the second dataset is larger because it has almost 17.000 points, as discussed in section 2. So, the second dataset needed more time to be computed in most cases. Overall, we see in Figure 23 that the K-Means outperform the CURE, in most cases with a big difference. For example, clustering on the second dataset, the CURE needs almost two times more time to compute the clusters than K-Means. It is very questionable why when 4 threads were used on the first dataset, the CURE algorithm outperformed the K-Means, while using only one thread, in the same dataset, the K-means outperforms the CURE.

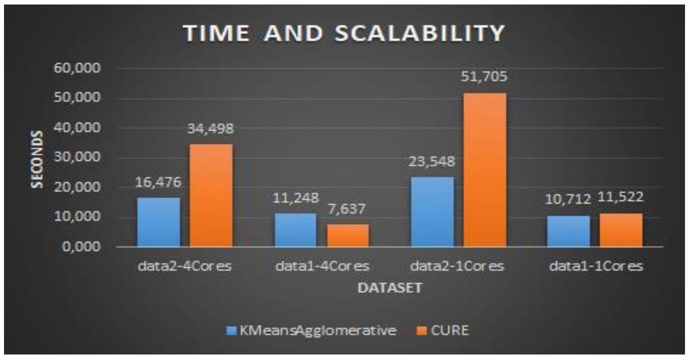


Figure 23: Time and Scalability for K-Means and CURE comparison

We discussed the CURE scalability when we described Figure 19. As for the scalability of the Agglomerative K-Means, Figure 23 showed us that even when we used a different number of threads, the algorithm was faster. The best results were performed on the second dataset where the K-Means were computed two times faster than the Cure, the clusters. But is that mean the K-Means is two times faster than CURE is the peak of the algorithm performance? To test that assumption we executed both algorithms with 4 threads on our largest dataset with almost 1.7 million points. The results of that experiment are shown in Figure 24 and they were insightful. The K-Means computed the clusters 80 times faster than the CURE Algorithm, which means the more we increase the data the more times the CURE will need to compute the clusters from K-Means.

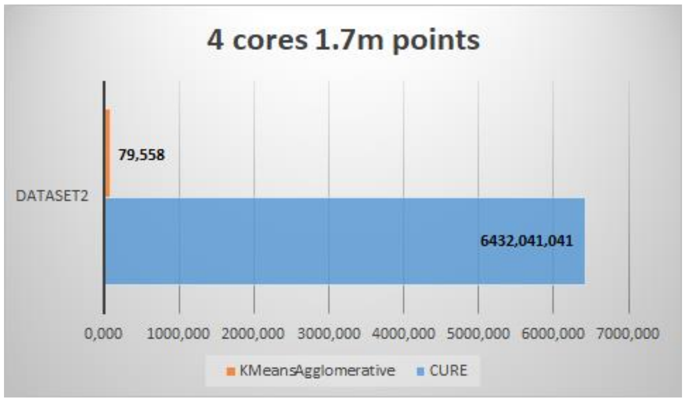


Figure 24: The difference in time between CURE and K-Means on the full dataset

We evaluated the clustering techniques in terms of scalability and execution time, so now, we have to compare their metrics and find the best metrics for each technique. The first thing we examined was the Silhouette metric. In Figure 25, the results of our experiments are shown. We tested Silhouette for a number of clusters and we recorded the score of each algorithm performed. Both algorithms increase the silhouette score to a number of 5 clusters and then both decrease. K-Means starts with a high silhouette score and reaches a higher score than CURE, which means that as clustering technique it performs better.

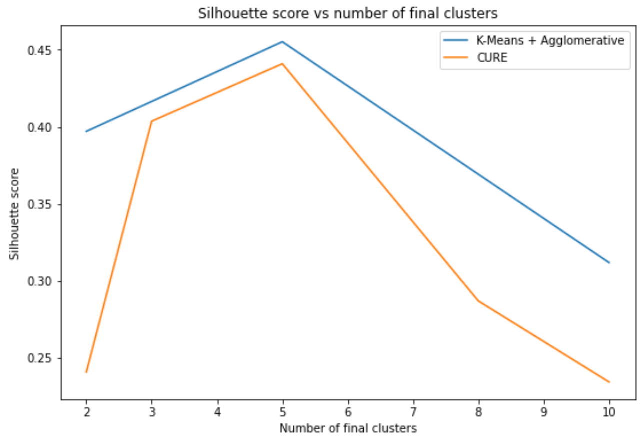


Figure 25: Silhouette comparison for both algorithms

After the Silhouette score, we examined the WSSSE score. In Figure 26 we present our findings after the experiments with the WSSSE on both clustering algorithms. For 2 clusters both algorithms have high WSSSE scores, with the CURE's be higher than the K-means. For 3 and 4 clusters the CURE abruptly outperforms the K-Means, while the second steadily decreases the WSSSE score. On 5 clusters, both algorithms line intersects each other because they produced almost the same score. After that point, the K-Means steadily outperform the CURE algorithm with a small difference.

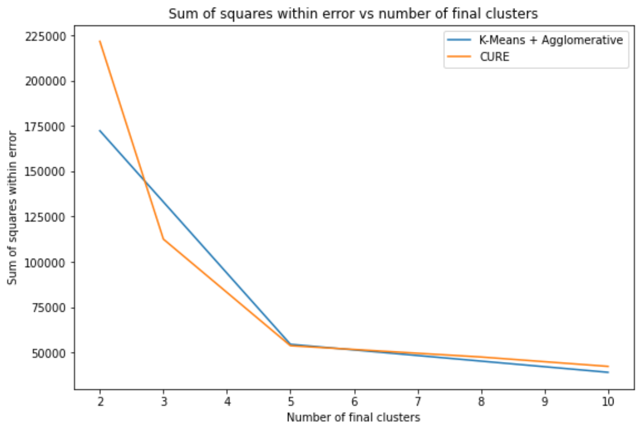


Figure 26: WSSSE comparison for both algorithms

5.6 Filtering outliers

CURE may not be the best algorithm so far, but it can handle the outliers. The implementation of the CURE algorithm lets us choose if we want to detect the outliers or not. To test this feature, we injected in the dataset some outliers, as we discussed in section 2. For that reason, we experiment with a different number of clusters each time. Firstly we had to initialize the CURE's variables such as:

- 4 representatives
- 0.5 shrinking factor
- 5 partitions
- From 169240 total points, we used a dataset with only 16924 points which had x10 cardinality, a sample size of 13428, and a sample ratio of 0.08.

The number of different clusters we experiment with was 2, 3, 5, 8, and 10 clusters. When filtering outliers, the results were good, with the number of 5 clusters being a very good fit, due to the fact that we observed 5 clusters in our dataset as shown in Figure 27. In contradiction when we set the filtering outliers option off we observed that certain clusters can be lost due to a lack in the number of representatives in those clusters.

To reach these results we had to experiment with all the clusters we mentioned before. For example for a number of 2 clusters, we never managed to retrieve 2 clusters, but instead regularly got back 1 cluster only. For this reason, the silhouette score, in this case, could not be calculated. When not filtering outliers for 2 clusters we get back only 1 cluster as shown in Figure 29.

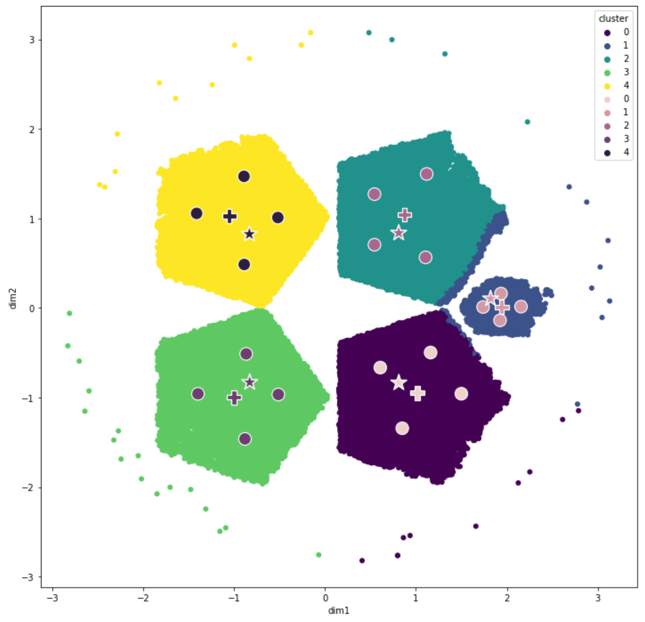


Figure 27: Clustering with filtering outliers and 5 clusters

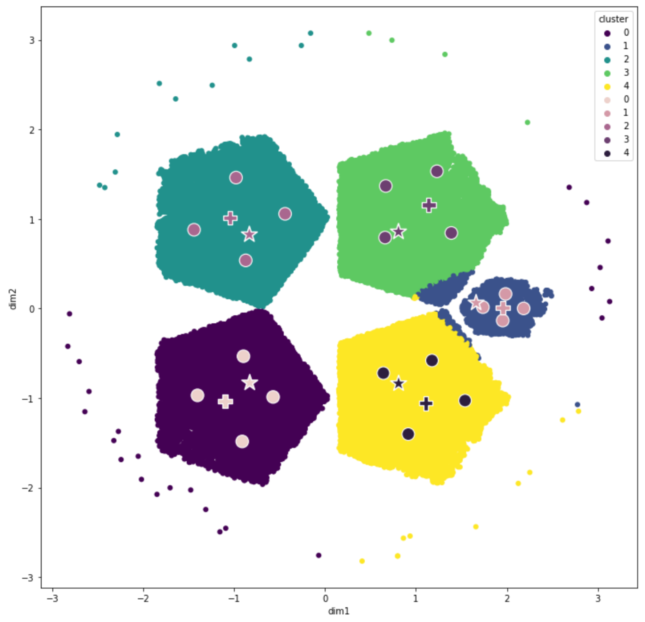


Figure 28: Clustering results without filter the outliers and 5 clusters

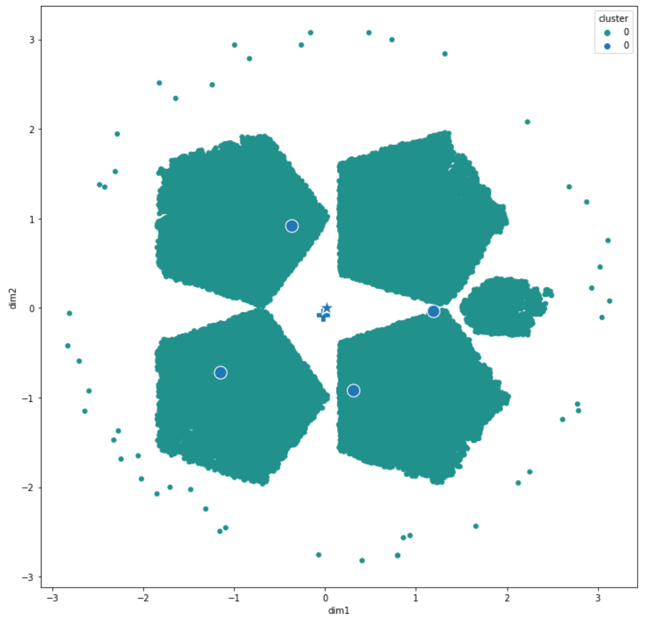


Figure 29: Only 1 cluster when we initialized 2 clusters without an outlier filter.

6 Conclusion

In this work, a comparison study has been performed between the CURE algorithm and Agglomerative K-Means algorithm according to their capabilities to scale, perform correct results, and metrics evaluation. In all experiments the Agglomerative K-Means we implemented outperformed the CURE algorithm in all metrics and aspects. More specifically, the larger the dataset is, the slower CURE creates the clusters, while K-Means increases the times it is faster than CURE. While both algorithms scale, K-Means performs better. While K-Means reaches higher Silhouette scores the CURE outperforms the K-Means in WSSSE score for a small number of clusters. However, with CURE's bad evaluations we have to consider the power of this algorithm, which specifically lies in its distributed nature over the agglomerative clustering algorithm and improved clustering quality over regular K-Means. There is a lot of research, nowadays, that shows the CURE algorithm is still a trending topic and each time it gets better [2].

REFERENCES

[1] Guha, Sudipto, Rajeev Rastogi, and Kyuseok Shim. "CURE: An efficient clustering algorithm for large databases." ACM Sigmod record 27.2 (1998): 73-84.

[2] Kumble, Nikita, and Vandan Tewari. "Improved CURE Clustering Algorithm using Shared Nearest Neighbour Technique." International Journal 9.2 (2021).