A CURE CLUSTERING ALGORITHM IMPLEMENTATION IN SCALA WITH SPARK

Mining of Massive Datasets
MSc Data & Web Science, AUTH, 2021

SCHOOL OF INFORMATICS
Data & Web Science MSc Program

Vasileios Moschopoulos 59
Georgios Michoulis 82

THE CURE CLUSTERING ALGORITHM

Clustering Using REpresentatives

Clustering algorithm

Can handle non-normally distributed data by using m representatives points for each cluster and is able to identify complex cluster shapes

Offers a distributed solution for clustering very large datasets and uses only a sample of size *n* from the original dataset

The sample can be further divided into *p* different partitions to be individually processed into clusters, allowing for massive parallelism

The algorithm begins by setting every point as a different cluster with one representative

The clusters having the closest distance between one of their representatives are merged into a new cluster until there are *k* clusters

Representatives are selected each time as the most dispersed points from the cluster mean

With every new merged cluster, selected representative points are shrunk towards the center of the cluster using a specified shrinking factor *alpha*

After each partition has collected *k* clusters, the clusters are collected and merged until there are *k* clusters, or all clusters have *m* representatives

THE CURE DATA STRUCTURES

CURE uses two data structures

- A MinHeap for storing the clusters
- A KD-tree for storing the representatives of each cluster

MinHeap

- Clusters are sorted in a list the form of a binary tree in increasing order based on the distance from their closest clusters
- The cluster having the smallest distance from its closest cluster is always at the top of the heap

KD-tree

- The KD-tree (K-Dimensional tree) is a binary tree for more than one dimensions
- Each level l of the KD-tree sorts points into left and right nodes using the $l \bmod k$ dimension for comparison
- The KD-tree allows for quickly retrieving the closest representative of a cluster belonging to another cluster

OUR CURE IMPLEMENTATION

- Scala with Spark
- Parameters
  - Number of clusters > 1
  - Number of representatives > 0
  - 0 < shrinking factor < 1
  - 0 < number of partitions < 100
  - Input file/folder
  - 0 < sampling ratio < 1
  - removeOutliers boolean for filtering outliers
- Three main methods
  - Main – top level
  - Cluster – intermediate level
  - ComputePartitionClusters – low level
- After the algorithm ends we save the following data in a new folder with a timestamp
  - Data points with their predicted clusters
  - Representative points
  - Centroids of sample clusters
  - Time taken for execution

Algorithm 1 CURE

```
1:  function MAIN
2:      Read the data points
3:      Pick a sample and repartition it
4:      Map each sample point to a cluster
5:      Map each sample point to a cluster
6:      Create the clusters at each partition : Cluster() function
7:      Collect the clusters of all partitions
8:      Retrieve the representatives of all collected clusters
9:      Merge collected clusters and clusters short of m representative points
10:     Label all data points using the representatives of the final clusters
11:     Return the labeled points, the final representatives, and the final sample cluster means
12:
13: function CLUSTER
14:     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
15:     Create the KD − Tree using all points as representatives
16:     Create the MinHeap containing all clusters, using all points as separate clusters
17:     If removeOutliers is true partially cluster points until there 2 ∗ k clusters and remove
        clusters with less than m representatives : ComputePartitionClusters() function
18:     Continue / Fully cluster all clusters : ComputePartitionClusters() function
19:     Return all clusters from the MinHeap
20:
21: function COMPUTEPARTITIONCLUSTERS
22:     while MinHeap.size > k or MinHeap.size > 2 * k do
23:         Merge the nearest clusters
24:         Remove their representatives from the KD − Tree
25:         Remove them from the MinHeap
26:         Insert the new merged cluster in the MinHeap
```

DATASET PREPROCESSING

Dataset characteristics

- The dataset comprised 2 different folders containing 100 duplicate files with several two-dimensional points
- Each file in the first dataset contained 5725 points, while in the second dataset each file contained 16924 points
- The data was shaped in the form of 5 pentagons, 4 large ones and a smaller one

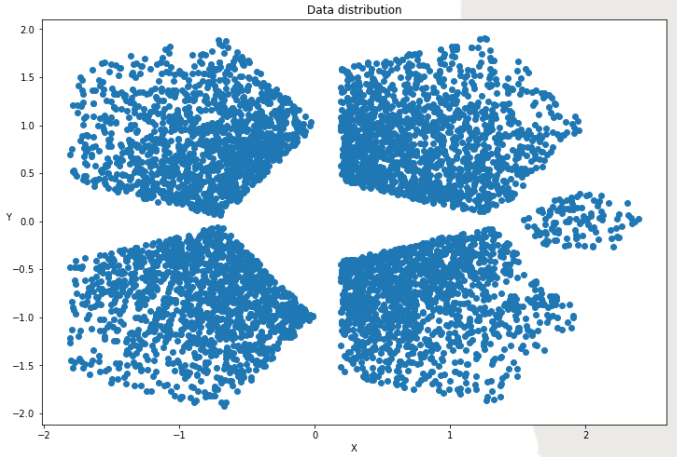
Increasing cardinality

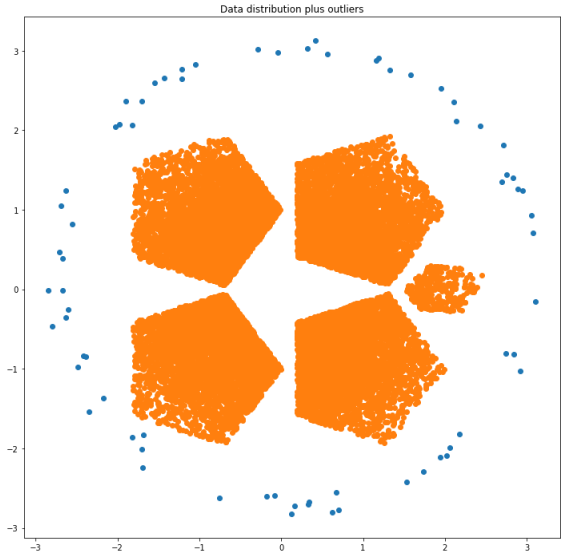
- In order to test the scalability and correctness of our implementation we increased the cardinality of both datasets by factors of 10, 30 and 100.
- How? We duplicated the points of each dataset and added an infinitesimally small amount of noise, using a uniform distribution ranging between -0.04 and 0.04.

Outlier injection

- To test our devised method for handling outliers we needed to add outliers in our dataset
- We created duplicates of the different cardinality datasets
- We added outliers to each duplicate in the form of a circle perimeter with a radius of 2.9, centered around the mean of all points.

Original dataset of cardinality 5725 with the shape of the five pentagons being clearly visible

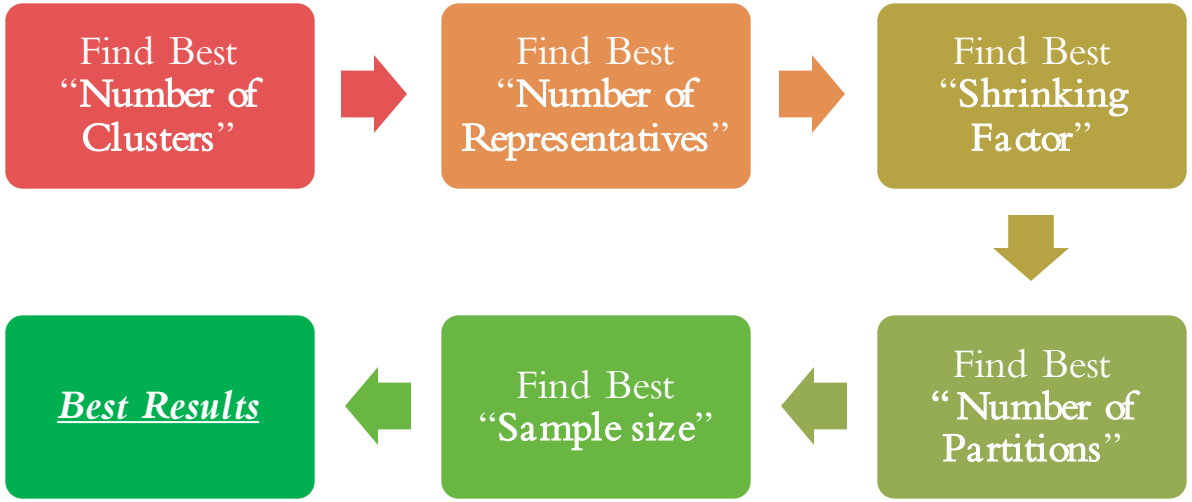




Original dataset of cardinality 16924 with outliers added in the form of a circle perimeter

TESTING CORRECTNESS

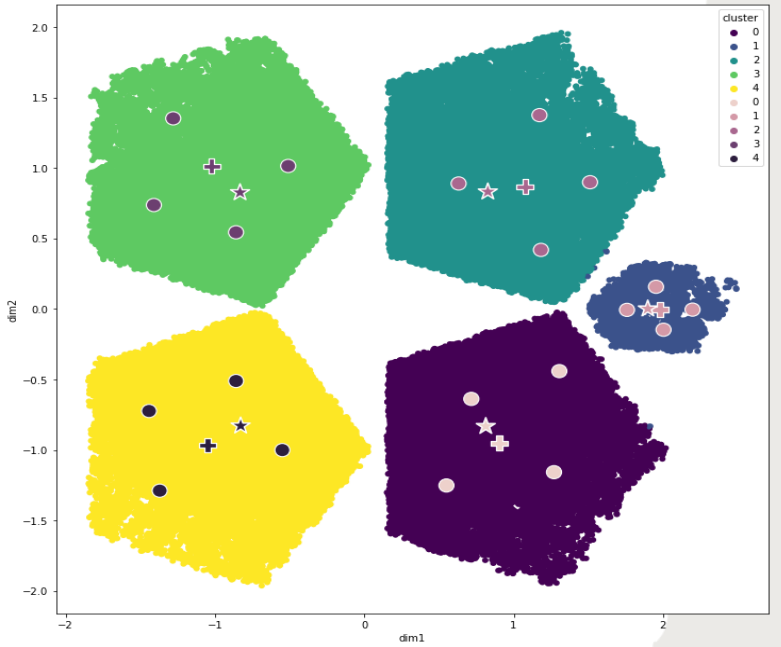
- In order to find the best clustering formula for CURE we had to examine 5 steps:



Evaluated each step by Silhouette score, Sum of Squares Within error and visualization.

The best metrics were:

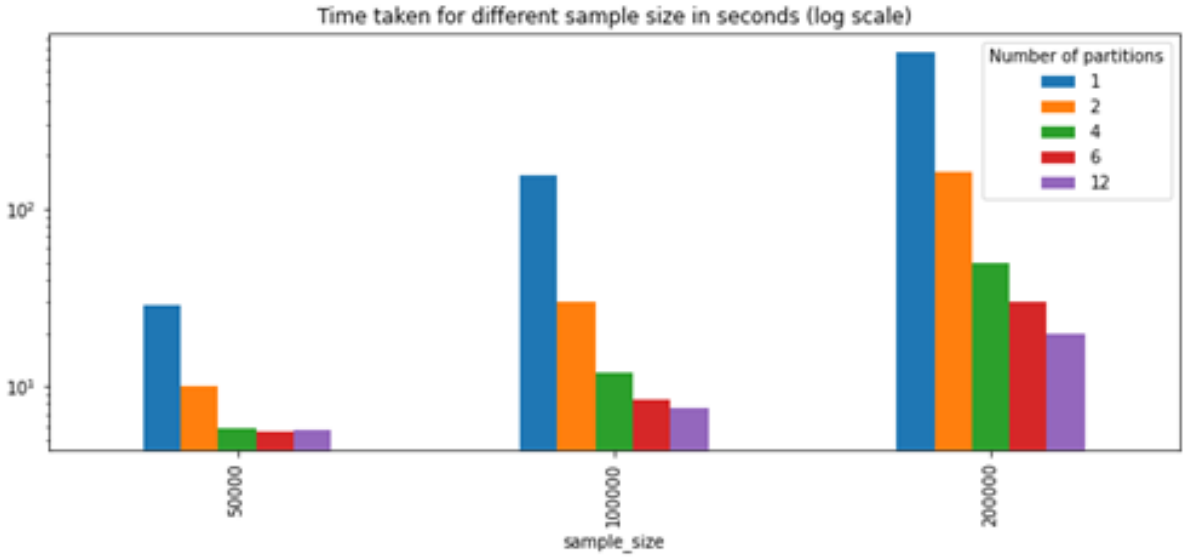
- 5 clusters
- 4 representatives
- 0.5 shrinking factor
- 5 partitions
- sample size 8035



Silhouette: 0,454664
WSSSE: 54.501,904594

TESTING SCALABILITY

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



To test scalability we ran different experiments

- Used a Ryzen 5 1600X, 6-core, 12-threads, with 24GB RAM
- Tested for different partitions for sample sizes such as 50000, 100000 and 200000 points.
- Best results were obtained using 12 partitions

HANDLING OUTLIERS

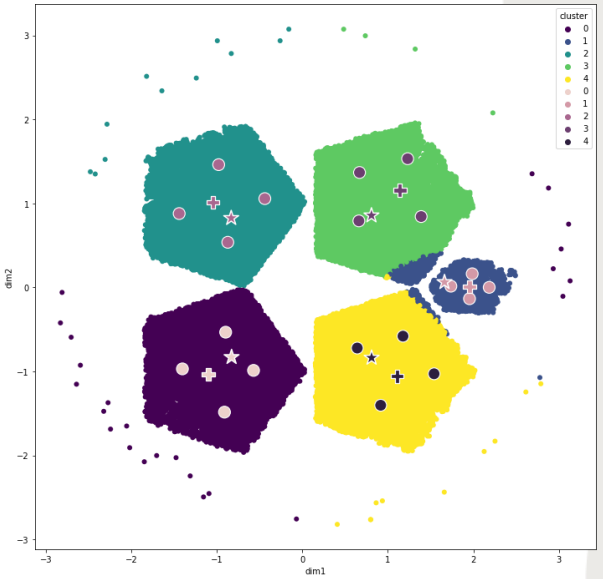
For handling outliers, we used a *removeOutliers* boolean property

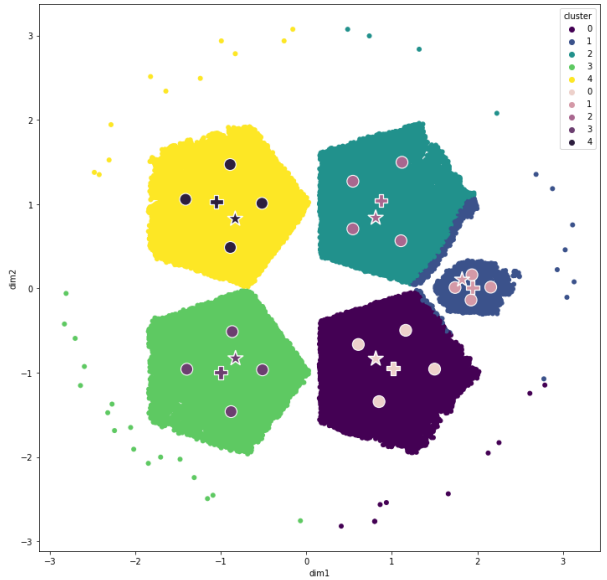
- When *removeOutliers* is true, points at each partition are partially clustered, until there are $2 * k$ clusters
- Clusters having less than m representatives are identified and removed
- The process continues by further merging partial clusters until there are k clusters at each partition

Results

- Experiments with 2, 3, 5, 8 and 10 clusters
- When filtering outliers, the results were very good
- When not filtering outliers, certain clusters could be lost due to lack in number of representatives

When not filtering outliers for 8 clusters we get back only 5 clusters.





When filtering outliers for 5 clusters we get a very good fit.

IMPLEMENTING A K-MEANS & AVERAGE-LINKAGE AGGLOMERATIVE SOLUTION FOR COMPARISON

A conjoined implementation for comparison

- Conjoined implementation with K-Means and simple average-linkage Agglomerative clustering on the predicted K-Means clusters
- Implemented in Scala
- We begin with a higher number of *initialClusters* clusters for the K-Means algorithm, and then agglomerate the predicted K-Means clusters into *finalClusters* clusters

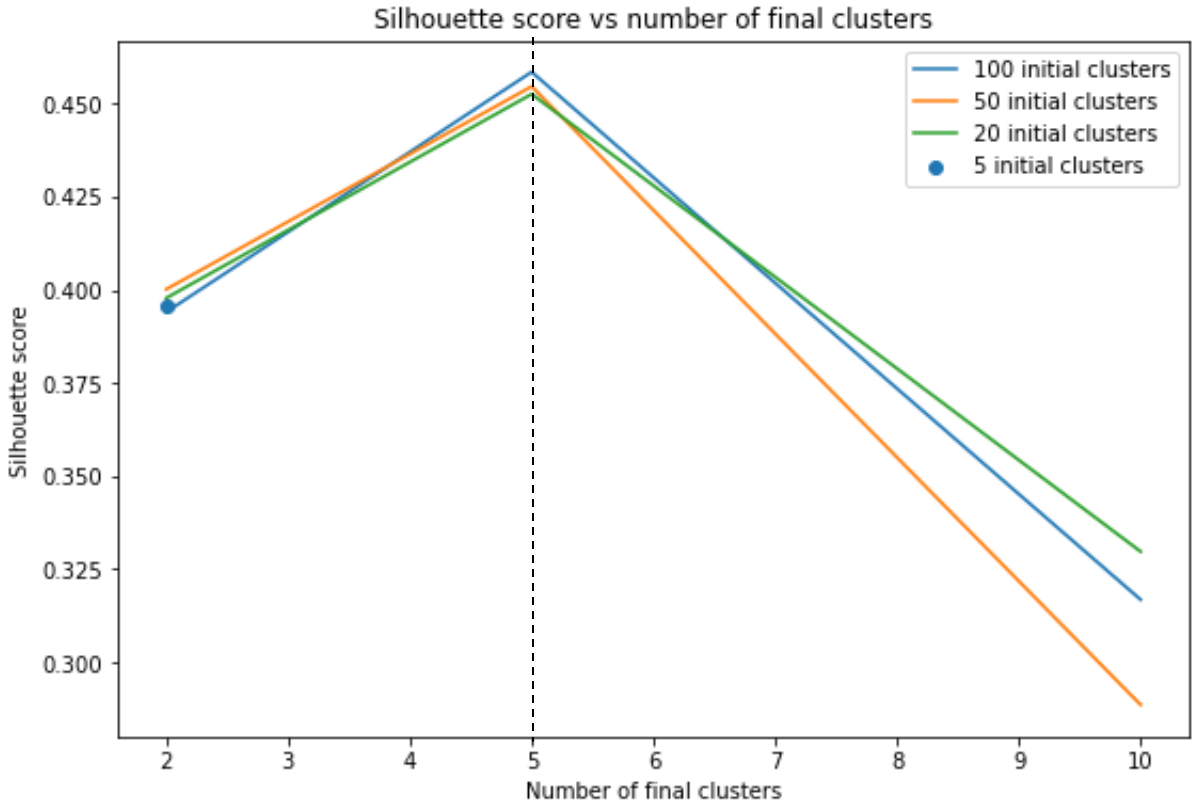
K-Means

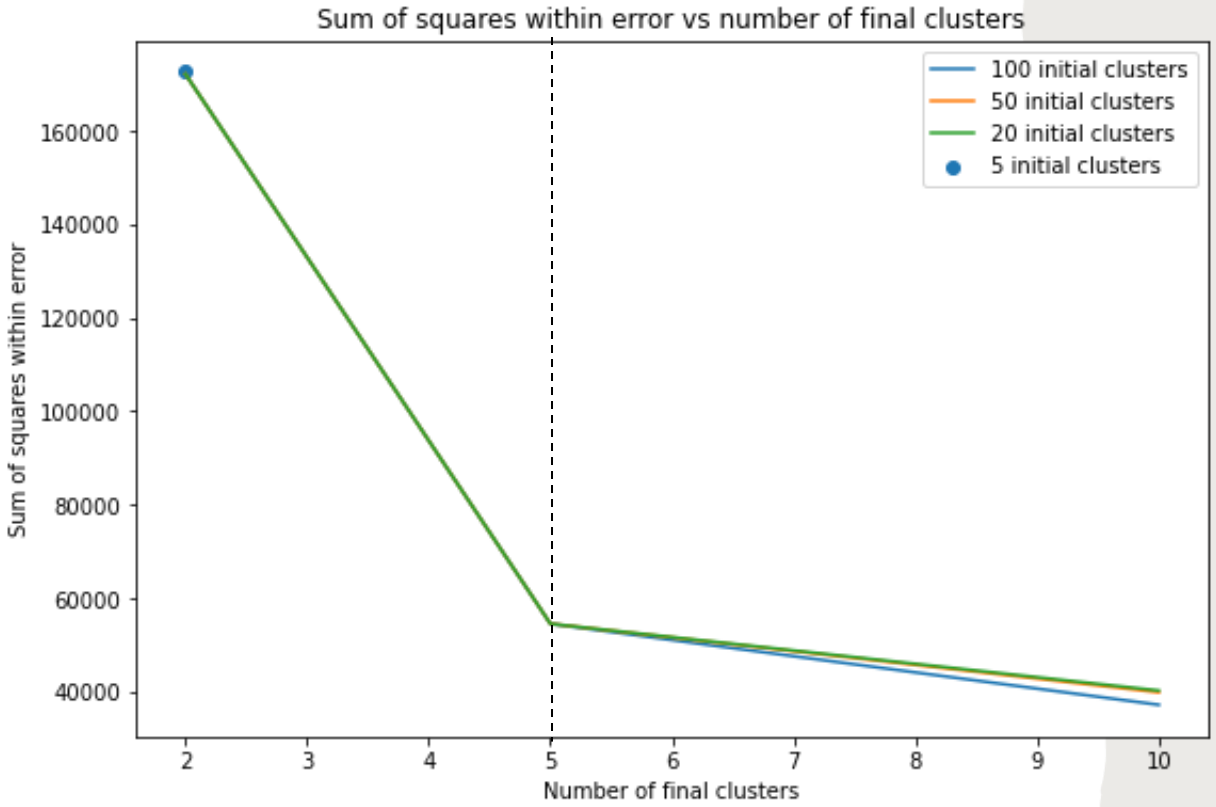
- One of the most commonly used clustering algorithms
- Used K-Means from Spark MLLib
- *maxIterations* = 100
- *initializationMode* = "k-means||"

Agglomerative clustering

- Very powerful but prohibitive cost
- Average-linkage flavor selected due to its simplicity
- Very simple implementation
  1. Provide the initial clusters to the algorithm
  2. Calculate the centroid of each cluster
  3. Calculate pairwise distances using the means
  4. Join at each step the clusters having the closest distance

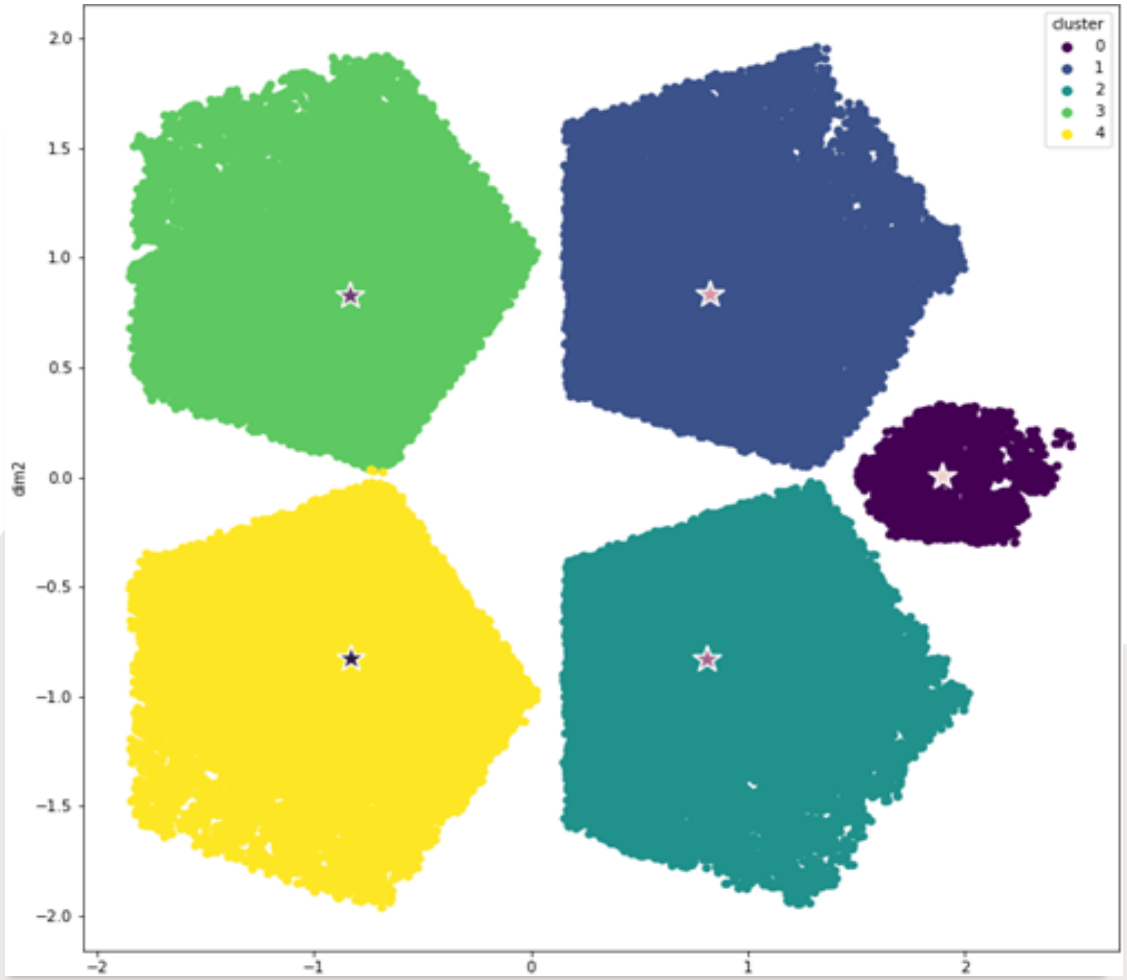
RESULTS K-MEANS WITH AGGLOMERATIVE

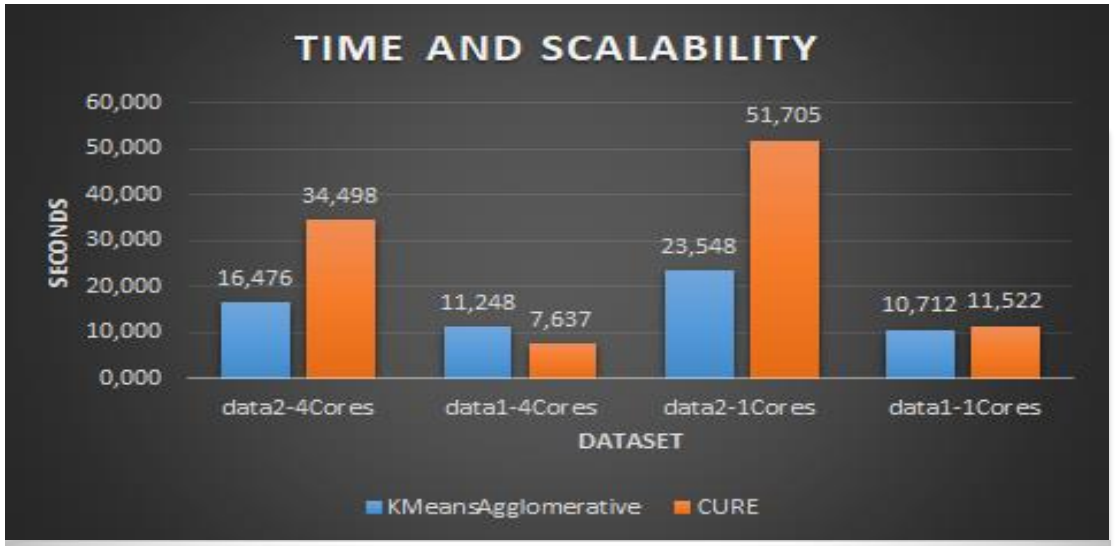


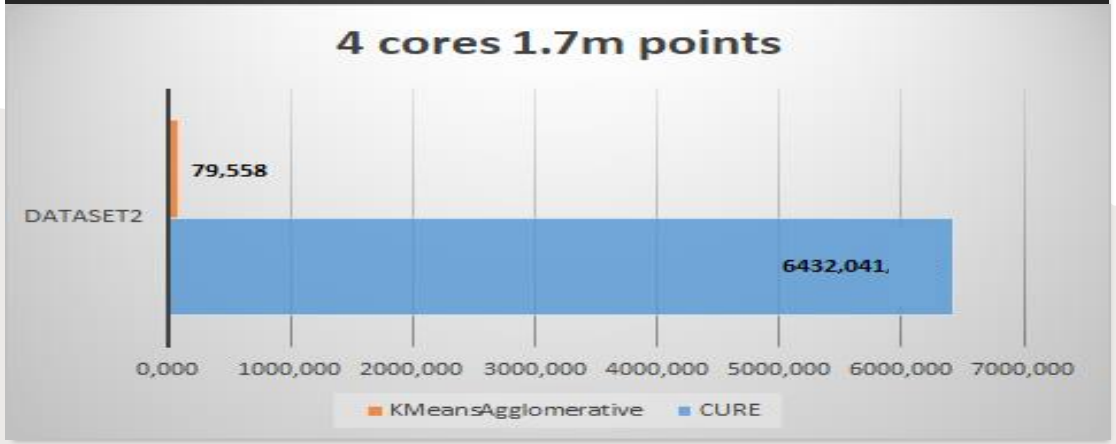


- In our experiments we used 5, 20, 50, 100 initial clusters and 2, 5 and 10 final clusters.
- The performance of the agglomerative algorithm on top of the K-Means predicted clusters was very good, and better than the performance of the CURE algorithm

- Silhouette scores and sum of squares within errors were very good and for initial clusters higher than 20
- Final clusters equal to 5 we observed a perfect fit to the 5 clusters of the data.



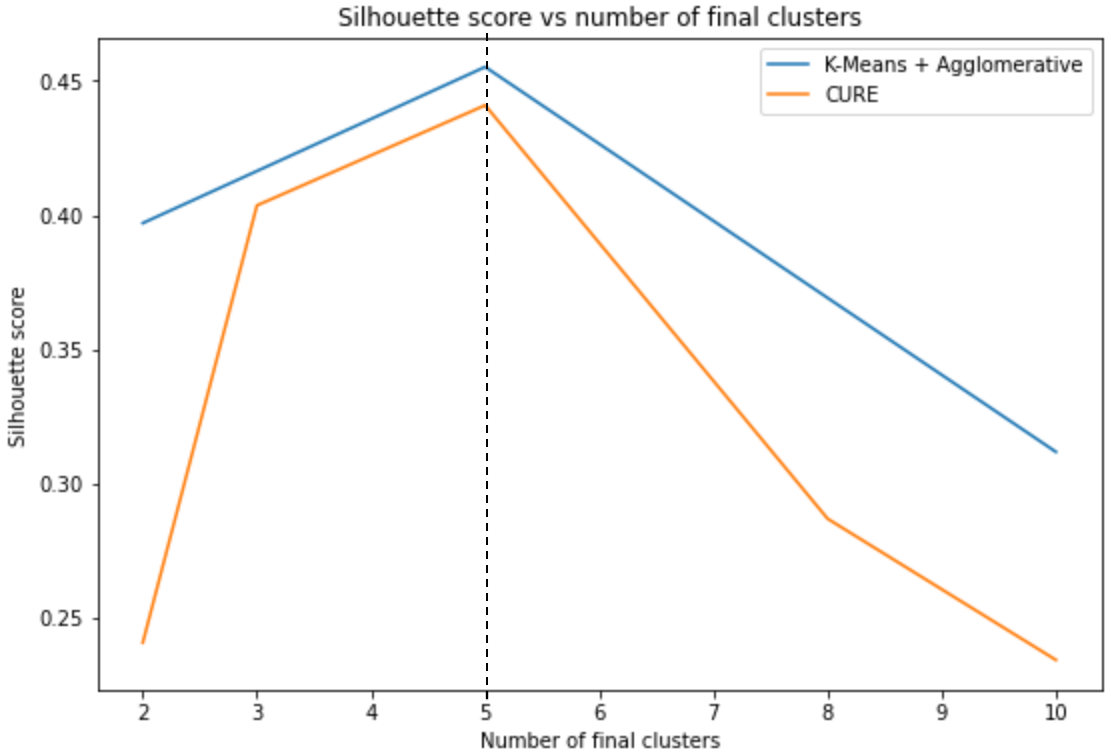


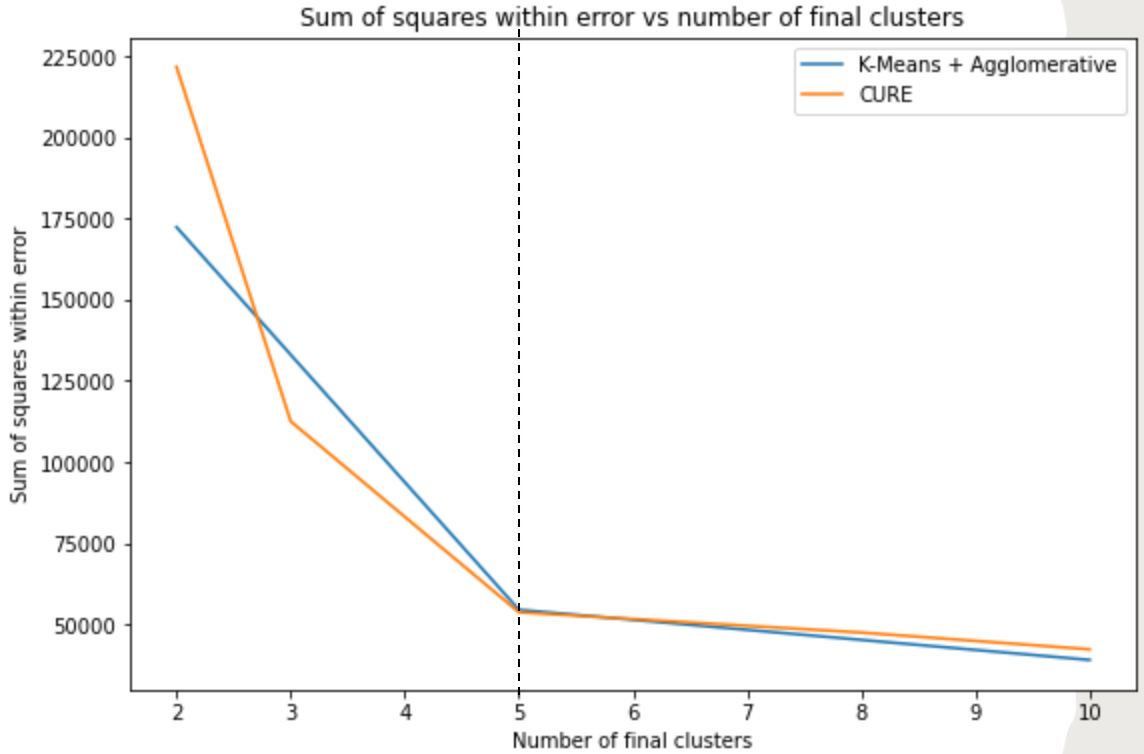


RESULTS AND COMPARISON AGGLOMERATIVE- CURE

- 169240 total points - dataset with 16924 points with x10 cardinality
- 100 maximum K-Means iterations

SILHOUETTE AND WSSE COMPARISON CURE – AGGLOMERATIVE K-MEANS





CONCLUSION

- K-Means with Agglomerative was better in all metrics
- However, despite CURE's relatively worse results we have to consider the power of the algorithm, which specifically lies in:
  - Its distributed nature over the Agglomerative clustering algorithm
  - Its improved clustering quality over regular K-Means
- We conclude CURE is better for very large datasets
- For smaller datasets, combining clustering algorithms is a very strong solution that is rarely discussed in literature, especially stacking ensembles, such as our K-Means & Agglomerative implementation.
- Due to the recent development of big data technologies, CURE has resurfaced as a potent clustering algorithm, suitable for distributed analytics applications. Due to this, we conclude CURE is a trending clustering algorithm, and with recent research expansions as well.

QUESTIONS?