Improved CURE algorithm for big data using Apache Spark

Ioannis Mavroudopoulos, 417[*]
mavroudo@csd.auth.gr
Aristotle University of Thessaloniki

Styliani Kyrama, 76[*]
kyrastyl@csd.auth.gr
Aristotle University of Thessaloniki

ABSTRACT

Clustering is a very important task which is considered as an unsupervised learning process. CURE is an agglomerative hierarchical clustering algorithm that overcomes many of the basic drawbacks of other existing clustering algorithms, such as the spherical shape of clusters. We developed CURE in Scala with Apache Spark to ensure the scalability in big data scenarios and its parallelization ability. Besides implementing the algorithm with its final pass parallelized, we also extended it to be able to efficiently detect outliers in the data. To test both the scalability and the quality of the results and detection of outliers, we performed a series of experiments presented in section 4.

ACM Reference Format:
Ioannis Mavroudopoulos, 417 and Styliani Kyrama, 76. 2021. Improved CURE algorithm for big data using Apache Spark. In *Proceedings of ACM Conference (Conference'17)*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In everyday terms, clustering refers to the assignment of objects into groups with similar characteristics. When it comes to data and data mining, clustering refers to a very important task, which is considered as an unsupervised learning process. Through clustering, we can discover important patterns and anomalies, find similarities and differences between the underlying entities of our data, but also group them, for the purpose of further analysis. More specifically, given n data points in a d-dimensional space, where each dimension represents a characteristic, the clustering task can be defined as the process of partitioning these data points into k clusters. The main concern of the clustering process is that the data within a cluster have to be more similar to other data in the same cluster, and less similar to data belonging to a different one.

Various algorithms have been implemented and used for the clustering task, and they can be divided into several different categories such as partition-based, density-based, hierarchical, etc. Each one of the existing algorithms that fall into one of the above categories has its own advantages and disadvantages. For example, the K-means algorithm is simple to implement and scales into a lot of data but is biased to more spherical clusters. On the other hand, since DBSCAN is a density-based algorithm, it overcomes the problem with the shape of clusters, detecting arbitrarily shaped clusters, but has difficulty detecting clusters of dissimilar densities.

CURE *(Clustering Using Representatives)* is a hierarchical clustering algorithm proposed by [1]. It adopts a middle ground between the centroid-based and all point extremes, and has some features that contribute to its superiority over other similar clustering algorithms. CURE, as also its name implies, uses a set of points called representatives in order to perform the clustering; instead of using only one point for each cluster, the centroid one, a fixed number of c elements for each cluster are selected, which reflect its shape and geometry. Contrary to K-means which as mentioned before works best with spherical clusters, CURE can detect correctly arbitrarily shaped clusters. Last, the shrinking factor alpha that the algorithm uses, make it robust to the presence of outliers.

In this work, we implement the CURE cluster algorithm to Scala, using Apache Spark. In addition, we parallelized the final step of the algorithm, in which each individual point is assigned to a single cluster.

The rest of this report is structured as follows. In the section 2, we present some related research work done, mainly aiming at improving the algorithm. The section 3 is dedicated to the presentation of the CURE algorithm and the pseudocode of the algorithm implemented in Spark, as well as the handling of extreme values. The experiments are illustrated and analyzed in detail in the section 4. A discussion about the results and some final conclusions are presented in section 5.

2 RELATED WORK

As mentioned above, CURE is an agglomerative hierarchical clustering algorithm, that overcomes many of the basic drawbacks of other existing clustering algorithms. Despite the fact that CURE is already an algorithm that can be used for large datasets, many researchers have proposed algorithms that are based on CURE but trying to improve it from different angles.

Authors in [4] proposed a generalized parallelization algorithm based on CURE clustering, in order to improve the efficiency and increase the performance of clustering. More specifically, they designed a parallel algorithm for CURE by combining data parallelism and shared memory systems.

In [3] authors implemented the CURE algorithm over distributed environment using Hadoop MapReduce Model, in order to handle massive amount of data and perform clustering in less amount of time.

A very novel technique which proposed recently is the one in [2], in which they modified CURE algorithm to use neighbors of points instead of representative points, to form the clusters. The representative points which share the same neighbourhood are put together in the same clusters, and this allows generating clusters of

[*]Both authors contributed equally to this research.

Conference'17, July 2017, Washington, DC, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

non-globular shaped clusters. The main advantage of this method is that it does not get affected by the shape of the clusters.

3 CURE IMPLEMENTATION ON SPARK

In this section, we present the implementation of CURE [1] in Scala with Apache Spark. CURE is a hierarchical clustering algorithm, who can recognize clusters with arbitrarily shapes and is robust to the presence of outliers (as we will show in Section 3.1, can find outliers as by-product). The algorithm has linear space complexity and $O(n^2)$ time complexity. In order to be applicable in big data, CURE is executed in a sample of the data and then use representatives for each cluster for the full clustering. In the final pass of the data, we use Apache Spark to ensure the scalability of this method in big data scenarios.

3.1 Clustering Algorithm

Algorithm 1 Algorithm Outline

```
1: **Input:** points,k,$\alpha$,c
2: clusters $\leftarrow$ each point is a cluster
3: minHeap: MinHeap $\leftarrow$ initlialize
4: **for** every cluster $C_1$ **do**
5:     $C_2 \leftarrow$ closest cluster to $C_1$
6:     minHeap.add($C_1, C_2, \textbf{dist}(C_1, C_2)$)
7: **while** clusters.size > k **do**
8:     **if** clusters.size = $\lceil points.size/3 \rceil$ **then**
9:         remove all clusters with **points** <= 2
10:    $(c_1, c_2, dist) \leftarrow$ minHeap.extract_min()
11:    clusters.remove($c_1, c_2$)
12:    $c_{new} \leftarrow \textbf{merge}(c_1, c_2, c, \alpha)$
13:    **update** closest clusters
14:    minHeap.add($c_{new}$)
15:    clusters.add($c_{new}$)
16: **return** clusters
```

The outline of CURE algorithm is represented in Algorithm 1. The main object of the algorithm is the cluster, which contains the number of points that belong in this cluster, the representative points and the center of the cluster. The algorithm take as parameters the input points, the number of clusters (k), the Shrink factor (α) and the number of representatives points per cluster (c).

In line 2 each input point is considered as separate cluster, where the set of representatives contains only the point in the cluster. The min heap is initialized in lines 3-6, where for each cluster the distance from its closest cluster is calculated and stored, along with the pair of clusters, in the heap. The points in the heap are arranged in increasing order of the distances. In the original work, they used a K-d tree to faster compute the closest cluster, but through experimentation we show that using the parallelization of Apache Spark was more efficient than maintaining a structure like K-d tree.

The distance between two clusters c_1 and c_2 can be defined as:

$$dist(c_1, c_2) = min\{dist(p, q) | p \in c_1.rep, q \in c_2.rep\}$$

i.e the minimum distance between the representative points of the clusters.

The lines 7-15 is the main loop. In each iteration the closest pair of clusters is merged, until the number of clusters reaches the input parameter k. Even though we ran CURE in a random sample of the dataset, there is still a possibility an outlier point to be chosen. The outlier points effects significantly the performance of the algorithm.To recuse their impact, we implemented a method proposed in [1]. When the size of clusters has reduced to one third, we remove all the clusters that contain only one or two points (lines 8-9).

The pair of closest clusters is extracted from the min heap and the two clusters are merged. In the merge procedure the points from the two clusters are combined, the new center is recalculated and new representatives are chosen in order to effectively capture the shape of the new cluster. The representative points are shrunk towards the center of the new cluster by the fraction α. Once the new cluster is created, the clusters that had as closest cluster one of the merged ones, will have to recompute it. Both merge and update procedures are implemented according to the [1]. At last, the new cluster will be added back to the min heap and the list of clusters (lines 14-15).

Once the size of the clusters reaches the parameters k, the clusters will be returned.

3.2 Final Pass

Apache Spark provides a framework for executing jobs distributed. At the final pass, each point has to be assigned to the cluster with the closest representative. This is a process that can be entirely parallelized. The representatives from all clusters are broadcasted, so all workers will have a copy of this information and then each point is processed separately.

3.3 Outlier Detection

As mentioned in the beginning of this section, CURE can find outlier points as by-product. While each point is assigned to the cluster with the closest representative from it, we also maintain this distance. Then we report as outliers the points that deviates more than n times the standard deviation from the mean distance. The n is parameter and determines how isolated a point has to be in order to be reported as outlier. As we will show in the Evaluation Section, for a adequate number of representatives per cluster, this method performs well.

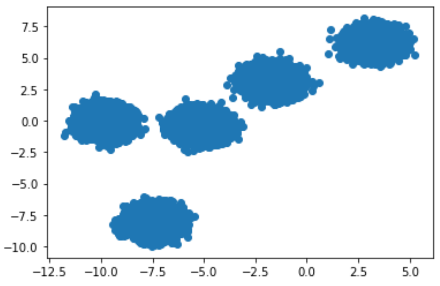
4 EXPERIMENTS

4.1 Datasets

In order to test both the execution time, the correctness, and the scalability of our implementation, we used various datasets.

Both for execution time and scalability test, we needed datasets that will vary regards the number of points containing. For that purpose, we created a simple dataset generator, that creates a synthetic dataset for a custom number of points, k user-defined clusters, as is depicted in Figure 1. The size of the datasets created, expressed in number of points, ranges from 10,000 to 1,000,000 points.

The correctness of our implementation was tested only using the two groups of datasets provided, which differed in the number of points they contained. The first data consisted of 5725 points while the second of 16924. Both sets consisted of a total of 5 clusters,


(a)

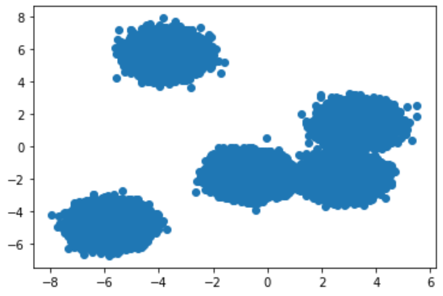

(b)

Figure 1: Two randomly generated datasets, with K = 5

which were structured as shown in Figure 9a. The implementation along with the scripts to generate data are publicly available on [1].

4.2 Experiment on Scalability

In this section are presented and discussed the results of the experiments regarding the scalability of our implementation, by measuring the execution time.

Aiming to extract a more complete picture of the performance of the CURE algorithm, and in particular of our own implementation, we performed runtime testing for different values on three key factors. These key factors are: i) the number of points in the dataset, ii) the sample size obtained for the first phase of the algorithm, in order to extract the representative points for each cluster, and iii) the number of representative points using for each cluster in order to perform the clustering task.

In Table 1 we present the parameters we used in the experiments in this Section, along with their default values, which is used when the other parameters are tested, and the range of their values.

Parameter	Default Value	Range
Sample size	0.025	0.025-0.3
Number of Representatives in Cluster	20	1-100
Shrink Factor α	0.3	0.2-0.7
Number of Points	100000	$5 * 10^3 - 10^6$

Table 1: Parameters used for Time Experiment

[1]https://github.com/KyraStyl/CURE_algorithm_in_Scala_Spark

Observing the plots for the three aforementioned experiments we can easily realize that by increasing the value of one of the above parameters, the execution time also increases.

Number of points: First, we examined the effect of dataset size on the execution time of the algorithm, the results of which are depicted in Figure 2. More specifically, for this experiment, we vary the number of elements, but we set the sample points to a constant number c, with c=300. We decided to limit the sample size because having the sample as a percentage of the number of points then it will constantly increase, and will affect the execution time even more. Thus, having the sample size constant, we observe that the execution time increases linearly with respect to the number of points.

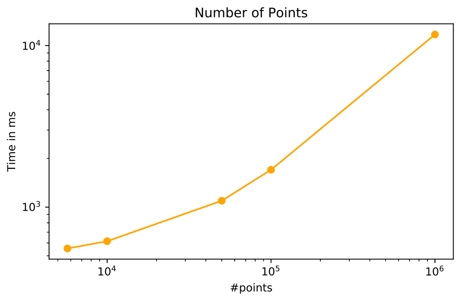


Figure 2: Execution time for different number of points.

Number of Representative points: We next varied the number of points used as representatives in each cluster from 1 to 100, and the results are shown in Figure 3. From this plot, we can see that the execution time increases linearly with respect to the number of representative points. This is due to the fact that the larger number of representatives has impact on the merge function, since more points have to be selected when 2 clusters are merged. Also at the final pass, each point has to be compared with more points, in order to be assigned to a cluster.

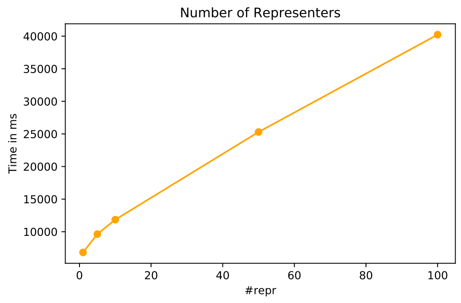


Figure 3: Execution time for different number of representative points.

Sample size: Finally, by varying the sample size that the algorithm takes into account in the first step, we can observe that the execution time increases exponentially, Figure 4.

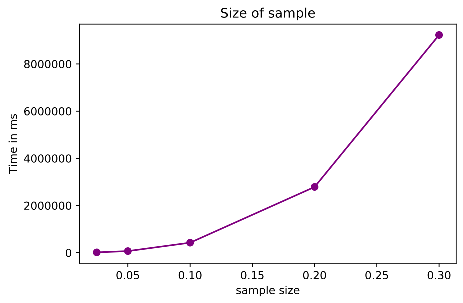
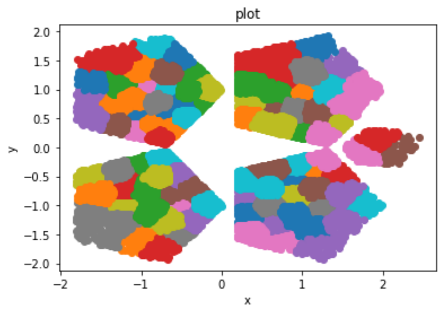
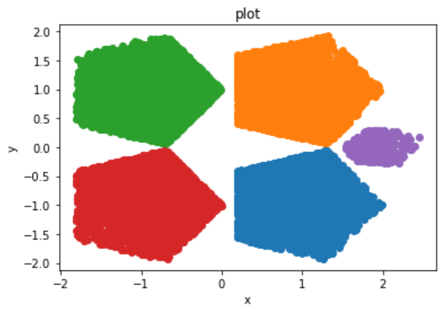


Figure 4: Execution time for different size of sample.



(a)



(b)

Figure 5: Ground Truth obtained by K-means with Hierarchical. (a)K-means with high K value, (b)K-means with Hierarchical

4.3 Experiments on Correctness

After examining the performance of the algorithm in terms of execution time, we proceeded to experiments on the correctness of the returned results. For these experiments, we considered it more appropriate to examine the performance of the algorithm in terms of recall and precision, based on a ground truth that was obtained using the solution of K-means with Hierarchical clustering. The accuracy of the algorithm is shown in Figure 5.

Our implementation exports for each run a folder with results, which includes a file with the points and the cluster this point is assigned to. In order to calculate the precision and recall based on the ground truth, we followed the below-described procedure.

First, we match the results returned by the algorithm with the ground truth data. In other words, we are trying to find a correspondence between the actual clusters and those identified by CURE. This is done by assigning to each real cluster the predicted cluster that covers the largest percentage of it, based on the points, but which has not already been assigned to another real cluster. To make the process more fair and accurate, we start the assignment of the predicted cluster with the highest percentage, then with the second-largest which has not been assigned yet, etc.

Then, after knowing the matching between real and predicted clusters, we calculate the true positive, false positive and true negative points for each of them. For each cluster i:

- **true positives (tp)** are the points that belong to this cluster and are assigned correctly to cluster i.
- **false positives (fp)** are the points that belong to another cluster but are assigned incorrectly to cluster i. In fact, false positives are all points assigned to cluster i, that do not belong to the true cluster matched with cluster i.
- **false negatives (fn)** are the points that belong to cluster i, but are incorrectly assigned to one of the other clusters.

The recall and precision metrics for each cluster individually are calculated by the following formulas

$$precision = \frac{tp}{tp + fp}$$

,

$$recall = \frac{tp}{tp + fn}$$

while the overall precision and recall are calculated using the macro-average formula.

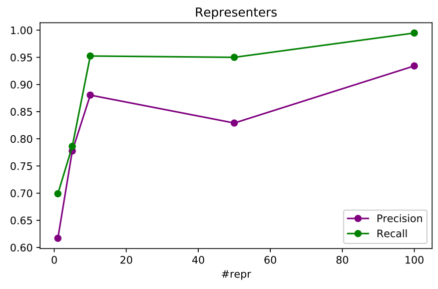
$$macro - precision = \frac{\sum_{\forall i \in Clusters} tp_i}{\sum_{\forall i \in Clusters} tp_i + fp_i}$$

$$macro - recall = \frac{\sum_{\forall i \in Clusters} tp_i}{\sum_{\forall i \in Clusters} tp_i + fn_i}$$

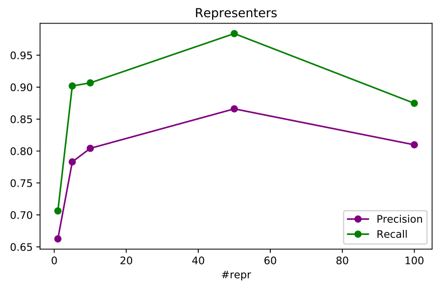
Number of Representative Points c. When varying the number of representative points per cluster, the results differ between the two datasets. From Figure 6 we can observe that for the small dataset the recall and precision both increase while the number of representatives increases. The above does not hold for the big dataset, in which the metrics increase while the number of representatives is less or equal to 50, but for representatives above 50 decreases.

Shrink Factor α. The next experiment is about varying the value of alpha between 0.2 and 0.7. The results which are shown in Figure 7 again differ between the two datasets. The best alpha values for the small dataset are either low, close to 0.3, or high, close to 0.6-0.7. On the other hand, as can be observed from the plot, the best alpha value for the big dataset is 0.7, while there is a general trend of precision and recall increasing while alpha increases.

Sample size. The last experiment concerns the variation of sample size, and how it affects precision and recall. As we can

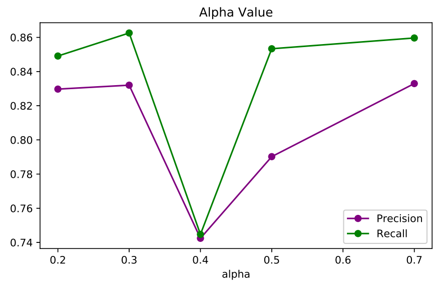


(a) Small Dataset

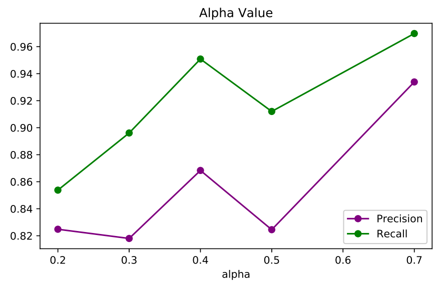


(b) Big Dataset

Figure 6: Precision and Recall for different number of representatives per cluster



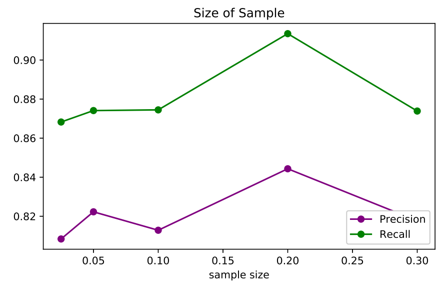
(a) Small Dataset



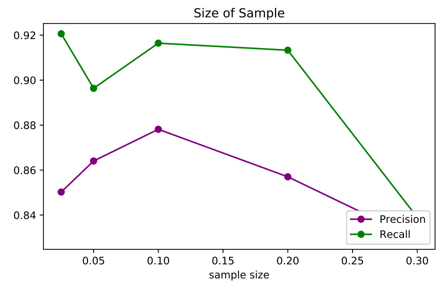
(b) Big Dataset

Figure 7: Precision and Recall for different values of parameter alpha

see from Figure 8, the results vary for both datasets between the values of sample size. The values chosen for the sample size, which are a percentage of the initial dataset, vary between 2.5% and 30%. From the plots, we can observe that when the sample size increases, the quality of clusters obtained from the small dataset improves while it worsens for the big dataset.



(a) Small Dataset



(b) Big Dataset

Figure 8: Precision and Recall for different sample size

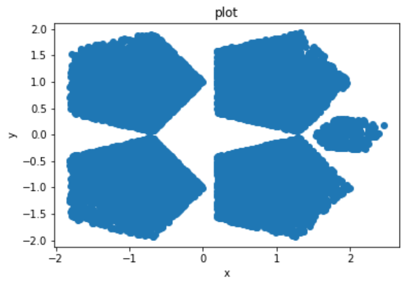
In general, we can observe from all the above plots that precision is always lower than recall, in both datasets, and in every experiment.

4.4 Experiments on Outlier Detection

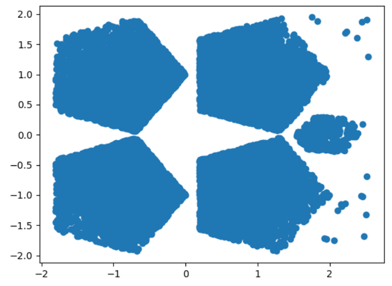
The goal of this section of the evaluation, is to test the performance of outlier detection. For both datasets provided, we created new ones with random artificial injected outliers, so as to validate the Precision and Recall of our implementation.

In order to create the artificial outliers, we first executed the k-means algorithm, with k = 100 to the original datasets and store the centers. Then we used a grid-method to spot empty space in the dataset. In these empty cells, we generate random points, using a normal distribution, making sure that the point's distance from the closest cluster deviates significantly from the original points in the dataset. We provide the code for the outlier injection, along with the rest of the code. In Figure 9(b), we can see 50 random outlier points injected in the provided dataset.

We generated 20 datasets, injected 50 outlier points to each one (10 for each provided dataset). Every experiment was executed 10 times and the mean value for Precision and Recall was kept. In Table 2 we present the parameters we used in the experiments in this Section, along with their default values, which is used when the other parameters are tested, and the range of their values.



(a)



(b)

Figure 9: The original dataset is depicted in (a) and in (b) is the original dataset with random outlier points injected.

Parameter	Default Value	Range
Sample size	0.3	0.025-0.4
Number of Representatives in Cluster	100	1-200
Shrink Factor α	0.3	0.2-0.7
n	3.5	1.5-4.5

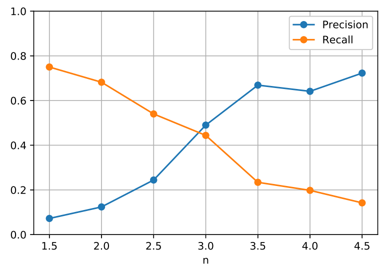
Table 2: Parameters

Number of standard deviations from the mean value (n): In this experiment we show how the performance of the CURE's outlier detection is effected by the parameter n. The parameter is varied from 1.5 times the standard deviation to 4.5 times and the results are presented in Figure 10.

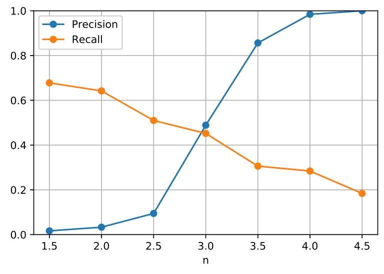
For small values of parameter n, more points are consider as outliers and thus even though Recall is high, Precision is close to zero. On the other hand, as parameter n increases, less and less points are reported as outliers, which causes some of the injected outliers to be missed, that is why Recall is decreasing and Precision is increasing. That is a clear trade off between these two.

Number of Representative Points c : We next varied the number of representative points from 1 to 200. For the smaller values of c, less than 10, the quality of the clusters suffered, as also mentioned in [1], which had impact in the performance of the outlier method.

Figure 11 shows the results of the experiment. Highest Precision for both datasets is achieved for c=100, reaching almost 100% for the large dataset, while Recall is maintained close to 40%.



(a)



(b)

Figure 10: Precision and Recall for different values of parameter n. (a) shows the results for the small provided-dataset and (b) the results for the large one.

Shrink Factor α: Figure 12 shows the Precision and Recall for the outlier detection when α is varied from 0.2 to 0.7. We used this range for parameter α, as it was recommended in [1].

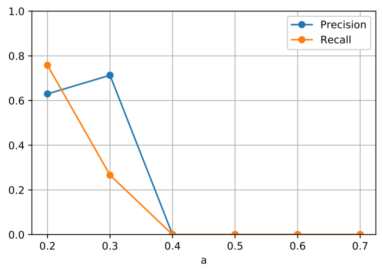
Both Precision and Recall are better with small value of α for both small and large dataset. That is, while α increases, points come closer to the center of the cluster, allowing more points at the edge of the cluster to be considered as outliers.

Figure 12(a) and Figure 12(b) are almost similar. We can see that large dataset has better performance than the small one, i.e. the as the number of points is increasing, it makes it easier to spot the points that deviates from the normal behavior. Cure achieves the better performance with respect to outlier detection for $\alpha = 0.2$, with Recall=0.88 and Precision=0.6.

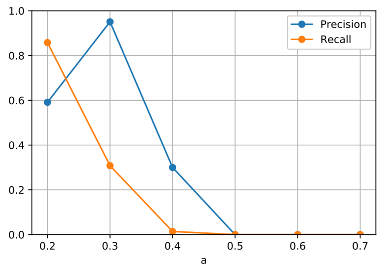
Sample size: We ran Cure while the sample size was varied from 2.5% to 40% and the results are depicted in Figure 13. In [1] they said that above 2.5% sample the cure will always return correctly identified clusters, but as show in the plot the precision of the method is highly related to the number of points that Cure executed on. While Precision is increasing as sample size increases for both datasets, that is not the case for Recall. In small dataset Recall is correlated to the sample size, unlike the large dataset, which shows an anti-correlated behavior.

5 CONCLUSION

As mentioned in the introduction, CURE is a hierarchical clustering algorithm that can easily detect arbitrarily shaped clusters. In this work, we implemented CURE in Scala, using Apache Spark, in order to take advantage of the parallelization that offers. In section

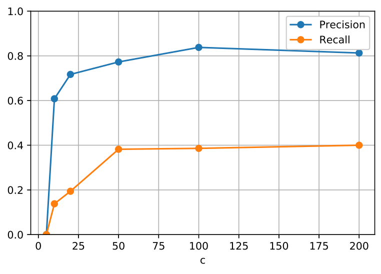


(a)

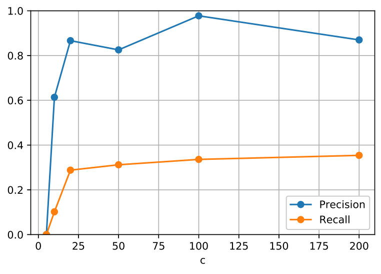


(b)

Figure 12: Precision and Recall for different values of parameter alpha. (a) shows the results for the small provided-dataset and (b) the results for the large one.

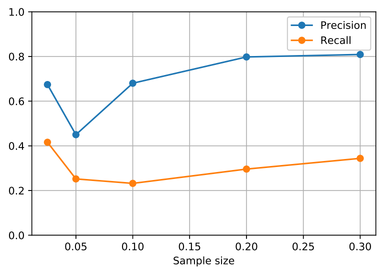


(a)

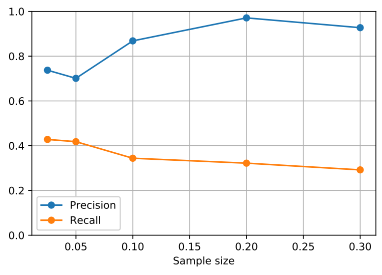


(b)

Figure 11: Precision and Recall for different number of representatives per cluster. (a) shows the results for the small provided-dataset and (b) the results for the large one.



(a)



(b)

Figure 13: Precision and Recall for different sample size. (a) shows the results for the small provided-dataset and (b) the results for the large one.

3 we describe in detail our implementation, where we parallelized the final pass the algorithm makes to the overall dataset. The section 4 presents the experiments we performed on the algorithm so that we can test both its scalability and the quality of the results it returns, as a function of some basic parameters such as shrinking factor a, number of points in the dataset, and the number of representative points per cluster. In addition, we modified the algorithm so that it can handle and detect anomalies in the dataset (outliers), something that is presented in section 3.3. In conclusion, as can also be observed from the experiments, because of its philosophy, CURE is an algorithm that can easily scale in large data.

REFERENCES

[1] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. 1998. CURE: An Efficient Clustering Algorithm for Large Databases *(SIGMOD '98)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/276304.276312
[2] Nikita Kumble and Vandan Tewari. 2021. Improved CURE Clustering Algorithm using Shared Nearest Neighbour Technique. *International Journal* 9, 2 (2021).
[3] Piyush Lathiya and Rinkle Rani. 2016. Improved CURE clustering for big data using Hadoop and Mapreduce. In *2016 International Conference on Inventive Computation Technologies (ICICT)*, Vol. 3. IEEE, 1–5.
[4] Seema Maitrey, CK Jha, Rajat Gupta, and Jaiveer Singh. 2012. Enhancement of CURE clustering technique in data mining. *International Journal of Computer Applications* (2012).