Pattern Recognition Letters 117 (2019) 90–96

Contents lists available at ScienceDirect

Pattern Recognition Letters

journal homepage: www.elsevier.com/locate/patrec

Sampling approaches for applying DBSCAN to large datasets ☆

Diego Luchi [a,*], Alexandre Loureiros Rodrigues [b], Flávio Miguel Varejão [a]

[a] Computer Science Department Federal University of the state of Espírito Santo, Vitória, Brazil
[b] Statistical Department Federal University of the state of Espírito Santo, Vitória, Brazil

ARTICLE INFO

Article history:
Received 22 January 2018
Available online 11 December 2018

MSC:
41A05
41A10
65D05
65D17

Keywords:
Clustering
Sampling
DBSCAN

ABSTRACT

DBSCAN is a classic clustering method for identifying clusters of different shapes and isolate noisy patterns. Despite these qualities, many articles in the literature address the scalability problem of DBSCAN. This work presents two methods to generate a good sample for the DBSCAN algorithm. The execution time decreases due to the reduction in the number of patterns presented to DBSCAN. One method is an improvement of the Rough-DBSCAN and presented consistently better results. The second is a new heuristic called I-DBSCAN capable of adapting and generating good results for all datasets without the need of any additional parameter.

1. Introduction

The DBSCAN (Density-based spatial clustering of applications with noise), proposed by [1], is an effective clustering technique and one of the most studied algorithms in the field, as it can be seen in [2] and [3,4]. The DBSCAN algorithm is known for its ability to find groups with arbitrary shapes and its capacity to isolate noise from the rest of the data in the clustering process [5]. For comparison, the Single-Linkage clustering (SLINK) [6] and the k-means algorithm [7], two of the most traditional clustering methods in the literature, are not robust or flexible enough to be considered in many clustering applications. The former, although known for its capability to detect clusters with arbitrary shapes, is very sensitive to noise. The latter has a severe limitation of detecting only compact and spherical shaped clusters [8]. In this context, DBSCAN appears as an attractive solution to many clustering problems. However, like several other clustering algorithms, it does not scale well for large datasets due to its high time complexity [9].

Several ways to overcome this high computational cost have been proposed. A survey of some approaches can be found in [10] and [11]. Many of these techniques aim to solve the nearest neighbor query with data structures, such as R*-Tree [12], M-Tree [13], X-Tree [14], or by putting a lattice over the data domain [15]. Those structures help to perform the nearest neighbor queries more efficiently for low-dimensional datasets, but if the number of dimensions is high the benefit of such data structures quickly vanishes. Such degradation of efficiency for many data structure frameworks is a well-documented effect [16,17].

For high-dimensional datasets, two viable alternatives are typically used to reduce the DBSCAN runtime. The first approach focus on solving the query of the nearest neighbors more efficiently [18] or in an *approximate* way, for example, to use hash methods such as *locality-sensitive hashing* (LSH) [19]. In [18] the authors proposed a clever way of avoiding unnecessary distance calculations for the nearest neighbor queries by taking advantage of triangular inequality. This proposal scales well for high-dimensional datasets unlike the aforementioned data structures used for spatial indexing. Therefore, it is an effective way to accelerate the execution of DBSCAN. However, this proposal has a greater neighborhood search complexity than approximate approaches such as LSH. On the other hand, by adopting hash schemes the density estimation for each element is susceptible to errors due to collisions in the hash table [20]. The second approach uses sampling techniques to improve the computational performance of DBSCAN to find an approximate solution even for large datasets. In those situations a small set of instances is chosen to represent the entire dataset. This size reduction greatly reduces the DBSCAN running time. While some approaches randomly generate the sample for the DBSCAN [21], others generate a sample from the output of clustering algorithms that have lower computational complexity, such as CLARANS in [22] and Leader in [23].

☆ **Conflict of interest.** The authors declare no conflict of interest.

* Corresponding author.
E-mail address: dluchi@inf.ufes.br (D. Luchi).

https://doi.org/10.1016/j.patrec.2018.12.010
0167-8655/

This work follows the latter approach and is motivated by the work in [23], which proposes a way to reduce the size of the dataset by applying the single pass Leader clustering algorithm [24] on the data. After the clustering step, the authors propose a method to decide whether or not the leaders of each cluster will compose the sample. This decision is made based on the density of each leader. However, with the output of the Leader algorithm alone it is very difficult to calculate their densities correctly. The authors present a clever way to approximate the densities, named *rough cardinality*. After running DBSCAN in the sample, the final partition is returned by assigning the remaining elements to their respective leader's cluster. Viswanath and Babu [23] call this method Rough-DBSCAN. Inspired by this work, this article presents two sampling methods to reduce the size of the dataset to carry out the DBSCAN algorithm. The first proposal, called Rough*-DBSCAN, is similar to Rough-DBSCAN, however the original Leader algorithm is modified to provide a trivial way to estimate the density of each leader. In the second method, called I-DBSCAN, a heuristic technique to draw a sample from the leaders and the elements contained in the intersections of the clusters found by the Leader algorithm is proposed.

Experiments measured the performance of the proposed methods by comparing their results with the one obtained by running the DBSCAN algorithm carried out over the entire dataset. While the first proposal achieved a better result approximation than the original method, the second proposal was the fastest method still getting good results.

The remainder of this paper is organized as follows: Section 2 briefly introduces the DBSCAN, Leader and Rough-DBSCAN clustering algorithms; Section 3 shows the Rough*-DBSCAN and the I-DBSCAN algorithms; Section 4 describes experiments where both methods proposed and the Rough-DBSCAN are applied on public domain datasets from the LIBSVM repository. Finally, Section 5 draws some conclusions.

2. DBSCAN, leader and rough-DBSCAN

This section shows a quick overview of DBSCAN, Leader, and Rough-DBSCAN algorithms.

2.1. DBSCAN

The DBSCAN algorithm, presented in [1], is one of the most successful clustering algorithms in the literature. Among its advantages, one can highlight its ability to identify clusters of various shapes and handle noisy data. DBSCAN (Algorithm 1) scans the entire dataset $\mathcal{D}$ (line 2) checking if each element $d \in \mathcal{D}$ has a density above a certain threshold *minPts*. The calculation of the density of any element is done by counting how many elements are within a distance less than a certain value ϵ. If the density is greater than *minPts* the element is considered a dense pattern (line 5) otherwise it will be temporarily classified as noise (line 6). If the element is dense, it will be assigned to a new cluster C (line 9) and a breadth-first search (BFS) in the dense neighboring elements of d that have not yet been examined begins (lines 16–18). After all iterations of the inner loop (line 11), all elements without cluster, or temporary classified as noise, discovered by the BFS, is assigned to the cluster C.

Despite the advantages described in the in the preceding section, DBSCAN has a very bad scalability due to the functions performed in lines 4 and 16 of the Algorithm 1. This function stands for nearest neighbors query (called NN-QUERY for short). It finds all elements of the dataset at a given distance from a given element. This operation is necessary to determine if the element is in a dense region or is a noisy pattern.

Algorithm 1: DBSCAN [25].

Data: Dataset $\mathcal{D}$, distance ϵ, minimum elements *minPts*

```
1  C ← 0;
2  for each d ∈ D do
3      if label_d is undefined then
4          Neighbors N ←NN-QUERY(D, d, ε);
5          if |N| < minPts then
6              label_d ← noise;
7          else
8              C ← C + 1;
9              label_d ← C;
10             S ← N \ {d};
11             for each q ∈ S do
12                 if label_q is noise then
13                     label_q ← C;
14                 if label_q is undefined then
15                     label_q ← C;
16                     Neighbors N ←NN-QUERY(D, q, ε);
17                     if |N| ≥ minPts then
18                         S ← S ∪ N;
19 return label_i (i = 1, ..., |D|)
```

Several papers focus on reducing the computational cost of NN-QUERY, while others focus on finding an approximate solution to NN-QUERY with lower computational burden. Another approach for reducing the cost of DBSCAN is to reduce the number of patterns in the dataset, either by random sampling or by using clustering algorithms to generate samples composed of prototypes. The Rough-DBSCAN and the two proposals presented in this paper use the Leader algorithm to search for prototypes. Therefore, in the next subsection the Leader algorithm is described.

2.2. Leader clustering algorithm

The Leader [24] is an incremental clustering algorithm which performs a *single scan* on the dataset. Unlike k-means, the Leader does not implement any optimization step. Therefore, it has a much shorter execution time.

The canonical Leader algorithm (Algorithm 2) starts making the first element $\mathcal{D}_0$ a leader and scans the remaining of the dataset

Algorithm 2: Leader.

Data: Dataset $\mathcal{D}$, Threshold distance τ

```
1  L ← D_0;
2  F_0 ← D_0;
3  for each d ∈ D \ {D_0} do
4      leader ← true;
5      for each l ∈ L do
6          if ||l − d|| ≤ τ then
7              F_l ← F_l ∪ d;
8              leader ← false;
9              break;
10     if leader then
11         L ← L ∪ d;
12         F_d ← d;
13 return {L, F}
```

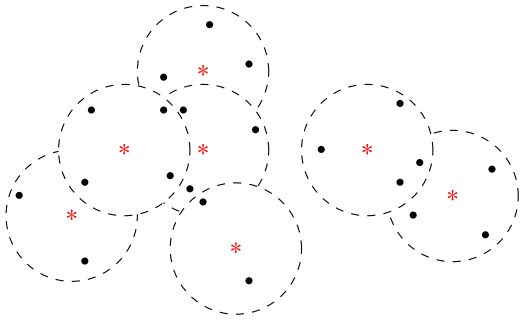

Fig. 1. Output example of the Leader algorithm, the highlighted red stars are the leaders and the followers are all the black dots inside the leader's region. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

verifying whether the current element is below a certain distance τ of any of the existing leaders. If an element is not close to any leader, it becomes a leader. Otherwise, it will be a follower of the first leader found within a distance less than τ. It is important to note that as $\tau \to 0$ all elements of the dataset will be leaders, $\mathcal{L} \to \mathcal{D}$.

It is easy to see that Leader is very sensitive to the order in which the data is presented to the algorithm. Elements at the beginning of the dataset have a higher chance of becoming leaders than the last ones. In addition, looking at lines 6–9 of Algorithm 2, one can note that the first leaders to be discovered will always have preference over others when deciding which leader the current element will follow (Fig. 1). If only the number of followers of each leader are considered to estimate their density, it will often underestimate the density of some clusters due to this priority in deciding which leader the follower will be associated. There will almost always be elements within a distance less than τ from the leader, but not following it. Therefore, calculating the density from the output of the Leader algorithm alone is not a trivial task.

2.3. Rough-cardinality

Viswanath and Babu [23] proposed a way to estimate the densities of the leaders from the output of the Leader algorithm, called rough-cardinality. The cardinality of a leader can be defined as the amount of elements existing within a given distance from the leader.

The rough-cardinality also uses the parameter ϵ (the parameter in DBSCAN used to determine whether or not a point is dense) to estimate the cardinality of each leader. The rough-cardinality is defined for each leader as the *sum of the cardinality of all leaders* within a radius ϵ. This procedure has a worst case complexity of $\mathcal{O}(k^2)$ where k represents the number of leaders found in the previous step. Although it has a quadratic complexity, normally k is much smaller than the number of patterns in the dataset. Even though the Leader algorithm guarantees that no leader will be closer than τ from another, note that if $\tau \leq \epsilon$ there can still be leaders with a less than ϵ distance from each other.

There are two types of errors in this estimation. First, leaders who are more than ϵ distance yet have priority over the leader which the procedure is trying to estimate the density, this effect will decrease the density of this leader. Secondly, leaders who are a distance above τ and less than ϵ with elements at a distance greater than ϵ of the current leader increases the density estimation of the latter. The authors show that on average the two effects tend to cancel each other and this method provides a good estimation for the leader's densities [23]. While the results obtained by the authors were satisfactory, it is possible to improve the results by using a mechanism to correct the density estimation.

2.4. Rough-DBSCAN

In Rough-DBSCAN, proposed by Viswanath and Babu [23], the Leader algorithm is used to generate a new dataset with the prototypes and their respective densities. These densities are calculated based on the number of elements in each cluster returned by the algorithm.

The Rough-DBSCAN combines the methods that have been presented so far. The first step is to generate a sample using the Leader clustering algorithm. The second step is to evaluate the density of each leader using the rough-cardinality method. This new dataset only consists of leaders who have an estimated density above the DBSCAN threshold. Thus, the sample is composed only by the leaders with cardinality greater than *minPts*. The DBSCAN algorithm runs on this sample and identifies the clusters. The result of DBSCAN is a grouping of leaders. To construct the final partition, each leader must be replaced by all his followers found during the execution of the Leader algorithm, and every follower will belong to the same cluster of the leader.

3. Rough*-DBSCAN and I-DBSCAN

This section presents two new adaptations of the DBSCAN algorithm to perform clustering tasks in huge datasets. The two proposals are based on a modified version of the Leader algorithm, called Leader*. Therefore, this section starts describing the Leader* and follows describing the Rough*-DBSCAN and the I-DBSCAN.

3.1. Leader* algorithm

One of the major weaknesses of Rough-DBSCAN is the density estimation based on the result of the Leader algorithm. In order to overcome this weakness, an element may be associated with more than just one leader. With this modification a more precise and straightforward density estimation can be performed.

The modified version of the Leader algorithm can be seen in Algorithm 3. Even though it is necessary to do a second scan on the dataset (lines 10–13), in this modification the preference effect of the Leader algorithm is eliminated. The density is estimated based on the number of elements associated with each leader. Fig. 2 illustrates the result of this modified version. The small increase in the computational cost leads to much better results due to the correct density estimation.

Algorithm 3: Leader*.

Data: Dataset $\mathcal{D}$, Threshold distance τ, Distance ϵ

```
1  L ← D_0;
2  for each d ∈ D \ {D_0} do
3      leader ← true;
4      for each l ∈ L do
5          if ||l − d|| ≤ τ then
6              leader ← false;
7              break;
8      if leader then
9          L ← L ∪ d;
10 for each d ∈ D do
11     for each l ∈ L do
12         if ||l − d|| ≤ ε then
13             F_l ← F_l ∪ d;
14 return {L, F}
```

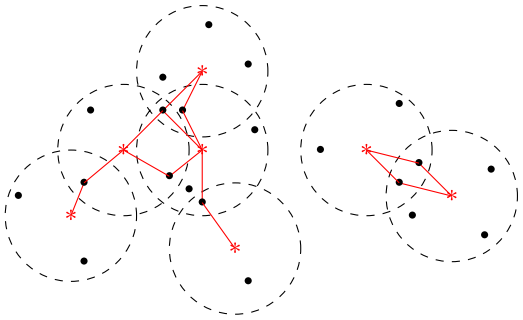

Fig. 2. Output example of the modified Leader algorithm, the highlighted red stars are the leaders and the followers are all the elements inside the leaderâs region (including intersections). The red lines connecting some elements to the leaders show followers who follow simultaneously more than one leader. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3.2. Rough*-DBSCAN

The Rough*-DBSCAN is very similar to Rough-DBSCAN, the difference is in the Leader algorithm itself. Instead of executing the canonical Leader algorithm and estimating the density of each leader through the rough-cardinality, the Rough*-DBSCAN uses the modified version of the Leader to obtain the densities of the leaders to carry out the DBSCAN.

Given the DBSCAN parameters, ϵ and *minPts*, and the τ parameter the Rough*-DBSCAN performs a modified version of the Leader clustering algorithm, presented in this section, and only selects the leaders with more than *minPts* followers in a sphere of radio ϵ to compose the sample. All leaders with less than *minPts* followers are discarded. Since all elements in the sample have more than *minPts* elements within a ϵ radius, they are above the density required by DBSCAN to not be marked as noise, so the *minPts* parameter is from now on irrelevant and it is possible to assign $minPts = 1$. The next step of the algorithm is run the DBSCAN algorithm with parameters ϵ and $minPts = 1$ on the sample. The final solution is returned by taking every leader on the sample and replacing by all the followers of that leader, without duplicates. Due to the intersections it is possible that the algorithm tries to add an element more than once. In this situation the element will be added in an arbitrarily chosen leader. The remaining elements are marked as noise.

It's is important to note when $\tau \to 0$ the result of Rough*-DBSCAN approximates the result of DBSCAN in the whole dataset. In this scenario, the output of the Leader algorithm will be composed of each element of the dataset $\mathcal{D}$ as a leader, forming $|\mathcal{D}|$ clusters without followers. The density of every leader will be estimated and input for the DBSCAN algorithm will be composed of all the dense elements of the dataset. The DBSCAN algorithm will group the leaders into clusters and every element assigned to the DBSCAN will be replaced by itself in the same cluster and the remaining of the elements left over the sample by the density criteria are marked as noise. The τ parameter acts as an approximation factor of the DBSCAN algorithm in the complete dataset. When $\tau = 0$, the algorithm returns the same result as the DBSCAN, whereas when τ increases the results become further from that one obtained by DBSCAN in the complete dataset, but with a reduction in the runtime.

3.3. I-DBSCAN

In addition to the Rough*-DBSCAN, this work proposes a heuristic to draw a sample for the DBSCAN that also uses the modification of the Leader algorithm, proposed in Section 3.1. The main idea is to compose a sample of leaders and some elements belonging to more than one leader, i.e. elements on the intersection of the leaders' spheres. For that reason it was called I-DBSCAN, where I stands for intersection.

Unlike Rough-DBSCAN and Rough*-DBSCAN, the I-DBSCAN does not have the τ parameter. Therefore, it does not need to be tuned. This heuristic only needs parameters ϵ and *minPts*, the parameters required by the DBSCAN algorithm.

Due to the absence of the τ parameter, the I-DBSCAN algorithm runs with $\tau = \epsilon$, and for that reason, can not execute using only the leaders as a sample. Since the leaders will be at a distance greater than or equal ϵ from each other, if DBSCAN were to run in that sample with $minPts > 1$ all elements would be marked as noise. After all, there will be no other element at a distance less than ϵ, or in case $minPts = 1$, each element would be assigned to a different cluster.

For ensuring the dense leader in the original dataset is also dense in the sample one must add more elements to the sample than just the set of leaders. Adding the elements at the intersection of leaders is a good idea for two reasons. Firstly, elements belonging to the intersections contribute to the density of more than one leader and, therefore, helps to keep the sample size small. Secondly, the elements at the intersections allow a path by which NN-QUERY can reach one leader from another. These elements are closer than ϵ from the leaders which ensures that all leaders who are linked through these elements will belong to the same cluster. The only exception happens when one of the two leaders is classified as noise by the DBSCAN. In this case the element at the intersection will be assigned to the cluster of the non-noisy leader.

The I-DBSCAN starts executing the modified version of the Leader clustering algorithm with $\tau = \epsilon$, as can be seen in the first line of the Algorithm 4. After the execution of the Leader algorithm, the set of all leaders are inserted into the sample (line 2). The remainder of the sampling algorithm basically examines each leader (line 3) to add new elements to the sample. If a leader contains less than *minPts* followers, all the followers of that leader who follow more than one leader are added to the sample (line 10). Otherwise, if the leader has more than *minPts* followers (line 5), two scenarios should be considered. The first happens when the leader has less than *minPts* elements at intersections. In this case, all elements in the intersection are added to the sample and also some elements chosen by a uniform random procedure SAMPLE (line 9). Those extra elements are just for ensuring that a leader who is dense in the original dataset remains dense in the sample. These random elements are inserted only to prevent the leader being marked as noise by the DBSCAN algorithm. The second sce-

Algorithm 4: I-DBSCAN.

Data: Dataset $\mathcal{D}$, ϵ, *minPts*

1 $\{\mathcal{L}, \mathcal{F}\} \leftarrow$ Leader*$(\mathcal{D}, \epsilon, \epsilon)$; /* Modified version */
2 $\mathcal{S} \leftarrow \mathcal{L}$;
3 **for** *each* $l \in \mathcal{L}$ **do**
4 $s \leftarrow$ Find all followers of l in any intersection;
5 **if** $|\mathcal{F}_l| > minPts$ **then**
6 **if** $|s| > minPts$ **then**
7 $s \leftarrow$ FFT-SAMPLING$(s, minPts)$;
8 **else**
9 $s \leftarrow s \cup$ SAMPLE$(F_l, minPts - |s|)$;
10 $\mathcal{S} \leftarrow \mathcal{S} \cup s$;
11 **return** $\mathcal{S}$;

Table 1
Datasets and DBSCAN parameters.

Dataset	#elements	#features	ϵ	*minPts*
Abalone (Scale)	4177	8	0.2	3
Mushrooms	8124	112	2.5	4
Pendigits	10,992	10	40	4
Letter	20,000	26	0.5	8
Cadata	20,640	8	200	8
Shuttle	58,000	7	0.03	20
Sensorless (Scale)	58,509	48	0.3	20
SensIT (acoustic)	98,528	50	0.5	5
SensIT (seismic)	98,528	50	0.4	5
Skin nonskin	245,057	3	60	10
Poker	1,025,010	10	4	10

nario occurs when the leader has more than *minPts* elements in the intersections. It is not needed to add all of them to ensure the density of the leader. For keeping the sample as small as possible, it is only necessary to add *minPts* elements. The elements are chosen using the Farthest-First Traversal sampling (FFT-SAMPLING) algorithm (line 7). Farthest-First Traversal sampling initially selects an arbitrary element and then successively selects the farthest element from the set of previously selected elements. This assures the selected elements are well spread out, probably covering more than one intersection.

After the sample is assembled, the next steps of the algorithm are similar to the others methods. The DBSCAN algorithm runs in the sample, with the *minPts* and ϵ parameters, and like Rough-DBSCAN and Rough*-DBSCAN the final partition is classified based on the leaders' clusters. Every leader on the sample is replaced by all the followers found by the Leader clustering algorithm. Note that all non-leader elements added in the sample are discarded in this final classification, they are added just to assure the minimum density of the leaders and help DBSCAN find the correct grouping for the leaders.

4. Experiments

The comparison of methods was performed on eleven datasets (Table 1). Two of them, Pendigits and Letter, were chosen for being present in the original work of Rough-DBSCAN. The others are public domain datasets, from different domains avaliable on the LIBSVM repository. They were chosen for verifying if the behavior of the algorithms remains the same in different scenarios. The experiments try to identify which method most closely matches the DBSCAN results and still significantly reduces its running. The performance is measured by the Adjusted Rand Index (ARI) [26], a metric that measures the level of agreement between two clustering results of a dataset. The Adjusted Rand Index has a maximum value of 1 indicating that the clusterings are exactly the same. In the experiments, the ARI was measured comparing the clusters obtained by each method, with the partitioning obtained by the DBSCAN method running in the complete dataset.

The two largest datasets, Poker and Skin nonskin, were included to demonstrate the ability of these methods to run in larger datasets. For those datasets the DBSCAN was not executed due to the high computational cost. The original DBSCAN was interrupted for these datasets when the running time exceeded 24 h. For that reason in these datasets each method had its results measured against the results obtained by the other two sampled methods.

The parameters τ, ϵ and *minPts* used on the experiments performed on the Pendigits and Letter datasets have the same values used on the experiments performed in [23]. For the other datasets, the parameters were chosen arbitrarily. Table 1 shows the values of the parameters ϵ and *minPts* for each dataset in addition to the number of elements and features.

The τ parameter affects the sample size of the Rough-DBSCAN and Rough*-DBSCAN. For each dataset four values were chosen for τ to generate from small to large samples. Due to the lack of a random component in the methods, each one was performed only once for each parameter setting.

Table 2 shows the experimental results for all datasets with exception of Poker and Skin nonskin datasets. The best ARI and time results are presented in bold. Fig. 3 summarizes the results of Table 2 showing the averaged ranking for ARI and running time for all methods considering all datasets and each τ value setting. The first value on the x-axis summarizes the results for the highest τ value for all datasets, the next value for the second largest τ value, and so on. When τ is large, yielding a small sample size for the Rough-DBSCAN and Rough*-DBSCAN, one may see that the I-DBSCAN gets the best average ranking for the ARI and the Rough-DBSCAN is the fastest method. The sample sizes rises as τ diminishes and the Rough*-DBSCAN becomes the closest method to the DBSCAN results, going from the worst ranking to the best. However, it is the slowest method in all cases. Although I-DBSCAN has a fixed running time it becomes the fastest since the other two methods require larger computational time as a result of the increase of the sample size. It is important to note that Rough-DBSCAN presented the worst rankings for the ARI (except when τ was at the maximum) and it is always worse than the rank obtained by I-DBSCAN.

Once DBSCAN and I-DBSCAN do not have the τ parameter, Tables 2 and 3 display the same value of running time of these methods for each dataset.

4.1. Running time

The runtime for all datasets (Tables 2 and 3) shows the proximity of the Rough-DBSCAN and the Rough*-DBSCAN. Rough-DBSCAN has a small advantage in time when compared against Rough*-DBSCAN in all datasets due to the modified version of the Leader algorithm, which requires two scans in the dataset. However, as it can be seen, this additional step in the Leader does not impact too much the overall execution time.

The results for the Abalone, Pendigits and Letter datasets (Table 2) show that the Rough-DBSCAN and Rough*-DBSCAN are slower than DBSCAN itself when the τ value is small. However, for larger datasets, even the slowest method (Rough*-DBSCAN) with the smallest τ parameter value runs faster than DBSCAN. These facts indicate that both methods have a relatively high computational time, but they have a better scalability than the DBSCAN algorithm. Therefore, for larger datasets, all methods offer a significant execution time reduction.

The I-DBSCAN had smaller overall execution times for all datasets. The only exception is seen when τ was close to ϵ. These cases coincide exactly with the smallest sample size and with a poor approximation in relation to the results obtained by the DBSCAN. In these situations Rough-DBSCAN and Rough*-DBSCAN algorithms achieved a better runtime performance.

The methods were run on larger datasets for testing their scalability. Table 3 shows the results for Skin nonskin and Poker datasets. In the Skin nonskin dataset the patterns seemed very dense and the runtime of both Rough-DBSCAN and Rough*-DBSCAN was low due to the small sample size required. Although fast, the procedure for selecting the elements at the intersections in the I-DBSCAN had a major impact on the overall execution time. The results for the Poker dataset repeats the same results presented for the smaller datasets (as shown by Fig. 3).

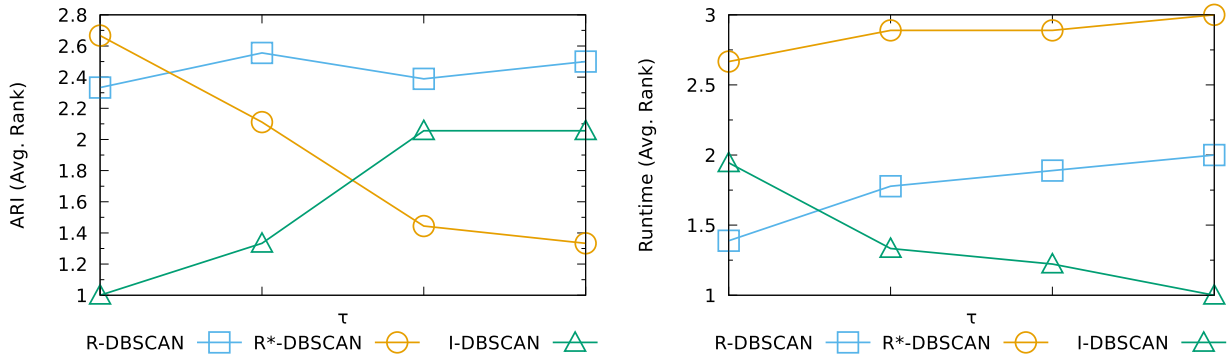

Fig. 3. Ranking comparison between Rough-DBSCAN, Rough*-DBSCAN, I-DBSCAN for all results presented in Table 2. For each dataset the three methods were ranked based on their performances and average across the datasets were calculated for each method.

Table 2
Experimental results.

		DBSCAN	Rough-DBSCAN		Rough*-DBSCAN		I-DBSCAN	
Dataset	τ	Time(s)	ARI	Time (s)	ARI	Time (s)	ARI	Time (s)
Abalone (Scale)	0.15	2.0	0.89498	**0.3**	0.82760	0.4	**0.98212**	**0.3**
	0.1	2.0	**0.99542**	0.8	0.97705	0.9	0.98212	**0.3**
	0.075	2.0	0.99834	1.5	**0.99878**	1.7	0.98212	**0.3**
	0.05	2.0	0.99953	2.7	**1.0**	2.8	0.98212	**0.3**
Mushrooms	2.25	97.8	0.90744	1.9	0.90723	4.3	**0.91018**	**1.8**
	2	97.8	0.90744	2.1	0.90967	29.5	**0.91018**	**1.8**
	1.75	97.8	**0.91018**	16.8	0.90967	29.5	**0.91018**	**1.8**
	1.5	97.8	**0.91018**	16.6	0.90967	29.4	**0.91018**	**1.8**
Pendigits	30	25.6	0.90735	5.9	0.87531	8.2	**0.97226**	**5.5**
	25	25.6	0.97434	12.6	**0.98108**	16.4	0.97226	**5.5**
	20	25.6	0.97997	25.5	**0.99382**	30.0	0.97226	**5.5**
	15	25.6	0.98396	43.8	**0.99873**	45.0	0.97226	**5.5**
Letter	0.4	83.2	0.48805	**25.9**	0.51722	35.9	**0.78703**	36.5
	0.35	83.2	0.60939	47.6	0.74672	62.1	**0.78703**	**36.5**
	0.3	83.2	0.65552	63.2	**0.82271**	77.5	0.78703	**36.5**
	0.25	83.2	0.71589	104.2	**0.92820**	114.6	0.78703	**36.5**
Cadata	190	105.5	0.00176	**2.6**	0.00086	4.0	**0.95008**	5.3
	170	105.5	0.32304	**3.0**	0.23303	4.8	**0.95008**	5.3
	150	105.5	0.80000	**3.8**	0.75335	5.8	**0.95008**	5.3
	120	105.5	0.88185	5.4	0.93695	8.0	**0.95008**	**5.3**
Shuttle	0.025	791.6	0.20430	**4.2**	0.17485	5.9	**0.82221**	8.5
	0.02	791.6	0.75313	**6.8**	0.74034	9.2	**0.82221**	8.5
	0.015	791.6	0.85545	9.3	**0.85470**	13.0	0.82221	**8.5**
	0.01	791.6	0.99861	15.3	**0.99992**	21.4	0.82221	**8.5**
Sensorless (Scale)	0.2	7379.3	0.79940	**32.6**	0.81089	104.7	**0.99891**	34.0
	0.175	7379.3	0.76622	55.4	0.77719	146.7	**0.99891**	**34.0**
	0.15	7379.3	0.80119	107.2	0.81217	264.8	**0.99891**	**34.0**
	0.125	7379.3	0.93949	208.5	0.95797	484.5	**0.99891**	**34.0**
SensIT (acoustic)	0.4	31640.9	0.77247	1265.0	0.86352	1775.5	**0.93737**	**1033.4**
	0.35	31640.9	0.79622	1879.4	**0.95174**	2530.8	0.93737	**1033.4**
	0.3	31640.9	0.79910	2734.2	**0.98590**	3461.8	0.93737	**1033.4**
	0.25	31640.9	0.79992	3953.5	**0.99487**	5155.7	0.93737	**1033.4**
SensIT (seismic)	0.35	10626.4	0.70192	**2405.2**	0.56338	3296.8	**0.96184**	3507.2
	0.3	10626.4	0.77549	4232.6	0.93424	4970.6	**0.96184**	**3507.2**
	0.25	10626.4	0.79154	5788.6	**0.98769**	6986.2	0.96184	**3507.2**
	0.2	10626.4	0.79323	7440.9	**0.99821**	7766.6	0.96184	**3507.2**

Table 3
Experimental results for the Skin nonskin and Poker data sets.

		Rough-DBSCAN			Rough*-DBSCAN		I-DBSCAN
Dataset	τ	Time (s)	ARI (Rough*-DBSCAN)	ARI (I-DBSCAN)	Time (s)	ARI (I-DBSCAN)	Time (s)
Skin nonskin	28	**2.5**	0.98217	0.97781	5.8	0.99014	6.3
	26	**2.6**	0.99450	0.99233	6.3	0.99015	6.3
	24	**3.0**	0.99395	0.99179	7.3	0.99124	6.3
	22	**3.7**	0.88711	0.88175	8.6	0.99124	6.3
Poker	3.8	**1844.0**	0.91414	0.98236	2371.0	0.98236	2340.9
	3.7	2599.6	0.99309	0.99874	3112.5	0.99874	**2340.9**
	3.6	3145.5	0.99866	0.99985	4088.8	0.99985	**2340.9**
	3.5	3355.1	0.99866	0.99985	4163.3	0.99985	**2340.9**

4.2. Adjusted Rand Index

All the methods obtained a good level of agreement with the results of the DBSCAN, considering a suitable choice of the τ parameter value. The I-DBSCAN does not need this extra parameter and still achieved a good level of approximation for all datasets. The worst results for the I-DBSCAN achieved an Adjusted Rand Index of 0.78 and 0.82 in the Letter and Shuttle datasets respectively (Table 2) and on the others datasets the ARI was always above 0.91. Except from Abalone, Pendigits, and Shuttle datasets, the I-DBSCAN obtained better results than Rough-DBSCAN for all values of τ with a shorter overall runtime.

The Rough-DBSCAN obtained the best results only in 3 cases. For most cases the Rough*-DBSCAN outperforms the Rough-DBSCAN. Indeed, the Rough*-DBSCAN with the two lowest setting of τ shows the best Adjusted Rand Index for the majority of the datasets.

To verify if the results have any statistical significance, the Wilcoxon signed-rank test was applied in the results presented in Table 2. As in Fig. 3, the results were grouped into four groups considering the parameter τ in order to avoid the dependence between the results of the same dataset. The first group is composed of all results of the highest τ values, second group composed of all results of the second largest value, and so on. The I-DBSCAN presented the best results for the highest values of τ (0.0039 p-value for Rough-DBSCAN and Rough*-DBSCAN) while the results between Rough-DBSCAN and Rough*-DBSCAN had no statistical difference on average. In descending order of τ values, the next test showed that there are statistical differences only between the I-DBSCAN and Rough-DBSCAN (0.0273 p-value). For this scenario I-DBSCAN is still achieved the best performance (alongside with Rough*-DBSCAN). In the next test, due to the increase of the sample size, Rough-DBSCAN was able to match the I-DBSCAN results, so no method showed any statistical difference. For the final test, with the lowest τ value, the I-DBSCAN and the Rough-DBSCAN still did not make a significant difference while the Rough*-DBSCAN stood out from the two with the best results against Rough-DBSCAN (0.0117 p-value) and with no statistical difference in relation to I-DBSCAN.

Table 3 shows the results for the two largest datasets, however, since ARI is a symmetric function, some existing results displayed in the previous columns may be omitted. In both datasets the agreement level between the methods was above 0.99 with the exception of the results for the Skin nonskin dataset when $\tau = 22$. In this case, Rough-DBSCAN found a different result, while I-DBSCAN and Rough*-DBSCAN presented a high level of agreement. The algorithms in fact allowed the execution in large datasets with a high degree of concordance.

5. Conclusion

This work proposed two methods for sampling large datasets in order to execute the DBSCAN algorithm. After obtaining the clustering result on the sample, the rest of the elements are quickly distributed in the clusters.

One of the methods presented is a variation of the Rough-DBSCAN, called, Rough*-DBSCAN, which obtained a better approximation for the DBSCAN than the original method with a marginally higher computational cost. The other method proposed, I-DBSCAN, was the fastest method without requiring any further parameter to be tuned. On average, it achieved approximately 92.5% agreement with the result obtained by DBSCAN.

Both Rough-DBSCAN and Rough*-DBSCAN need the τ parameter and a wrong choice for the τ value greatly impacts the quality of the results obtained by both methods. Until today there is no established method to determine a value for this parameter. This problem will be studied in future works.

References

[1] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al., A density-based algorithm for discovering clusters in large spatial databases with noise, in: Kdd, 96, 1996, pp. 226–231.
[2] SIGKDD, SIGKDD test of time award, 2014.
[3] R. Xu, D. Wunsch, Survey of clustering algorithms, IEEE Trans. Neural Networks 16 (3) (2005) 645–678.
[4] V. Chandola, A. Banerjee, V. Kumar, Anomaly detection: a survey, ACM Comput. Surv. (CSUR) 41 (3) (2009) 15.
[5] A. Kassambara, Practical Guide to Cluster Analysis in R: Unsupervised Machine Learning, 1, STHDA, 2017.
[6] R. Sibson, Slink: an optimally efficient algorithm for the single-link cluster method, Comput. J. 16 (1) (1973) 30–34.
[7] J. MacQueen, et al., Some methods for classification and analysis of multivariate observations, in: Proceedings of the fifth Berkeley symposium on mathematical statistics and probability, 1, Oakland, CA, USA, 1967, pp. 281–297.
[8] A.K. Jain, Data clustering: 50 years beyond k-means, Pattern Recognit. Lett. 31 (8) (2010) 651–666.
[9] J. Gan, Y. Tao, Dbscan revisited: mis-claim, un-fixability, and approximation, in: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM, 2015, pp. 519–530.
[10] T. Ali, S. Asghar, N.A. Sajid, Critical analysis of dbscan variations, in: Information and Emerging Technologies (ICIET), 2010 International Conference on, IEEE, 2010, pp. 1–6.
[11] K. Khan, S.U. Rehman, K. Aziz, S. Fong, S. Sarasvady, Dbscan: past, present and future, in: Applications of Digital Information and Web Technologies (ICADIWT), 2014 Fifth International Conference on the, IEEE, 2014, pp. 232–238.
[12] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The r*-tree: an efficient and robust access method for points and rectangles, in: Acm Sigmod Record, 19, ACM, 1990, pp. 322–331.
[13] P. Ciaccia, M. Patella, F. Rabitti, P. Zezula, Indexing metric spaces with m-tree, in: SEBD, 97, 1997, pp. 67–86.
[14] S. Berchtold, D.A. Keim, H.-P. Kriegel, The x-tree: an index structure for high--dimensional data, in: Proceedings of the 22th International Conference on Very Large Data Bases, in: VLDB '96, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996, pp. 28–39.
[15] S. Mahran, K. Mahar, Using grid for accelerating density-based clustering, in: Computer and Information Technology, 2008. CIT 2008. 8th IEEE International Conference on, IEEE, 2008, pp. 35–40.
[16] E. Chávez, G. Navarro, R. Baeza-Yates, J.L. Marroquín, Searching in metric spaces, ACM Comput. Surv. (CSUR) 33 (3) (2001) 273–321.
[17] D. Arlia, M. Coppola, Experiments in parallel clustering with dbscan, in: European Conference on Parallel Processing, Springer, 2001, pp. 326–331.
[18] Y. Chen, S. Tang, N. Bouguila, C. Wang, J. Du, H. Li, A fast clustering algorithm based on pruning unnecessary distance computations in dbscan for high-dimensional data, Pattern Recognit. (2018).
[19] Y.-P. Wu, J.-J. Guo, X.-J. Zhang, A linear dbscan algorithm based on lsh, in: Machine Learning and Cybernetics, 2007 International Conference on, 5, IEEE, 2007, pp. 2608–2614.
[20] M. Datar, N. Immorlica, P. Indyk, V.S. Mirrokni, Locality-sensitive hashing scheme based on p-stable distributions, in: Proceedings of the twentieth annual symposium on Computational geometry, ACM, 2004, pp. 253–262.
[21] X. Wang, H.J. Hamilton, Dbrs: a density-based spatial clustering method with random sampling, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, Springer, 2003, pp. 563–575.
[22] Y. El-Sonbaty, M.A. Ismail, M. Farouk, An efficient density based clustering algorithm for large databases, in: Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on, IEEE, 2004, pp. 673–677.
[23] P. Viswanath, V.S. Babu, Rough-dbscan: a fast hybrid density based clustering method for large data sets, Pattern Recognit. Lett. 30 (16) (2009) 1477–1488.
[24] J.A. Hartigan, Clustering Algorithms, 99th, John Wiley & Sons, Inc., New York, NY, USA, 1975.
[25] E. Schubert, J. Sander, M. Ester, H.P. Kriegel, X. Xu, Dbscan revisited, revisited: why and how you should (still) use dbscan, ACM Trans. Database Syst. (TODS) 42 (3) (2017) 19.
[26] L. Hubert, P. Arabie, Comparing partitions, J. Classif. 2 (1) (1985) 193–218.