Implementation Report

IDBSCAN sampling approach for applying DBSCAN algorithm on large datasets

URL

Tair Tahar

June 1, 2021

Contents

1 Introduction

DBSCAN is a density-based spatial clustering algorithm, that proved to be an effective method and one of the most studied algorithms in the field of clustering. The important property of DBSCAN is that it allows capturing groups with arbitrary shapes, while being robust to noise to a large extent. Other competitors in the clustering field, such as k-means, are not equally able to address those data properties.

With those advantages said, DBSCAN has a problem to scale for large datasets given its high complexity. Some suggestion for improving the nearest neighbors search have been published. However they are applicable only with low-dimensional datasets.

DBSCAN has another drawback that it has 2 parameters that affect the performance deeply: Epsilon (ϵ) which is the radius to look for neighbors, and minimum points (minpts), minimal quantity of neighbors to find inside epsilon. Finding the adequate parameters for DBSCAN is not an easy task, and it is also a subject that has been investigated separately.

In light of the main difficulty of the algorithm to scale to large datasets, a new algorithm called IDBSCAN was suggested, where the "I" in the name stands for "Intersection". In this algorithm we apply some methods to sample the data in a representative manner, which allows to then apply DBSCAN on much smaller sample size, that eventually reduces the runtime. Applying IDBSCAN prior to the DBSCAN clustering process means that the clustering only performed on a subset of the original data, meaning less calculations and less complexity. Eventually there is a step of inferring the classes of the rest of the data from the subset predictions.

In this work I document an implementation of the whole IDBSCAN algorithm, and its auxiliary functions (as well as DBSCAN itself). First I will explain the theory and how the algorithm work. Then I will explain more details about the implementation. Furthermore, I will present some experiments performed with algorithm, comparing to the baseline which is DBSCAN, and comparing to other variants of DBSCAN. I will then open some discussion and thoughts observing the results. Finally, some conclusions and future work that I would be very interested to keep and investigate/develop will be presented.

In this paper I use the ARI, Adjusted Rand Index to evaluate the performance of different algorithms. This score compares two different labeling for the same data. the inputs are arrays in the length of the data with cluster number. ARI computes a similarity measure between two clusterings, by considering all the pairs that are assigned to same cluster. That is the answer to the question "how much the clusterings agree?". The score can vary between 0 to 1, where 0 means the results are not correlated at all, random labeling, and 1 means that the two clusterings compared are identical.

2 Algorithms Explanations

This work includes an implementation of the different sub-algorithms that together compose the IDBSCAN: DBSCAN, Leader, Leader_asterisk (referred as "leader*" in the paper), FFT_sampling. Each algorithm will be explained briefly in the following sub-sections.

2.1 DBSCAN

DBSCAN name stands for Density-Based Spatial Clustering with Noise. It is one of the most successful approaches for clustering, and its main highlights are its handling with noise and the ability of capturing different more amorphous shapes of clusters. The shape of the scattered

cluster is crucial for clustering algorithms that are based on centroids, e.g `k-means` and its variants, to cluster well. However, relying on the sample distance from center of mass is no longer meaningful when having some more "complex" shape of scattering than spherical (e.g spiral/moon etc), as can be seen in the following image:

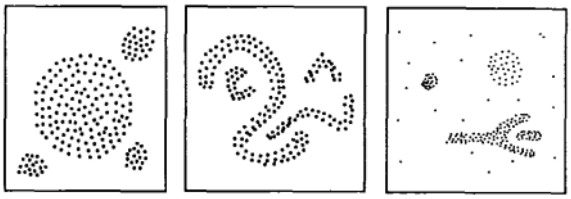

Figure 1: Data scatterings that can be clustered well using DBSCAN

Clustering adequately data with those shapes would be very challenging to `k-means`, while for `DBSCAN` it would be just natural.

In figure 2 we can observe the algorithm as a whole.

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|)
```

Figure 2: DBSCAN algorithm

`DBSCAN`, as named, relies on the density of the sample. It has two parameters: ϵ and `minpts` (minimum points). The algorithms scans the whole dataset, while evaluating the density of each sample: for each sample it counts the number of samples that are found within a radius of ϵ around it. That process is also referred as `Nearest Neighbors Query`. After that, the two following scenarios and assignments are possible:

1. If the number of the neighbors in the radius ϵ is less then `minpts`, the currently evaluated sample will be temporarily considered as noise.

2. If the currently evaluated sample has more the `minpts` neighbors inside the above ϵ, we consider that sample to be **dense**. Then This sample will be assigned to a new cluster, and a new similar repeating breadth-first search is being done on its neighbors (the samples that are within the ϵ neighborhood) that have not yet passed the process. That search is being

repeated for all the neighbors and the neighbors of them and so on, until we have processed all of the samples in the current density. All the samples that are being processed during the repeating search, that were formerly not assigned to any cluster, whether because they were classified temporarily as noise or because they have not been identified until that point - are assigned to the same cluster of the first example we started the search with.

This manner of moving "the scope", each time observing a neighborhood, and then continue to adjacent neighborhoods - allows capturing the different shapes explained before. Of course that the success of the clustering is also deeply depend on the ϵ and `minpts` parameters: overly large radius will include several different clusters and assign them as one, and might not capture a particular shape in the data like in central image in figure 1. On the other hand, if it is too small, then it might separate one cluster into several clusters when reaching to areas that are a bit less dense. A similar trade-off but in the opposite direction, happens with `minpts`. Having overly large `minpts` means that less samples will be considered as dense, and possibly we will obtain a lot of small clusters or a lot of samples that will be classified as noise, or both the possibilities together.

An important drawback of this algorithm is its complexity. It is needed to perform a nearest neighbors query for each sample in the dataset. Theoretically, we need to know the distances between all of the examples in the dataset, meaning a runtime complexity of $O(n^2)$. However, there are ways to perform it more efficiently. If we look carefully, we are scanning through all of the samples, but we perform `Nearest Neighbors Query`. In our case we will use `sklean.neighbors.KDtree`, that has an average runtime complexity of $O(nlog(n))$, and is calculated only for the first sample in a cluster. That means that the algorithm as a whole reaches a runtime complexity of $O(nlog(n))$.

2.2 IDBSCAN - Intersection DBSCAN algorithm

Following the former presented major disadvantage of `DBSCAN` runtime complexity, `IDBSCAN` suggests a smart way to sample the data such that dense leaders in the data are considered dense also in the sampled data subset. The `DBSCAN` is then applied on the data subset, and eventually we pass the clustering information for the data that was not fed into `DBSCAN`.

An important fact is that `IDBSCAN` does not have any parameters of its own, but it only uses the already needed ϵ and `minpts`, while for cases τ is needed like in `Leader*`, than it uses the very same ϵ as τ. It is possible to see the algorithm in Figure 3.

Algorithm 4: I-DBSCAN.

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

```
1  {L, F} ← Leader*(D, ε, ε) ;          /* Modified version */
2  S ← L;
3  for each l ∈ L do
4      s ← Find all followers of l in any intersection;
5      if |F_l| > minPts then
6          if |s| > minPts then
7              s ← FFT-SAMPLING(s, minPts);
8          else
9              s ← s ∪ SAMPLE(F_l, minPts − |s|);
10     S ← S ∪ s;
11 return S;
```

Figure 3: IDBSCAN algorithm

`IDBSCAN` process contains the following main steps:

1. The first step of `IDBSCAN` is applying the `Leader*` algorithm, which is explained in section 2.4. This algorithm is based on `Leader` algorithm, that is explained under section 2.3, but the difference is that it allows examples to be assigned to more than on leader. `Leader*` outputs the leaders list (`L`), those are "delegates", and followers list (`F`), which are examples that are represented by those delegates. Those leaders are inserted into a list that we will refer as `S`. Recall that the `Leader*` is called using the ϵ as both ϵ and τ parameter.

2. Then an important repeated process is performed, where we scan all the leaders and for each leaders we perform oversampling in its neighborhood in a specific manner. First we find the followers of the currently observed leader that **intersect** with other leaders, that is, they are also followers of other leaders. We will refer to the intersection as `s` (notice that it is different from capital `S` used as the list of the leaders and sampled intersection that will be later fed to `DBSCAN`).

   (a) In case that the current leader followers quantity is less than `minpts`, then `s` will be added into `S` as is.

   (b) In case that the current leader has more than `minpts` followers, it means it is a dense leader and we want it to keep being dense for the `DBSCAN` phase. Hence, we perform on of the following sampling approaches:

   - If `s` size is less than `minpts`, than we add to this `s` a quantity of `minpts`-|`s`| uniformly sampled examples from the current leader's followers list, to have an amount of `minpts`. This way we assure that the leader would not be classified as noise. If `s` has the same size of `minpts` then sampling of size 0 is performed, in other words - no sampling is being performed.
   - In the other case where the quantity of `s` is more than `minpts`, then we use FFT_sampling that is described in section 2.5 to sample them in a sparse way so they represent their leader "territory" better.

   In the completion of each loop, which is a scan of one leader followers and intersections, we end up appending the new sampled examples to `S`

3. `S` which is the sampled data subset composed by the leaders and some addition of meaningful samples are together the output if `IDBSCAN` algorithm. Then `S` is fed to `DBSCAN` algorithm. This way the complexity is expected to decrease, having the data size much smaller.

4. The last step is to assign the followers their leader's label that obtained applying the `DBSCAN` in the former step. This is done using the followers list that is the output of Leader. That algorithm is explained in section 2.3, and it is different from the followers list of `Leader*` since every follower is assigned to one leader. Otherwise, using `Leader*` we would have encountered ambiguity since a sample might have more than 1 possible label (more than 1 leader possibly).

By the end of this process each sample of the original dataset has a label - if it is a leader the label was calculated by directly applying the `DBSCAN` on subset `S`. If it is not a leader, than it has the same label of its leader.

Steps 3-4 above are not found in the pseudo-code of `IDBSCAN` since it contribution is the sampling part. However, when we want to apply the `IDBSCAN`, those last two steps are essential, and to my believe it should have appear also. I found it important, so I chose to present those steps, also having my implementation flow in that way.

2.3 Leader

`Leader` is an incremental algorithm, where we scan a dataset and find a list of leaders and a list of followers to those leaders. Each leader is a representative of a group of followers, that are in a certain distance from it - τ. The result of this algorithm is having the data divided into kind of spheres of samples, each sphere contains samples that are the followers and one leader for these followers. The algorithm is presented in Figure 4.

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}
```

Figure 4: Leader algorithm

This algorithm uses a threshold τ that is the maximum distance from which we decide to define a new leader, and the radius of the sphere where followers of a leader would be found. It has 2 nested loops. The outer one goes over all of the data, while the inner loop runs along the updated incrementally list of leaders. First we define the first example to be a leader in order to start with a 1 leader list `L`. Then we assign the first example itself to its followers list (line 2). Each leader is also a follower of itself.

For each data point, we scan all the leaders list and check if it located within less than τ from any leader. When we find a leader that is close enough to this point, we add this point to the relevant leader's list of followers and stop searching across the rest of the leaders. That means that each sample that is not a leader follows only one leader. If we scanned all of the leaders and have not found a leader that is closer than τ to the current example (completing the inner loop), then this example will be considered as leader too, and we append it to the leaders list.

This algorithm outputs `L` which is a list of leaders, and `F` which is a list of followers to each of the leaders.

The result of a leader algorithm can be visualized with the following image:

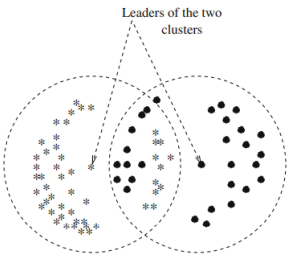

Figure 5: Leader grouping example

2.4 Leader*

`Leader*` is also an incremental algorithm based on `Leader` algorithm. It encompasses two loops over the data, and it is presented in Figure 6.

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}
```

Figure 6: Leader* algorithm

In the first loop it only finds the leaders without assigning followers, and in the second loop, it assigns the followers to each leader.

To begin, we are going through the data, assuming currently observed sample is a leader, unless during an inner loop over the leaders, we discover that it is located within a distance that is less than τ from an existing leader. In that case a flag will help us to sign that it is not really a leader, so it will not be appended to the list of leaders (lines 8-9).

The second pass over the data is a nested loop of the data and the leaders, and each sample that is less than ϵ from the one leader, is assigned to be one of its followers. This step is very important since `IDBSCAN` uses the information regarding the intersection between different leaders followers, to eventually assure that a dense leader in the original data, will be also dense in the new representation. If we want to visualize that, than the examples that are in the intersection as appears in 5, would be followers of the two leaders. This algorithm outputs, similarly to the `Leader` algorithm: `L` which is a list of leaders, and `F` which is a list of followers to each of the leaders. The list of leaders generated with both **Leader** and **Leader*** is the same list. The difference is in the followers list. In Figure 7 it is possible to get the visualized impression of the algorithm output.

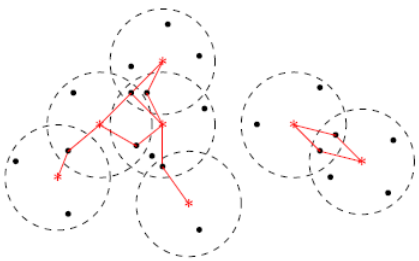

Figure 7: Leader* output: The red stars are the leaders, the black dots are the followers corresponding to each leader in a sphere around it, the red lines connect between the leaders via their intersected followers. That illustrates the benefit of this algorithm by allowing the connections between different groups leaded by different leaders.

2.5 FFT_sampling

FFT_sampling stands for Farthest-First traversal sampling. This algorithm select a random sample given a dataset, and then successively selects the farthest element from the set of previously selected elements. The benefit of this algorithm is that the chosen samples are well spread and hopefully cover more than only one intersection. The strength of selecting samples that are in more than one intersection is that they will help to conserve the continuity of the density (cluster obtained) in the `DBSCAN`, and they help more leaders having added only one sample to the subset of the data. The aspiration to get the smallest addition as possible while having representation to the real denseness of the leaders.

3 Implementation explanation

The work contains the following files/folders:

1. `main.py` where it is possible to execute the code. Instructions for executing appear in the `readme` file.
2. `readme` as explained for main.
3. `algorithms.py` that is where all the algorithms and their auxiliary functions are located.
4. `utils.py` where all the needed functions for loading data and some small calculations.
5. `datasets` that is where all the used datasets can be found.
6. `results` in case it is required to save the results of `IDBSCAN` labels.

In the implementation I tried to stick to the symbols used in the theoretical explanations. I created 3 classes that each have `fit` method and `fit_predict` method and they follow the conventions of `scikit_learn`: `DensityGeneral`, `IDBSCAN`, `DBSCAN_manual`. I will explain each class used and present some general comments and for orientation in the implementation.

3.1 DensityGeneral Class

`DensityGeneral` was created to hold the methods that are needed for applying some sampling method for delegates and then the `DBSCAN`. That is in our case `IDBSCAN`, or `Leader` algorithn followed by `DBSCAN`. This class' main contribution is for maintaining the distance matrix that is calculated by the `Leader` / `Leader*` and to avoid repeating calculations. It also contains some more functions that are shared for two possibilities mentioned.

- There are several different functions for accessing and perform calculations on the `distance_matrix`. The most important and used ones are the following:
    1. `find_specific_dist` that is aim to extract a particular distance from the matrix between two given indices. All the functions that access the distance matrix make use of this auxiliary function.
    2. `find_row` that is, if we had a square-form matrix of distances - find the row of a relevant index. Also it is possible to give this functions the columns wanted, that are the list of indices in the second argument of the function.

- Another function in this class is called `passing_predictions` that is for assigning classes to the followers according their corresponding leaders. It uses the `Leader` followers list `F_parallel`. It is parallel to the `F` list that is the followers according to `Leader*`.
- `DBSCAN` function is found under this class, while the data this is processed is subset of the data. Recall we are in a class that is used only for applications of sampling for delegates and then applying the clustering on the subset. We will see this function in the other classes too, since it process different subset of the data in each class. More details about `DBSCAN` function under the general details subsection (3.4).

3.2 `IDBSCAN Class`

This class is for handling the specific variable and methods that are used during `IDBSCAN`. It extends the class `DensityGeneral`.

- `fit` function is kind of like the main of this class. It calls the other functions and perform the predictions.
- `Leader*` algorithm:
  - In principle, `IDBSCAN` uses `Leader*` algorithm for leaders and followers finding. However as stated in `IDBSCAN` explanation (2.2), in order to eventually assign the the labels to the followers, we need to also have in parallel the followers list that is created by `Leader` algorithm. To avoid having more runtime complex, the `Leader*` encompasses also the `Leader` algorithm, and incrementally creating a list of the followers that would have been generated by the basic `Leader` algorithm.
  - `Leader*` requires as an input the data set `D`, threshold distance τ and distance ϵ.
  - The outputs are: `L` that is a list of **indices** of the leaders found. `F` a list of lists in the length of the data, such that a list of followers of a certain leader is placed in the index of the leader. For example if sample `i` is not a leader then `F[i]` will be an empty list. If a sample `j` is a leader than `F[j]` is a list that its first element is `j`, and the rest of the list are its followers. Having `L` a list of indices of leaders, we will use a lot the following syntax to approach to a list of followers of a certain leader `F[L[leader_ordered _number]]`. Lastly, it also outputs `F_parallel` that is the list of followers as would have been created by `Leader` basic algorithm for the purpose of the last step of assigning the label of the leaders to the followers.
  - This algorithm requires a distance matrix, since we need to threshold the distances in order to find the leaders and followers. This distance matrix was calculated in several different ways until converged to the final decision. I make use of the function `scipy.spatial.distance.pdist`. The matrix can then be represented in its squared form, which facilitates a lot the use of it. Not only that, but it results with shorter runtimes, since there is no need for most of the auxiliary functions created for the distance matrix. However, when executing the algorithm with datasets larger than 20,000 entries, the execution was failed due to problem to allocate the needed memory. I tried then to perform some manipulations as zeroing the lower part of it since it is symmetric so no loss of information. I did not have another possibility than to work the vectored version of the distance matrix.
- `IDBSCAN_sampling` is a function that encompasses all the sampling process that is performed. Respectively it calls to FFT_sampling that performs the explained in the theory in section 2.5. It also performs a uniform sampling if needed as explained in the theory.

- `find_intersect_followers` is for finding the common indices between current list of followers to the rest of the leaders followers.

3.3 `DBSCAN_manual Class`

- The method `neighbors_labeling` is the loop of the DBSCAN for expanding the "currently growing" cluster. It gets a list S of indices that are the current neighbors, current "scope", and check for each neighbor if it has more than `minpts` neighbors inside the radius of ϵ around it. In the positive case - this sample is added to a list that will be later go through the neighbors scanning. That is how we "conquer" more samples to be clustered in the current density.

3.4 General Details

- The implementation includes some precautions that are meant to capture errors and might raise an exception / or print details in one of the following cases:

  1. The number of samples in the subset after `IDBSCAN` should be equal to the number of leaders (output `L` of `Leader*`) summed up with the number of addition samples (from `IDBSCAN` sampling). Formally: `|S| = |nun_followers_not_leaders| + |L|`. Otherwise an error is raised presenting the length of each of them.
  2. The output of `DBSCAN` during sampled version (`IDBSCAN`/`Leader` execution with clustering), that is defined as `predictions`, should have the length of `S` (that is the data subset that is processed during `DBSCAN`.
  3. It is possible to print the number of leaders and sampled examples that are produced with the different algorithms execution by changing `verbose` to 1 (its default is 0). That is for vigilance for reasonable quantities.
  4. A function `validate_F_contains_all` that makes sure the `F/F_parallel` lists contain all of the examples in the data. Both versions of `Leader` algorithm are expected to have all of the data samples at least in one list of followers.
  5. In the end of `DBSCAN` function (inside its auxiliary function) I check if all the examples in the data/subset were classified. They all start with label of 0 and during the processing are expected to be classified differently: to be noise (label of -1) or in a cluster (integer larger than 0). In case that not all samples were classified, this will raise an error and print those examples that were not classified. It helped during debug phases.

- In `DBSCAN` the task of searching the neighbors in the specific radius has been through several optimization tries that included:

  – Manual calculation of `l2norm`.
  – `numpy` different ways of calculations.
  – `sklearn.neighbors.NearestNeighbors`.
  – `scipy.spatial.KDTree`.
  – `sklearn.neighbors.KDTree`.
  – Lastly I tried to use the distance matrix that is generated during the `Leader*` algorithm.

The last option seemed promising only theoretically. The results were a bit slower when using the distance matrix, and the ARI stayed the same. Apparently the way to access the relevant parts of the distance matrix encompass complexity themselves. This was implemented via functions to retrieve specific distances in the data subset of leaders and sampled intersection (`find_neighbors_in_radius`, `find_row`, `find_specific_dist`, found in class `DensityGeneral`). This option is still available in the code when adjusting the "neighbor_calc" of the class/instance to 0 (it is 1 by default). The Default calculation of neighbors that I found as the most adequate for my function is `sklearn.neighbors.KDTree`.

- `DBSCAN` function appears in the three classes. In the `DensityGeneral` it process the only the leaders that are the result of `Leader` algorithm. That is different than the case for `IDBSCAN` where the processed data by `DBSCAN` not only contains the leaders but also so oversampling performed. In the third case `DBSCAN` process the whole data.

- As stated, each Class has `fit` function that works according to `scikit-learn` conventions. Example:

  clustering = IDBSCAN(eps, minpts, tau, flag_save, path).fit(df)

  In addition, each one of the classes contains a function of `fit_predict` that outputs the labels directly.

  predictions = clustering.labels_

- For the parameters of ϵ and `minpts`, I followed those that are used in the paper. In the paper the authors explain they chose the parameters **arbitrarily**. Importantly, `ARI` results of sklearn `DBSCAN` comparing to the ground truth, are very low for most cases. It is not where the focus of this work is, and it not emphasized enough in the referenced paper, but it is, in my opinion important for judging the algorithm and its results. They claim that the relevant evaluation is to check the agreement with the baseline algorithm, even that baseline is a bad performer on the data with the chosen parameters. I would like to point out that I think it is not optimal to work that way. Further discussion around this is found under `Discussions` and `Conclusions`.

- The whole control on the different lists is obtained by indices. It can be a tough task to follow it, and the creation and debug were challenging mainly because of this reason. To sum up the important lists to know and are:

  1. `F` - A list with the length of the data, while each element in it is also a list that contains **indices** of followers for the specific example. Empty list in F means there are no followers, it is not a leader. That is the output of `Leader*` algorithm.
  2. `F_parallel` - similar to `F` as the output of basic `Leader`.
  3. `L` that is the list of **indices** of leaders. Its length is the number of the leaders.
  4. `followers_not_leaders` is the list of indices of the additional sampling of `IDBSCAN`.
  5. `S_idx` is the list of indices conbines the leaders and followers added by `IDBSCAN`
  6. `S_data` is the subset of the data. Not indices.
  7. `leader_labels` that can be either the labels of the leaders of the labels of all `S`.
  8. `labels_` is the list of the labels of all the data.

4 Experiments

My implementation of basic DBSCAN was evaluated using all the datasets comparing to sklearn DBSCAN and the ARI obtained was always 1.0. That means that ARI that is less than 1 is a direct results of IDBSCAN. sklearn DBSCAN works faster than mine. I tried and actually managed to improve the DBSCAN runtime a lot, however it is still slower than the already implemented sklearn version.

As explained, for the quality evaluation we compare the clusterings to the sklearn.DBSCAN clusterings as ground truth. Having that said, together with the fact that the parameters chosen by the authors are arbitrary, it is not surprising that when applying **original** DBSCAN and compare it to the ground truth, we get poor score in lots of cases.

I used for the experiments 7 different data sets while the smallest contains **4,177** entries and the largest contains **58,509** entries. I have tried to execute using a dataset that contains 100,000 entries but my laptop could not handle the computation and reached to out of memory. In the table 1 it is possible to see the datasets used and their original size (under "Original volume" column). The smallest dataset is Abalone. Then the medium ones are Mushroom, Pendigits and Letter, that are within the range of 8,000 to 20,000 samples. Eventually the largest are Shuttle and Sensorless.

4.1 Data volume comparison to baseline

First, I would like to present the range of the obtained data subsets by IDBSCAN comparing to the original dataset volume. In the following table I present the number of samples detected by IDBSCAN algorithm: The number of leaders generated by Leader* algorithm, the "Addition" which stands for the extra samples that are added to the leaders by the sampling methods of IDBSCAN, total number of samples from IDBSCAN and then comparing to the original samples volume in each data set.

Dataset	Number of Leaders	Addition	Total Samples	Original volume	Subset from data
Abalon	344	547	891	4,177	21.33%
Mushrooms	149	504	653	8,124	8.03%
Pendigits	1,154	2,074	3,228	10,992	29.37%
Letter	2,463	6,541	9,004	20,000	45.02%
Cadata	1,712	3,208	4,920	20,640	23.84%
Shuttle	590	6,077	6,667	58,000	11.49%
Sensorless	510	5,915	6,425	58,509	13.24%

Table 1: Original Datasets size compared to sampled by IDBSCAN

It is possible to see that IDBSCAN is able to diminish the quantity of samples fed into DBSCAN, in some cases to 8% of the original data volume. The average sample size after IDBSCAN is 21.76% of the original sample size. The number of samples that IDBSCAN outputs depends on the parameters, that as explained for most cases were set randomly.

4.2 Run-Time and quality (ARI) comparison to baseline

The results of the experiments are shown in the following tables. Here we can see the runtime of DBSCAN algorithm compared to IDBSCAN runtime, and ARI score that is the ARI of IDBSCAN with DBSCAN.

Dataset	DBSCAN runtime	IDBSCAN runtime	runtime difference	ARI score
Abalon	2 s	1.39 s	-30.5%	0.98
Mushrooms	17.63 s	4.54 s	-74,24%	0.94
Pendigits	10.8 s	12.61 s	+16.76 %	0.73
Letter	83.24 s	65.2 s	-21.67%	0.82
Cadata	84.3 s	42.9 s	-49.11%	0.93
Shuttle	124.88 s	32.99 s	-73.58%	0.57
Sensorless	1446.15 s	511.04 s	-64.66%	0.9978

Table 2: `IDBSCAN` Runtime and ARI compared to baseline

Observing the table above, for most of the datasets, `IDBSCAN` improve the performance in the sense that it shortens the runtime while maintaining relatively high ARI score. The results with my implementation show similar results to the original paper ones.

However There are some trends and phenomenon I would like to spotlight on:

- In the paper `DBSCAN` alone was always slower. Here I have a case for `Pendigits` dataset that `IDBSCAN` results with a bit longer runtime: 10.8 seconds with `DBSCAN` comparing to 12.61 seconds with `IDBSCAN`, which is an increase of 16%. However Observing the rest of the results, `IDBSCAN` actually improved the runtime to a larger extent.
- When comparing to the original paper results, the runtime for their `IDBSCAN` was shorter than mine for all the cases. In the extreme case with Sensorless datasets, they managed to execute `IDBSCAN` in 34 seconds, while for my implementation it took 511 seconds.
- The runtime of `DBSCAN` in the original paper, is consistently and, for most cases, significantly, **longer** than with my `DBSCAN`. In the extreme cases of Sensorless datasets their execution lasted 7,379.3 seconds while mine lasted 1,446.15 seconds. That is more than 5 times longer.
- ARI scores for `IDBSCAN` were very similar between the original paper and my implementation e.g 0.98 for Abalone, 0.99 for sensorless. However in the case the Shuttle, my implementation obtained only 0.57 while theirs 0.82. I tried to investigate the root cause without success currently. Similarly with Pendigits, they reached to 0.97 ARI while I 0.73. With that said, in cases like letter my implementation obtained ARI of 0.82 while theirs only 0.78.

In average `IDBSCAN` managed to **decrease the runtime with 0.42%** from the original runtime, while average **ARI score 0.85**.

4.3 Comparison with Other Variants of DBSCAN

The search of other already implemented models that try to overcome `DBSCAN` complexity and are comparable to `IBSCAN` was challenging. The reason it is challenging is that the evaluation of the algorithm is done by comparing the results with the ones obtained with `DBSCAN` with the given parameters, and not with the ground truth clustering. The meaning of this is that it can not be compared to some other clustering algorithm but the comparisons should be specifically a `DBSCAN` variant. Otherwise the evaluation of them via `ARI` comparing the `DBSCAN` results would not be adequate.

I decided to try also to implement the original Leader algorithm and then apply the `DBSCAN` on the leaders, eventually passing the followers the leaders labels. I executed this experiment

using the `GeneralDensity` class. Eventually I did not count it in the results since it has no significant output, a poor `ARI` score of 0. Clearly it is not working this way. It is possible that the τ parameter that was chosen by the authors was not adequate for this algorithm. Also, as claimed in reference number [4], "Rough-DBSCAN: A fast hybrid density based clustering method for large data sets" , basic `Leader` algorithm is not very effective for this purpose.

The other option was to implement the `Leader*` without performing the extra sampling of the `IDBSCAN` is redundant, since that will results with exactly the same as original `Leader*` algorithm. That is because the main contribution of `Leader*` is the fact that a sample can be a follower of more than one leader, to later take advantage of the intersections.

I would have tried to compare m results with the ones of `Rough DBSCAN` and `Rough* DBSCAN`, but did not find an implementation of those in the web.

I managed to encounter several variants of `DBSCAN` such as `l-DBSCAN`, `NQ-DBSCAN`, `V-DBSCAN` etc. that could be compared with `IDBSCAN`. However I was not able to execute some of them, and the ones that I was able to are: `Hierarchical-DBSCAN`, `Spatio-Temporal-DBSCAN`. In the following section I will give a brief explanation of each of the two, mainly about their contribution/improvements over `DBSCAN`.

4.3.1 HDBSCAN: Hierarchical DBSCAN explanation

This is an algorithm that performs `DBSCAN` over varying ϵ (radius) values and integrates the result to find a clustering that gives the best stability over ϵ. The authors propose an hierarchical clustering method where a simplified tree of significant clusters can be constructed from a clustering hierarchy. They suggest a cluster stability measurement, and a way to obtain the optimal solution given the formulation based on this measurement. The method looks for a cluster hierarchy shaped by the multivariate modes of the underlying distribution. Instead of searching for a particular shape, it searches for regions of the data that are denser than the surrounding space. This allows `HDBSCAN` to find clusters of varying densities (unlike `DBSCAN`), and be more robust to parameter selection. The algorithm finds the ϵ according to the k-nearest neighbor distance. In practice this means that `HDBSCAN` returns a good clustering fast with little or no parameter tuning.

4.3.2 ST-DBSCAN: Spatio-Temporal DBSCAN explanation

Another algorithm based on `DBSCAN` that analyses the data via 3 aspects: core objects, noise objects, and adjacent clusters. This algorithm has the ability to detect clusters according to non-spatial, spatial and temporal values of the objects. For the tabular datasets used in this paper, this algorithm might be over-forcing. However, it is interesting to see how it behaves comparing to `IDBSCAN`. Spatial–temporal data refers to data which is stored as temporal slices of the spatial dataset. This kind of algorithm can be used for applications as geographic information systems, medical imaging, and weather forecasting. The authors explain the three main improvements in this algorithm over the `DBSCAN`:

- The algorithm can cluster spatial–temporal data according to its non-spatial, spatial and temporal attributes.
- They introduce a new attribute which is the `density factor`, which is assign to each cluster, and is aim to aid when density is different between clusters.
- The marginal examples, that are in the border between clusters, can vary a lot comparing two sides of a border. The proposed algorithm handles with that by averaging the clusters

values and compare this average with a new example that is candidate to be clustered the same.

4.4 runtime comparison

Dataset	IDBSCAN runtime	HDBSCAN runtime	ST-DBSCAN runtime
Abalon	1.39 s	**0.19 s**	0.77 s
Mushrooms	**4.54** s	7.87 s	6.85 s
Pendigits	12.61 s	**1.05 s**	2.43 s
Letter	83.24 s	**6.43 s**	11.21 s
Cadata	84.3 s	**1.93 s**	36.73 s
Shuttle	124.88 s	**7.77 s**	error: out of memory
Sensorless	1446.15 s	**98.92 s**	error: out of memory

Table 3: Runtime comparison: `IDBSCAN`, `HDBSCAN`, `ST_DBSCAN`

4.5 ARI comparison

Dataset	IDBSCAN ARI	HDBSCAN ARI	ST-DBSCAN ARI
Abalon	0.98	0.96	**1.0**
Mushrooms	0.94	**1.0**	**1.0**
Pendigits	**0.74**	0.57	**0.74**
Letter	**0.82**	0.67	0.78
Cadata	0.93	0.7	**1.0**
Shuttle	0.57	**0.99**	error: out of memory
Sensorless	**0.9978**	**0.9967**	error: out of memory

Table 4: ARI comparison: `IDBSCAN`, `HDBSCAN`, `ST_DBSCAN`

Observing the results we can notice some trends:

- `HDBSCAN` was faster for most cases significantly. Only for Mushrooms dataset, `IDBSCAN` was faster.
- `ST-DBSCAN` obtained highest ARI for most cases but not a big difference from the other two. Notice that when it reaches ARI of 1.0, the other two obtained above 0.93.
- `IDBSCAN` shows impressive ARI scores for most cases: For Pendigits, obtained ARI by `IDBSCAN` and `ST-DBSCAN` is the same which is 0.74. For Letter dataset `IDBSCAN` is higher than the other two. In general `IDBSCAN` outperformed `HDBSCAN` in the ARI aspect, except of the Shuttle case. For the "problematic" Shttle case where `IDBSCAN` reaches only 0.57 ARI, `HDBSCAN` manages to get 0.99.
- `ST-DBSCAN` algorithm was not able to handle the large datasets of 58,000 examples and more for memory reasons.

5 Discussion

- **Parameters Effect**.

  As I have claimed before, the parameters choice is crucial for `DBSCAN` good performance. In the presented paper the authors explain that the parameters they chose were random,

emphasizing that the important thing is the ARI comparison to the ground truth DBSCAN. With that said, what if the chosen parameters result with awful clustering by DBSCAN itself, and in extreme cases result with not-representative result e.g 90% of the samples are detected as noise, or in other case, one giant cluster that covers most of the samples in the dataset. The choice of ϵ and minpoints affect the leader* and IDBSCAN algorithms directly: the decision whether current example is a leader and later the decision of which examples in the leaders intersection will be processed with the DBSCAN, are two crucial decisions - both for the runtime and for the correctness of the algorithm.

In my opinion random choice of parameters does not fully prove the correctness of the algorithm and its benefits. If the DBSCAN results itself are poor with those parameters, it is not assured that with well adjusted parameters, we would still get the reduction of run-time.

It is true that adjusting DBSCAN parameters can be a difficult task, however I think an important step for future work is to perform the experiments again after parameters adjustments.

- The contribution of IDBSCAN for the largest datasets was significant in terms of runtime, that decreased by an average of 69.12% (for both Sensorless and Shuttle).
- When comparing the algorithm to **competitors**, it is hard to detect trends that derived from sample size. IDBSCAN seem to be less efficient from the competitors in most cases runtime-wise. It is thought important to point out that we are using ARI with DBSCAN results. Those two algorithms HDBSCAN and ST-DBSCAN where meant to outperform the original DBSCAN, also in the clustering decision. That means that having them with less ARI then others, does not necessarily implies that they performed worse, or better.
- The runtime presented in the original paper is significantly lower than for my implementation. The reason for that could be the use of square-form of distance matrix. Otherwise it is possible that there are some other ways to optimize my code that I could not figure out in the limitation of time.

  It is possible that with other stronger environment I would have been able to test larger datasets, or use the square-form of the distance matrix. That could result with better performance runtime-wise.

  Also, it is to be assumed that they optimize the IDBSCAN in order to reach such a short runtime. Those optimizations are not mentioned in the paper at all. It would be very interesting thought.
- The unclear runtime of the original paper for DBSCAN raises some question marks. This paper was published 2 years ago, meaning they were able to use the sklearn DBSCAN as baseline. I suspect that their baseline of DBSCAN were based on more primitive method that required the original $O(n^2)$ runtime complexity.
- The Leader* algorithm addition of the second loop, that does not exist in the original Leader, is claimed in the paper to be redundant runtime addition. It is true that in the results we saw faster execution for IDBSCAN, but from my experience, this runtime addition is not redundant. In the larger datasets it was more notable.
- Sklean.neihgbor.KDtree is implemented in cythom which makes use if C code which is faster the python, while the textttscipy.spatial.KDTree is pure python code. That explains the results of my preliminary experiments for deciding the best method.

- The benefit of having a dense leader in the original data as dense in the data subset is that it assures that leaders that are dense will not get classified as noise. Moreover, it also indirectly assured that leaders that share followers will be clustered in the same group. That explains to some extent why my try to apply the basic `Laeder` and then `DBSCAN` obtained an `ARI` of 0 since we bypassed the phase of allowing intersections like in `Leader*`. In practice it caused that almost all the leaders were classified as noise. That actually demonstrates the `IDBSCAN` sampling method works well.

6 Conclusions and possible future work

`IDBSCAN` is a method that allows a quality of clustering that is similar to the original `DBSCAN` algorithm, but allows to apply the clustering on much larger datasets and faster.

We saw though, that this algorithm has its drawbacks comparing to other algorithms. More specifically, the correctness of the algorithm is not always promised. A further investigation of the cases where the `ARI` score is low. Understanding the rout cause could help to adjust/change the algorithm accordingly (or to find a bug). We know that reason is not stack in varying densities of clustering since we compare to `DBSCAN` which is not robust to this varying densities.

Comparing my `DBSCAN` to `sklearn DBSCAN`, the already implemented is much faster than mine. I guess that makes sense, since my tries to optimize it, are of course limited by time and resources, and of course that the library of `sklearn` is very optimized. It is possible that if I would have used it for the experiments, we would have witnessed other results.

In the short term, further investigation should be done to assess `IDBSCAN` in an absolute way, instead of comparing only to `DBSCAN`. That will require a research for each one of the datasets to adjust the parameters to the adequate ones. There are some ways to find the right parameters for `DBSCAN`, however they do not necessarily work. Moreover, it is possible that when applying `IDBSCAN` as a preliminary step before `DBSCAN`, we find that other parameters would be preferable. That could also be an interesting direction to check. If I had some more time to work on this project, I would complete my research to adjust the parameters better to the different datasets and then compare the `IDBSCAN` results to the ground truth. Of course that it would also allow a wider comparison with other clustering algorithms that are not necessarily derivatives of `DBSCAN` such as OPTICS, DENCLUE, or even clustering algorithms that are not based on density such as k-means and its derivatives.

`IDBSCAN` is able to represent the problem of leaders and followers, but also maintaining more specific and important characteristics of the data, like which leader is denser, and how "well spread" is it territory. That is an interesting approach that looks promising.

Taking this property of `IDBSCAN` one step forward could be optimizing the `Leader*` algorithm in a way that allows capturing less leaders but with sufficient information regarding its density.

A possible optimization could be taking advantage of the already found intersections, and perform merging between leaders' territory. That will results with shapes that are more flexible than the `Leader*` circles. In fact, it could even get the output of the `Leader*` to be closer to `DBSCAN` output itself, since the properties that the samples need to hold are similar for both cases. This idea can be visualized pretty well in the output figure of `Leader*` (figure 7). The red lines should help to connect between spheres to merge them. Then in order to represent the **density** of them we can use `FFT_sampling` on the followers of each group, and that will be added to the leaders, to be then fed into the `DBSCAN`. Having smaller number of leaders, with larger, more representative groups, and introducing the extra sampling to assure the denseness

of the leaders, might perform a lot faster.

Another option that I thought could benefit, is to have the addition of samples from the intersection more monitored. We can see in the algorithm (figure 3) that the way to decide whether a leader is dense is to threshold its number of followers. The threshold is `minpts`, and when the followers number is larger, we add to this leader a quantity of `minpts` followers, to represent its density in `DBSCAN`. Notice that there are of course leaders that can withstand the threshold but vary a lot in their level of density. It is possible to have some kind of leveling of density to leaders by simply adding some more thresholds parallel to that in line number 6, which can be **factorized** `minpts`. For example if the number of followers is larger than twice `minpts`, the addition of followers will be accordingly, of twice the size of `minpts`. This enlarges the complexity of the algorithm only in the sense that is adds more samples, but it might be interesting to check. It is expected to be more representative. Maybe that is what `IDBSCAN` is missing to get more correct labeling in `DBSCAN`, and get a higher `ARI`.

Combining `IDBSCAN` with some kind of dimensionality reduction, such as `PCA`, could be a nice direction to check. In `IDBSCAN` we are also measured by the distance calculations, and having less attributes, could very much help. This way we use the important attributes (less dimensions), and less examples, meaning `DBSCAN` potentially can run much faster.

Bibliography

[1] Diego Luchi, Alexandre Loureiros Rodrigues, Flávio Miguel Varejão, Sampling approaches for applying DBSCAN to large datasets,Pattern Recognition Letters, Volume 117, 2019,
(https://doi.org/10.1016/j.patrec.2018.12.010.)

[2] Martin Ester, Hans-Peter Kriegel, Jiirg Sander, Xiaowei Xu, A Density-based Algorithm for Discovering Clusters in large spatial Databases with Noise, 1996.

[3] LIBSVM Data: Classification, Regression, and Multi-label.
(https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/).

[4] P. Viswanath, V. Suresh Babu, Rough-DBSCAN: A fast hybrid density based clustering method for large data sets, Pattern Recognition Letters,Volume 30, Issue 16, 2009,
(https://doi.org/10.1016/j.patrec.2009.08.008.)

[5] McInnes et al, (2017), hdbscan: Hierarchical density based clustering, Journal of Open Source Software, 2(11), 205,
(doi:10.21105/joss.00205 https://github.com/scikit_learn_contrib/hdbscanar=2017)

[6] L. McInnes and J. Healy, "Accelerated Hierarchical Density Based Clustering," 2017 IEEE International Conference on Data Mining Workshops (ICDMW), 2017, pp. 33-42, doi: 10.1109/ICDMW.2017.12.

[7] Derya Birant, Alp Kut, ST-DBSCAN: An algorithm for clustering spatial–temporal data, Data & Knowledge Engineering, Volume 60, Issue 1, 2007, Pages 208-221, ISSN 0169-023X, https://doi.org/10.1016/j.datak.2006.01.013.

[8] Campello R.J.G.B., Moulavi D., Sander J. (2013) Density-Based Clustering Based on Hierarchical Density Estimates. In: Pei J., Tseng V.S., Cao L., Motoda H., Xu G. (eds) Advances in Knowledge Discovery and Data Mining. PAKDD 2013. Lecture Notes in Computer Science, vol 7819. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-37456-2_14

[9] Pepe Berba, (2016), 'Understanding HDBSCAN and Density-Based Clustering', *Towards-datacience*, Jan 17, 2020. https://towardsdatascience.com/understanding-hdbscan-and-density-based-clustering-121dbee1320e