Data mining

Assignment 2: DBSCAN and FA-DBSCAN

Submitted to:
Prof. Mahmoud Sakr
INFO-H423-0-201819 : Data mining

Submitted by:
Carlos Martinez Lorenzo (000477671)
Haftamu Hailu (000472133)
Ioannis Prapas (000473813)
Sokratis Papadopoulos (000476296)

November 2018

Publication understanding

Clustering analysis has been a very popular field for research already for many decades and it is getting even more attention as we are constantly moving to an even more data-producing world, increasing the clustering complexity. DBSCAN, being a common approach for density-based clustering was an interesting algorithm to dive into and code.

However, even though practically DBSCAN runs in $O(nlogn)$ time, the theoretical worst-case scenario is $O(n^2)$, which is restricting when experimenting with very large datasets. That is a good motive given to the research community to create DBSCAN alternatives that can guarantee faster and consistent results. Already, some such algorithms, has been established: Sample-based (IDBSCAN, FDBSCAN) and grid-based (GF-DBSCAN, GriDBSCAN). However, none of the above can guarantee both exactly same clustering and $O(nlogn)$ complexity, which is the gap the assigned paper is trying to cover, introducing a new algorithm: FA-DBSCAN.

As shown by the experiments on the paper, testing with datasets up to 25,000 points (as original DBSCAN is low on large datasets) it is obvious that the new algorithm outperforms DBSCAN, being two times faster for uniform fill data. That is explained by the fact that in these kind of datasets the new algorithm computes 20% of the distance calculations the original algorithm performs. It is an expected result taken into account the way it is structured: skipping calculations when finding a cell of at least MinPoints (marking them immediately all as core) and skipping calculations when minPoints is reached on neighbors search. Specifically for gaussian discs, we notice that there is a speed improvement of scale 6 because gaussian discs display high number of cells with more than minPoints. That is very beneficial for FA-DBSCAN as it is taking great advantage of its grid structure, resulting in performing only 4% of the computations the original DBSCAN does. Lastly, for strips data FA-DBSCAN achieve 1.3 times faster time than original DBSCAN on uniform data fill and 2.5 times faster on gaussian discs.

Regarding how eps affects performance, it is clear that while for original DBSCAN eps is not an important factor, for FA-DBSCAN it is, because the larger the eps the more cells will have at least minPoints and as a result the more computations will be skipped.

Regarding minPoints, again original DBSCAN is not sensitive on its variance. However, FA-DBSCAN displays slower performance as minPoints value goes up because then less cells will have sufficient points to become core and skip distance calculations. Same applies for neighbor distance calculations. But after a specific minPoints threshold when all points result in one cluster, FA-DBSCAN is no longer sensitive to minPoints increment.

As a conclusion, the greater the eps and the smaller the minPoints, the faster the new algorithm is. To reach the necessary values on proper cluster creation, it is advisable to start with high value on eps and low on minPoints and then lower eps and increase minPoints accordingly.

Implementation

Java was used as the programming language in coding both original DBSCAN algorithm and the faster FA-DBSCAN version as described on master thesis of Ade Gunawan (A faster implementation for DBSCAN). Here is some overall points regarding our code:

- The **Main** class is only used to call the version of the algorithm (DBSCAN, FA-DBSCAN) the user specifies along with the necessary parameters (eps, minPoints, dataset).
- An abstract class **Scan** is implemented in order to serve as a base class with common elements of the two different algorithms (DBSCAN, FA-DBSCAN).
- Part of Scan class is the function on **reading dataset**, which should follow these rules:
    - Include a header on first row
    - Values are separated by tab "\t"
    - First column is used for IDs
    - Second and third columns are used for X and Y dimensions accordingly, on double type.
    - Points' dimensions may display negative values.

ID	x	y
1	31.95	7.95
2	31.15	7.3
3	30.45	6.65
4	29.7	6
5	28.9	5.55
6	28.05	5
7	27.2	4.55

- **Point** class is used to store the pointID, X-value and Y-value along with an attribute to store the clusterNum of the cluster the point belongs to and a **PointLabel** enumeration to store whether the point is a Core, Border or Noise. Also, it includes a *getDistanceFrom(Point p)* function, used to compute euclidean distance between points.
- **Cluster** class is used to store the different clusters found during the algorithm execution. It has an id attribute for cluster number and a Set of Point instances to store the points within it. It also includes a function isInCluster to check if a given point is part of a cluster.

DBSCAN

Simple algorithm implementation in $O(n^2)$ time. Following the algorithm showed in class and consulting Wikipedia (https://en.wikipedia.org/wiki/DBSCAN) and some youtube videos for better understanding of the algorithm we have implemented the simple DBSCAN in Java. We mainly use three classes: **DBSCAN** implementing the algorithm logic, **Cluster** storing the points' clusters and **Point** for the 2 dimensional points representation.

A DBSCAN instance is created when this algorithm is chosen for execution. It is getting constructed with eps, minPoints and filename as parameters to fill the values in its corresponding attributes. Then scan function is called to execute the algorithm. It begins with a loop over every point in our dataset. We search for all nearby points using regionQuery function, which returns all neighbour points respecting eps and minPoints thresholds. If the number of neighbor points are not sufficient (less than minPoints) then the point is marked as a Noise. Else, if its size is sufficient, point becomes a core and we call expandCluster function to further expand this newly-created cluster with its points' neighbours.

ExpandCluster function iterates over each point of the neighborhood of current point:

a) It marks it as a core if that point's neighborhood also meets the minimum value (minPoints) and then merges it into the current cluster.
b) It marks it as border if that point's neighborhood does not meet the minimum value (minPoints) and then merges it into the current cluster.

This algorithm complexity is $O(n^2)$, but it can be improved to $O(nlogn)$ if an index like BTree is used when computing neighbors of a point. However, this is out of scope for this assignment.

FA-DBSCAN

A faster grid-based algorithm that runs on theoretical complexity of $O(nlogn)$ in worst case. FA-DBSCAN logic is implemented on **FADBSCAN** class, using **Grid** & **Cell** classes (for grid creation) and of course the rest of classes we had already created for DBSCAN. The algorithm is structured into 4 steps:

a) construct grid and assign points into corresponding cells
b) determine core points
c) merge clusters
d) determine border points

For constructing the grid we created **Grid** class that is basically a hashmap of integer (to store the key) and a **Cell** instance. Cell itself is a list of Point instances, including getNearestPoint and setClusterNum function which sets the same cluster number both to the Cell itself but also to every point it contains. Grid class contains a setPointInCell function to set a point into the corresponding cell based on the hash result. Also, it includes the calculateNeighboringCells function, used to identify the neighbor cells for a given cell as described on the assigned paper.

Regarding the 4 algorithmic steps, we followed the pseudocode available on paper's description and adjusted into java specifics. We noticed that there were some unclear points on this new algorithm implementation that we overcame through our general understanding of the algorithm and of course testing. Plus, there was no pseudocode provided for merging clusters step.

Execution starts on *grid construction*, an empty hashmap ready to receive data on its cells. We pass each point of the dataset through the hash function and assign it to the appropriate cell.

Then, we start *determining core points* by iterating through our non-empty grid cells. If the size of the list of a cell is more than minPoints, we set all points of the cell as core and set the cell as a cell that has a core point, else if the cell has at least one point we iterate the cell points. For each point of this cell we calculate its neighbor cells, which are the ones that potentially contain neighbor points (points that are distanced in less than eps). We iterate through all neighbor cells up until the minPoints value is reached or there are no more cells to examine. If the point has

sufficient neighbors (at least minPoints) then we set it as a core point and we set its cell as a cell that has a core point.
Following step is *merging clusters*. We run an iteration over our hashmap and if current cell has a core point, we calculate its neighbor cells and for every neighbor cell that has a core point we call findNeighborCluster function which decides on cluster merging. Inside findNeighborCluster function we search for the every point of current Cell with every point of neighbor cell and merge clusters accordingly. For ease of explanation of our algorithm we define:

- pointA: the current point we are examining
- pointB: the neighbor cell's point
- clustered: point that already belongs to a cluster
- unclustered: point that has no cluster assigned yet

We implemented the logic of following 4 cases:

a) If both pointA and pointB are unclustered we create a new cluster and assign both points' cells into this newly created cluster number.
b) If pointA is clustered, but pointB is unclustered, then we assign pointA's cluster number to all points of pointB's cell.
c) If the pointA is unclustered, but pointB is clustered, then we assign pointB's cluster number to all points of pointA's cell.
d) If both pointA and pointB are clustered but belong in different clusters, then we set pointA's cluster number to all points that have pointB's cluster number.

Lastly, we *determine border points* and noise. This step starts with an iteration over all non-empty cells of the grid. Then if the cell has no core points we search for each of its points the nearest point among all neighbors. If the nearest point is either not found (which means that all neighbor cells are empty) or it is distanced more than eps value, then we set the current point as a noise. Else, we set it as a border point and set the nearest-neighbor-point's cluster to it.

Experimental comparison

We performed our experiments using a computer with the following specifications.

- Processor: Intel Core i7-770HQ CPU @2.80GHz
- RAM: 16GB
- Operating system: Windows 10 64-bit

We have used RapidMiner as source of truth to verify our results, making sure that the output is identical for every dataset and parameters for all 3 runs (DBSCAN, FA-DBSCAN, RapidMiner-DBSCAN). This task has been quite tough, as in the beginning our algorithms appeared slightly different results. We reasoned this variation due to points that may belong in different clusters (DBSCAN has been noticed by research community to not always provide the same clusters), though the continuous such examples alarmed us to debug our code, resulting in achieving complete homogeneity on the results of our algorithms and rapidminer.

We tested our approach with different kind of data sizes and cluster shapes, our sources were:

- University of Eastern Finland: http://cs.joensuu.fi/sipu/datasets
- Philipps-Universität Marburg: https://www.uni-marburg.de/fb12/arbeitsgruppen/datenbionik/data

After experimenting with different values of eps and minPoints for each dataset, we concluded on the below values, as they produced the expected clustering result (as shown on the Appendix). We executed every algorithm multiple times over the same dataset in order to get the average running time to level out fluctuations due to other processes. Below you can find a table with all average running times of our algorithms for all datasets.

Dataset	eps	minPoints	# of points	DBSCAN	FA-DBSCAN
Input (lab)	10.0	5	165	17ms	14ms
Spiral	1.2	3	312	23ms	20ms
Lsun	0.5	4	400	32ms	20ms
Target	0.5	4	770	130ms	27ms
Aggregation	2.0	4	788	57ms	29ms
TwoDiamonds	0.119	4	800	50ms	32ms
WingNut	0.29	3	1016	82ms	32ms
Input-5000	0.5	5	5000	764ms	100ms
Input-100000	35.0	5	100000	9333825ms (2.6 hours)	4252ms (4.2 sec)
Input-1000000	20.0	5	1000000	25959760ms (7 hours)	19789ms (19.8 sec)

We output the table into 2 graphs to get a visual understanding.

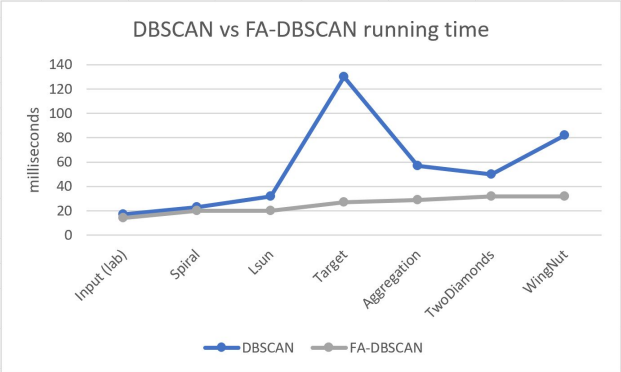


Figure 1a) Running time on low sized data

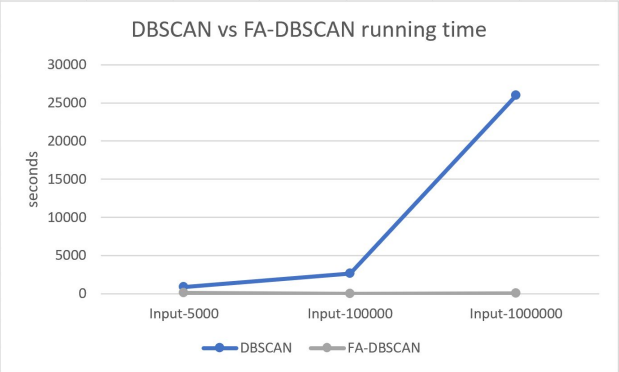


Figure 1b) Running time on large sized data

As displayed on the figures, FA-DBSCAN outperforms the original DBSCAN algorithm in every case. As the size of the dataset goes up, we observe that the difference in speed is bigger and that is expected as DBSCAN runs in $O(n^2)$ while FA-DBSCAN complexity is $O(nlogn)$. We also notice that even though Target shape is a small dataset it took original DBSCAN relatively significant time to compute its clusters. Our explanation is that the second (red) class on target dataset has big length across the 2-dimensional graph and as a result the expandNeighborhood function will take significant time to complete.

Although the original DBSCAN algorithm is not sensitive to either eps or minPoints values, FA-DBSCAN shows sensitivity in both parameters. As displayed on the below graphs, DBSCAN performance is stable, while FA-DBSCAN performs poorly as the eps value is getting smaller and also as minPoints value get higher, until a certain value where onwards it becomes stabilized. That is why it is advisable on your experiments to start with high value on eps and low on minPoints and then lower eps and increase minPoints accordingly until you form good clusters.

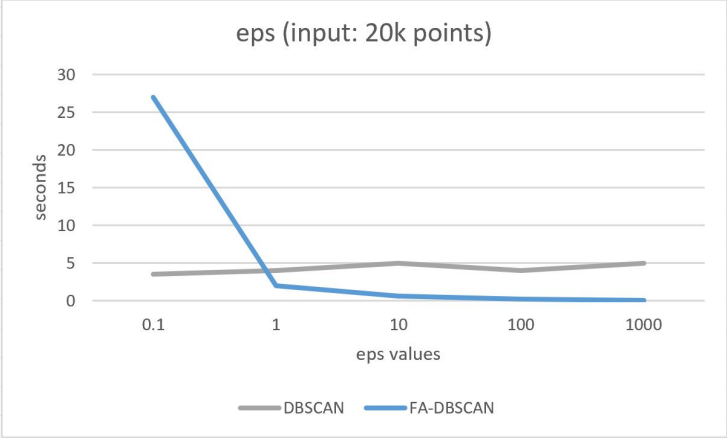


Figure 2) Running time of DBSCAN and FA-DBSCAN on different eps values

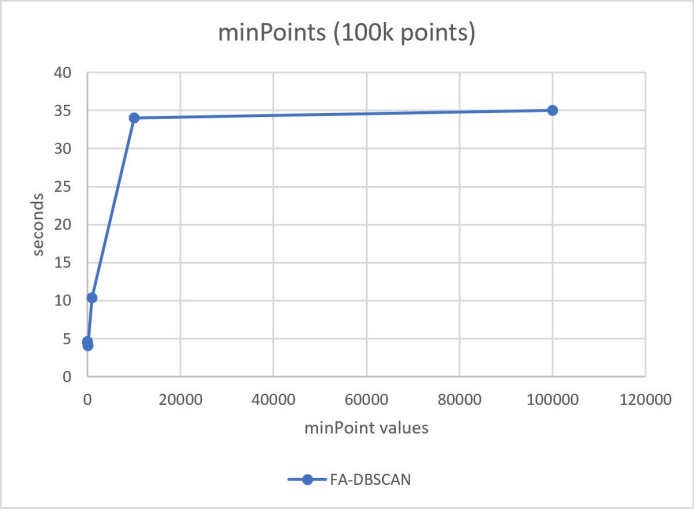


Figure 3a) Running time on different minPoint values for FA-DBCSAN on 100k points dataset

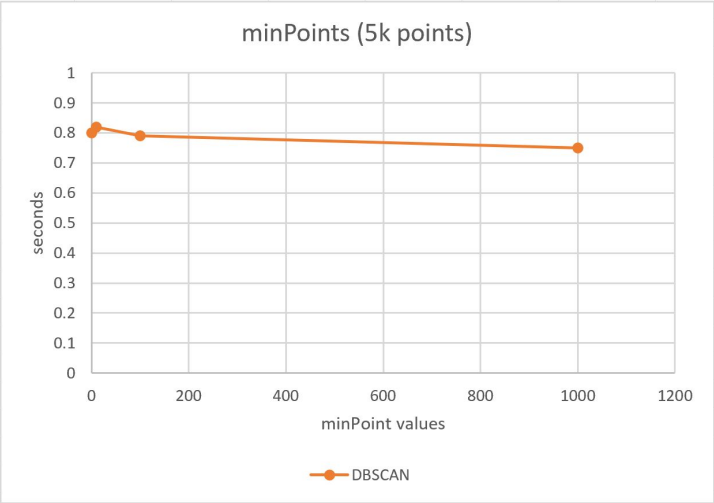


Figure 3b) Running time on different minPoint values for DBCSAN on 5k points dataset

User guide

In order to run our algorithm you can execute the jar file through command line.

Program accepts four (4) arguments:

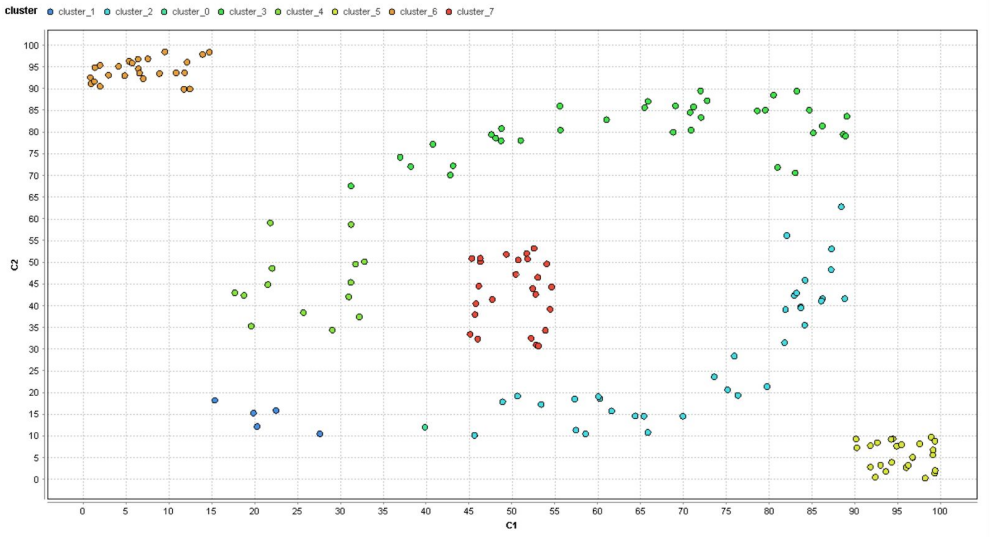
1. the algorithm that you want to run (1=DBSCAN, 2=FA-DBSCAN)
2. the value of EPS
3. the value of MinPoints
4. the dataset you want to use (input.txt , input-5000.txt , input-100000.txt)

Example command:

```
java -jar DBSCAN.jar 2 20 5 input-1000000.txt
```

APPENDIX

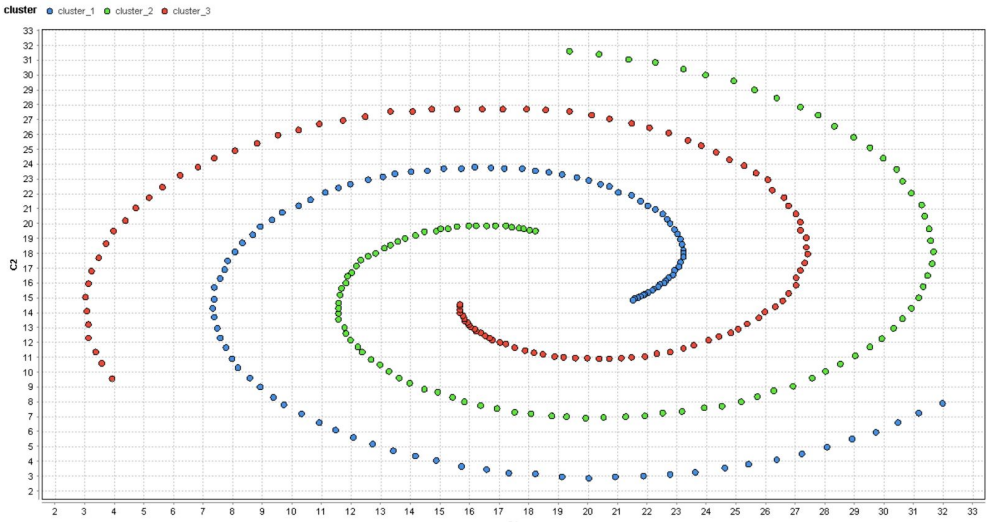
Input (lab) with eps: 10.0 and minPoints: 5



Total points: 165
Noise: 1 points
Total clusters: 7

Cluster_0: 5 points
Cluster_1: 34 points
Cluster_2: 36 points
Cluster_3: 14 points
Cluster_4: 25 points
Cluster_5: 25 points
Cluster_6: 25 points

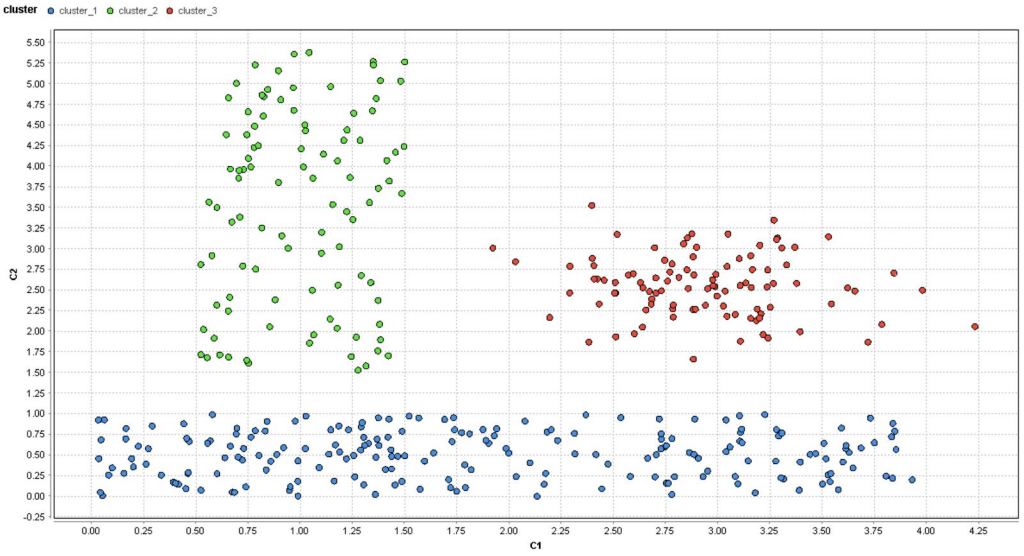
Spiral with eps: 1.2 and minPoints: 3



Total points: 312
Noise: 0 points
Total clusters: 3

Cluster_0: 105 points
Cluster_1: 106 points
Cluster_2: 101 points

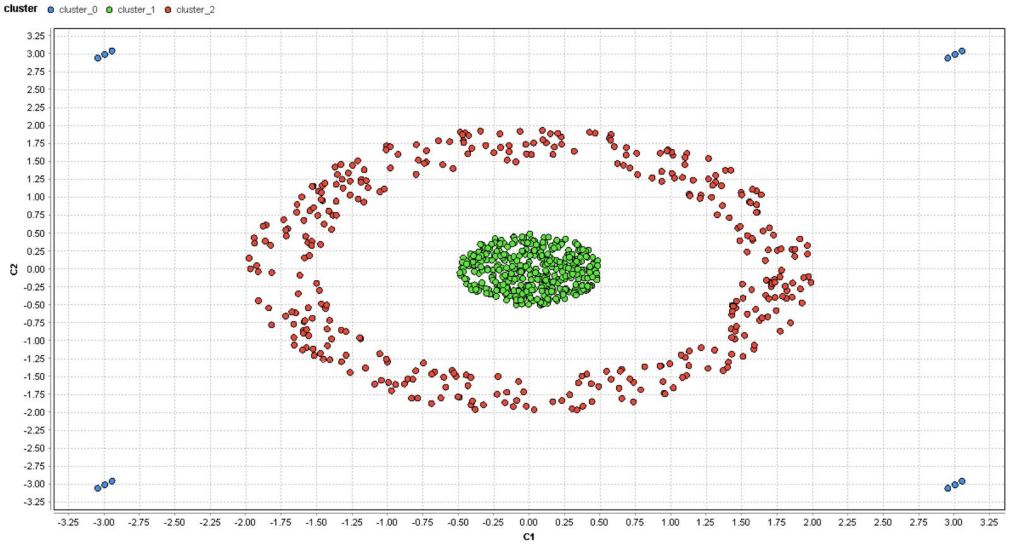
Lsun with eps: 0.5 and minPoints: 4



Total points: 400
Noise: 0 points
Total clusters: 3

Cluster_0: 200 points
Cluster_1: 100 points
Cluster_2: 100 points

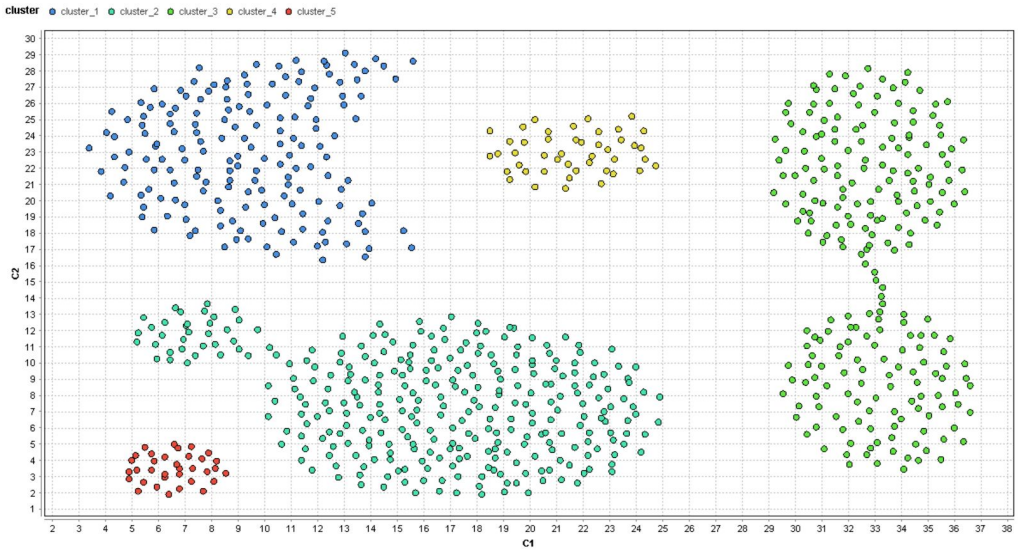
Target with eps: 0.5 and minPoints: 4



Total points: 770
Noise: 12 points
Total clusters: 2

Cluster_0: 395 points
Cluster_1: 363 points

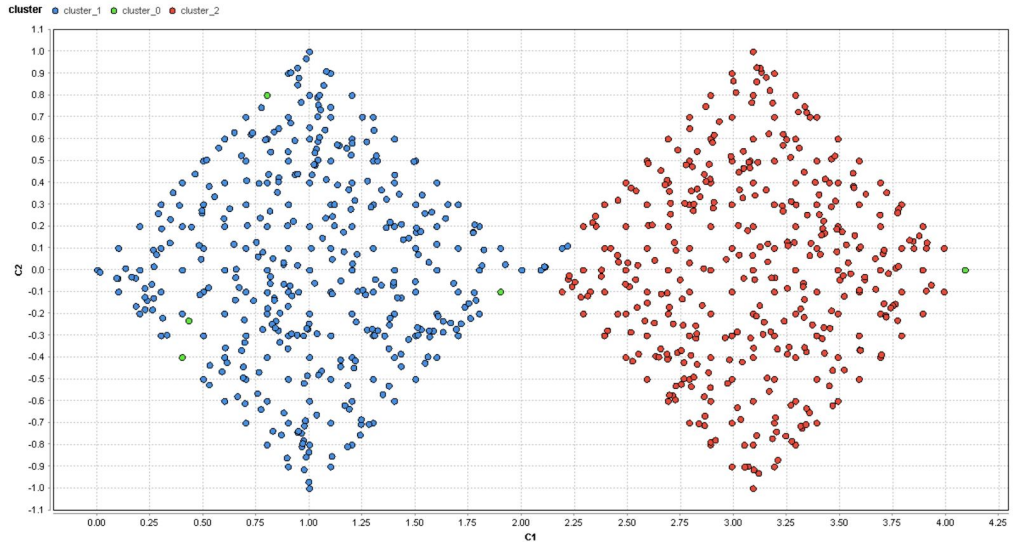
Aggregation with eps: 2.0 and minPoints: 4



Total points: 788
Noise: 0 points
Total clusters: 5

Cluster_0: 170 points
Cluster_1: 307 points
Cluster_2: 232 points
Cluster_3: 45 points
Cluster_4: 34 points

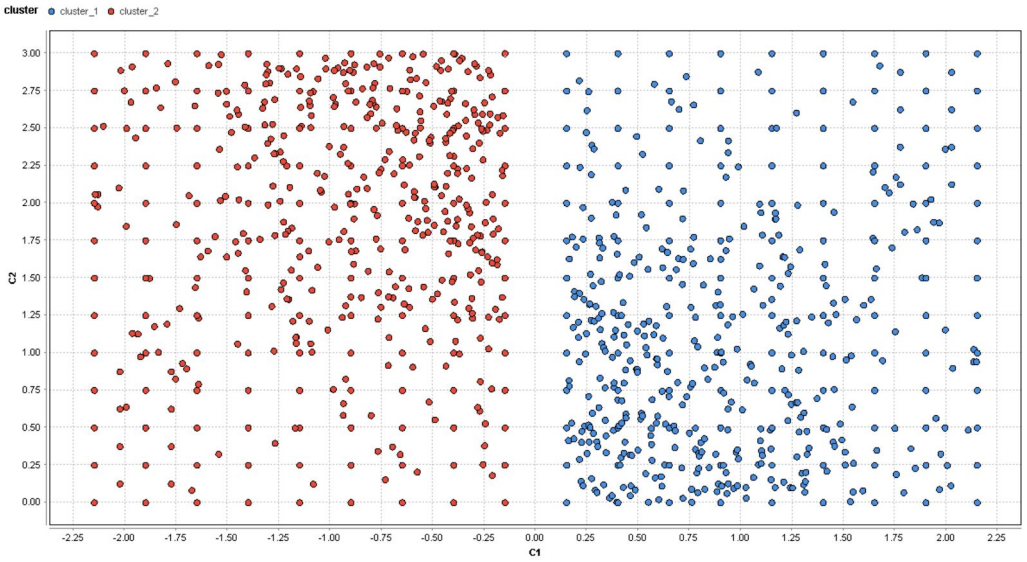
TwoDiamonds with eps: 0.119 and minPoints: 4



Total points: 800
Noise: 5 points
Total clusters: 2

Cluster_0: 401 points
Cluster_1: 394 points

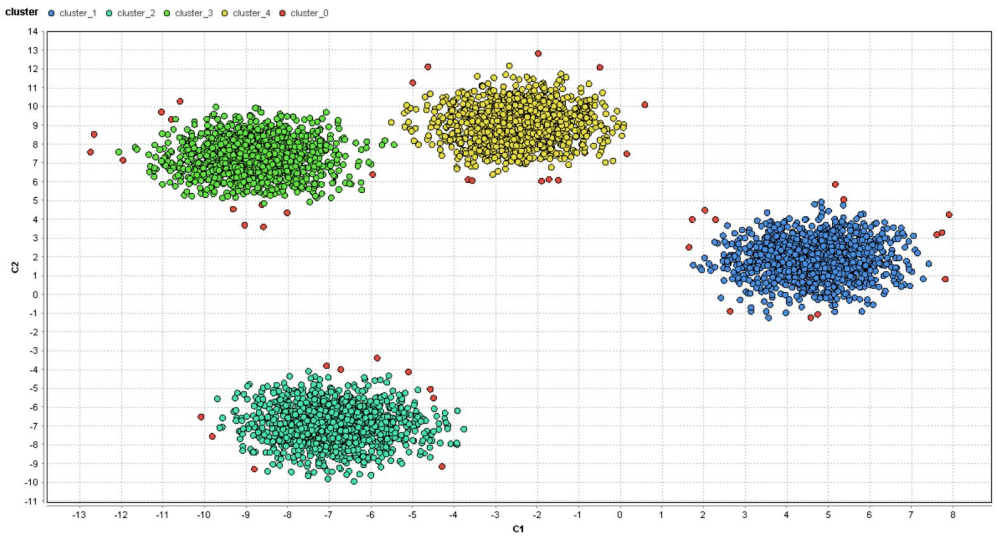
WingNut with eps: 0.29 and minPoints: 3



Total points: 1,016
Noise: 0 points
Total clusters: 2

Cluster_0: 508 points
Cluster_1: 508 points

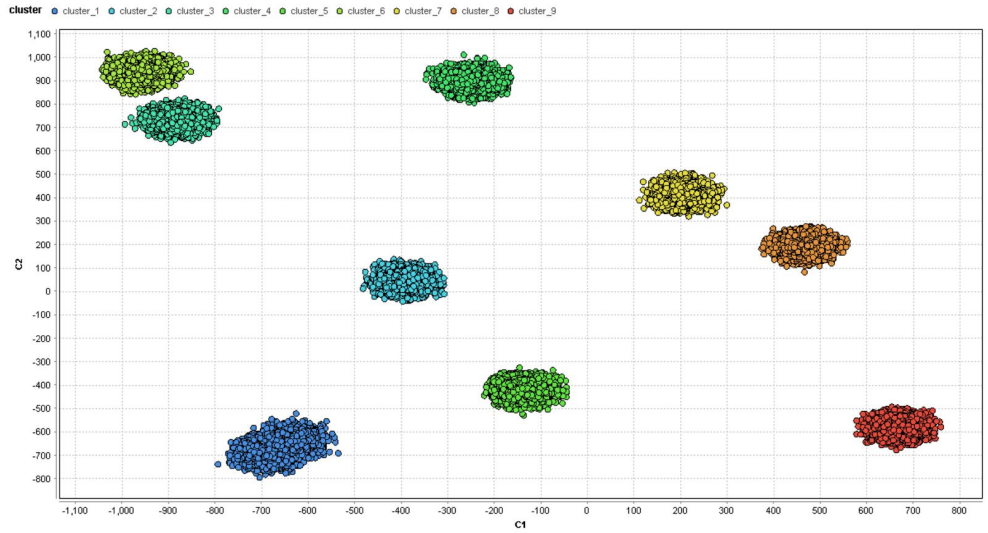
Input-5000 with eps: 0.5 and minPoints: 5



Total points: 5,000
Noise: 46 points
Total clusters: 4

Cluster_0: 1237 points
Cluster_1: 1240 points
Cluster_2: 1239 points
Cluster_3: 1238 points

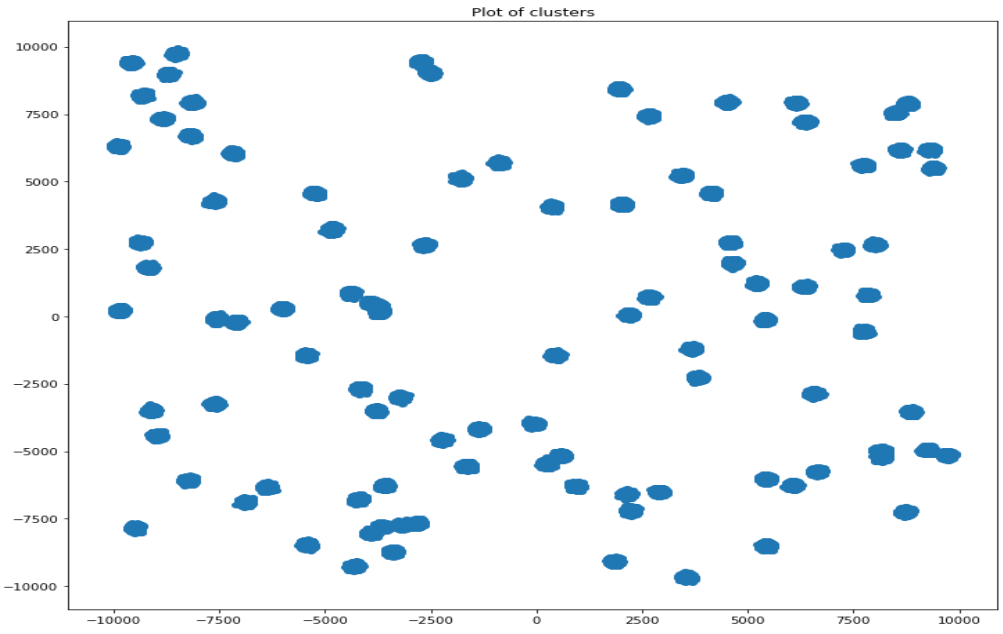
Input-100000 with eps: 35 and minPoints: 5



Total points: 100,000
Noise: 0 points
Total clusters: 9

Cluster_0: 10000 points
Cluster_1: 10000 points
Cluster_2: 10000 points
Cluster_3: 10000 points
Cluster_4: 10000 points
Cluster_5: 10000 points
Cluster_6: 10000 points
Cluster_7: 20000 points
Cluster_8: 10000 points

Input-1000000 with eps: 20 and minPoints: 5



Total points: 1,000,000
Noise: 1992 points
Total clusters: 99
