# Batch Incremental Neighbour Graph Density Based Scalable Clustering for Arbitrary Data

Samay Varshney
IIT Guwahati
180101097, BTech CSE
samay@iitg.ac.in

Pulkit Changoiwala
IIT Guwahati
180101093, BTech CSE
changoiw@iitg.ac.in

Komatireddy Sai Vikyath Reddy
IIT Guwahati
180101036, BTech CSE
komatire@iitg.ac.in

Sai Sumanth Madicherla
IIT Guwahati
180101068, BTech CSE
madicher@iitg.ac.in

## ABSTRACT

Incremental data mining algorithms process frequent updates to dynamic datasets efficiently by avoiding redundant computation. Due to the large size of the databases, it is highly desirable to perform these updates incrementally. This paper presents the first NG-DBSCAN incremental clustering algorithm named **BING-DBSCAN**, which is applicable to any database containing data from any metric space. It's approximate nature makes it fast in assigning clusters to new points. We show the effectiveness of our algorithm by performing experiments on large synthetic as well as real-world datasets. It also yields significant speed-up factors over static NG-DBSCAN for a large number of updates in batch mode.

## KEYWORDS

Density Based Clustering, Dynamic Datasets, Approximation

## 1 INTRODUCTION

Clustering algorithms are core to many fields, which involve grouping of the data based on the similarity between their objects. DBSCAN: Density-Based Scalable Clustering for Arbitrary Data [11] was the first algorithm that introduced the density-based clustering of the data points. Density-based clustering refers to grouping data packed in high-density regions of the feature space. DBSCAN offers two very distinctive features: 1) It labels outliers as noise points, i.e., it differentiates between "core points" (which are part of clusters) and "noise points" (which do not belong to any cluster). 2) It can classify clusters of arbitrary shapes as well. But DBSCAN has some limitations.

The problem with DBSCAN was that in high dimensional datasets, it partitions the feature space and then merges the spaces which lead to high computational complexity in large datasets. Also when applied to arbitrary distance measures, it requires retrieving each point's ε-neighborhood, for which the distance between all node pairs needs to be computed, resulting in $O(n^2)$ calls to the distance function. Hence it can't work with high dimensional, arbitrary data and large databases; these shortcomings of the DBSCAN are targeted with an arbitrary, approximated, scalable and distributed implementation namely NG-DBSCAN [7].

NG-DBSCAN: Neighbour Graph Density-Based Scalable Clustering For Arbitrary Data is a scalable, distributed, and approximated version of DBSCAN. Like other DBSCAN methods, it also follows *Map Reduce* method but it does not partition Euclidean space; rather, it is based on a vertex-centric approach, whereby we compute the neighbor graph; the neighbor graph is a data structure that stores the neighborhood of each database point. This vertex-centric approach allows NG-DBSCAN to solve the problem of arbitrary data and datasets with high dimensionality.

NG-DBSCAN outperforms DBSCAN implementation in terms of handling arbitrary databases, whereas approximating the nature of NG-DBSCAN introduces a small or negligible impact on the cluster quality. The main merits of NG-DBSCAN are efficiency and versatility. Efficiency: It often outperforms other distributed DBSCAN implementations with a small to negligible impact on results. Versatility: The vertex-centric approach allows us to represent item dissimilarity through any symmetric distance function, thus removing the need for Euclidean spaces to partition. In section 2 of our paper, we discuss the NG-DBSCAN algorithm, analyze the results and compare it with DBSCAN implementation to show that NG-DBSCAN outperforms DBSCAN with only a slight impact on the results.

NG-DBSCAN works on a static dataset, i.e., it can't work on a dataset that undergoes frequent changes where some points are added and some points are deleted. However, many real-world applications such as search engines like Google and recommender systems like one used in e-commerce websites are supposed to work with dynamic datasets. A naive way to get clusters of a dynamic dataset is via running a clustering algorithm each time the dataset is changed, but this method involves redundant computation. When datasets are large, or update frequency is high, this method becomes highly inefficient. Thus we have proposed an incremental version of NG-DBSCAN which prevents redundant computations and gives output identical to the non-incremental counterpart.

We have used the following observation to design the algorithm BING-DBSCAN (Batch Incremental NG-DBSCAN). Due to


*CS568: Data Mining, Jan - Apr, 2021, IIT Guwahati*

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

the density-based nature of the DBSCAN or NG-DBSCAN, the insertion or deletion of an object affects the current clustering only in the neighbourhood of this object and hence we can approximately calculate the epsilon neighbourhood of each newly added and deleted point.

The rest of the paper is organized as follows. In Section 2, we discussed about static NG-DBSCAN algorithm and it's results. In Section 3, we discuss various other incremental algorithms related to NG-DBSCAN. Section 4 introduces the BING-DBSCAN algorithm for incrementally updating a clustering and discussing its novelty. We demonstrate our algorithm's efficiency with an extensive performance evaluation on real and synthetic datasets in Section 6 and 7. Section 8 and 9 concludes with a summary and some directions for future work.

## 2 NG-DBSCAN ALGORITHM

NG-DBSCAN follows a vertex centric approach [10], in which computation is partitioned by and logically performed at the vertices of a graph, and vertices exchange messages whereby building a neighbour and $\epsilon$-graph and clusters are built based on the neighbour graph content. This approach enables distribution without needing Euclidean spaces to partition. In most of the existing DBSCAN algorithms [2], [5], [6] where d>=6 or n is high, it is computationally infeasible to run the algorithm either due to memory errors or large time complexity. But NG-DBSCAN, is independent of the dimensionality of the dataset and is able to run in the time linear to the number of data points. NG-DBSCAN also requires that the distance function used is *symmetric*: that is $d(x, y) = d(y, x)$ for all x and y.

Several vertex centric frameworks exists [4], [9]: these are distributed systems that iteratively execute a user-defined program over vertices of a graph, accepting input data from adjacent vertices and emitting output data that is communicated along outgoing edges. The existing implementation of NG-DBSCAN uses Apache Spark Framework which employs a shared nothing architecture geared toward synchronous execution; while our implementation involves C++ compiler execution with simple computers used in day to day life.

The NG-DBSCAN algorithm happens in two phases. Dataset is represented in the form of a graph where each node or vertex is a data point and edges represent the similarity distance measure between points. In the algorithm we are having ε, $Minpts$, $M_{\max}$, ρ, $T_{\mathrm{n}}$, $T_{\mathrm{r}}$, k as parameters and these are tuned in accordance with the number of data points and as per most efficient computational complexity. NG notation is used for denoting Neighbour Graph here.

### 2.1 Phase 1: Building the ε-graph

It is implemented through a neighbour graph which converges to k-nearest neighbour graph. It creates the ε-graph and avoids the ε-neighbourhood queries (which were creating high computational cost in DBSCAN).

The ε-neighborhood of a point p is the set of points within distance ε from p; ε-neighbourhood queries for a given vertex is finding nodes that are within the ε-distance which can take upto $O(n^2)$ if performed naively; ε-graph nodes are data points where each node's neighbors are a subset of its ε-neighborhood.

At each iteration, all pairs of nodes $(x, y)$ separated by 2 hops in the neighbor graph are considered and if their distance is less than ε, then this edge is added in ε-graph. To speed up the computation, nodes with at least $M_{\max}$ neighbors are removed from the neighbour graph.

**Algorithm 1:** Phase 1 – $\varepsilon$-graph construction.

```
εG ← new undirected, unweighted graph      // ε-graph
NG ← random neighbor graph initialization
for i ← 1 ... iter do
    // Add reverse edges
    for n ∈ active nodes in NG do in parallel
        for (n, u, w) ← NG.edges_from(n) do
            NG.add_edge(u, n, w)
    // Compute distances and update εG
    for n ← active nodes in NG do in parallel
        N ← at most ρk nodes from NG.neighbors(n)
        for u ← N do
            for v ← N \ {u} do
                w ← DISTANCE(u, v)
                NG.add_edge(u, v, w)
                if w ⩽ ε then εG.add_edge(u, v)
    // Shrink NG
    Δ ← 0          // number of removed nodes
    for n ← active nodes in NG do in parallel
        if |εG.neighbors(n)| ⩾ M_max then
            NG.remove_node(n)
            Δ ← Δ + 1
    // Termination condition
    if |NG.nodes| < T_n ∧ Δ < T_r then break
    // Keep the k closest neighbors in NG
    for n ∈ active nodes in NG do in parallel
        l ← NG.edges_from(n)
        remove from l the k edges with smallest weights
        for (n, u, w) ← l do
            NG.delete_edge(n, u, w)
return εG
```

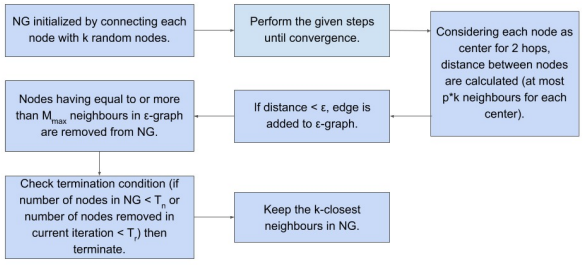


**Figure 1: Overview of Phase 1**

### 2.2 Phase 2: Discovering Dense Regions

Realizing the analogies between density-reachability and connected components, Phase 2 was inspired by Cracker [8], an efficient,

distributed method to find connected components in a graph. It takes ε-graph as input to build a clustering and neighbour lookups are performed instead of ε-neighbourhood queries. All nodes are given different roles in the ε-graph.
Core nodes are those having at least $MinPts - 1$ neighbours; Border nodes are those having at least 1 core node as neighbour while Noise nodes are the remaining nodes.

Each node coreness is then referred to as $(degree, nodeID)$. This labeling with degree is called *coreness dissemination* of the graph. Seed of a cluster is called a node with highest coreness. The Propagation forest is created having seed as root for each cluster and from that different clusters are created.

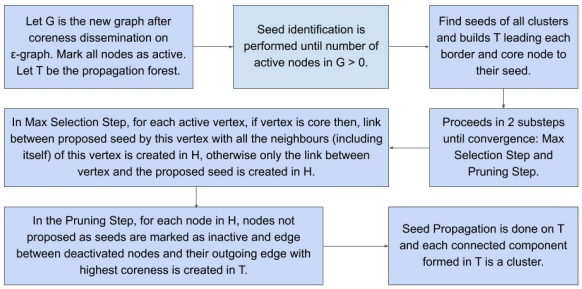


**Figure 2: Overview of Phase 2**

**Algorithm 2:** Phase 2 – Discovering dense regions.

$G = \texttt{Coreness\_Dissemination}(\varepsilon G)$
**for** $n \leftarrow$ *nodes in* $G$ **do in parallel**
  $n$.Active $\leftarrow True$
$T \leftarrow$ empty graph  // Propagation forest
// Seed Identification
**while** $|G.nodes| > 0$ **do**
  // Max Selection Step
  $H \leftarrow$ empty graph
  **for** $n \leftarrow G.nodes$ **do in parallel**
    $n_{max} \leftarrow \texttt{maxCoreNode}(G.\text{neighbors}(n) \cup \{n\})$
    **if** $n$ *is not-core* **then**
      $H$.add_edge$(n, n_{max})$
      $H$.add_edge$(n_{max}, n_{max})$
    **else**
      **for** $v \leftarrow G.neighbors(n) \cup \{n\}$ **do**
        $H$.add_edge$(v, n_{max})$
  // Pruning Step
  $G \leftarrow$ empty graph
  **for** $n \leftarrow H.nodes$ **do in parallel**
    $n_{max} \leftarrow \texttt{maxCoreNode}(H.neighbors(n))$
    **if** $n$ *is not-core* **then**
      $n$.Active $\leftarrow$ False
      $T$.add_edge$(n_{max}, n)$
    **else**
      **if** $|H.neighbors(n)| > 1$ **then**
        **for** $v \leftarrow H.neighbors(n) \setminus \{n_{max}\}$ **do**
          $G$.add_edge$(v, n_{max})$
          $G$.add_edge$(n_{max}, v)$
      **if** $n \notin H.neighbors(n)$ **then**
        $n$.Active $\leftarrow$ False
        $T$.add_edge$(n_{max}, n)$
      **if** IsSeed $(n)$ **then**
        $n$.Active $\leftarrow$ False
**return** Seed_Propagation$(PropagationTree)$

## 2.3 Evaluation Metrics

The evaluation of the algorithm was carried out by running it on both synthetically generated and externally used datasets and comparing the results and metrics with DBSCAN. Quality metrics such as **compactness, separation, recall, speed-up** are used to compare.

Compactness measures how closely are items in each cluster; Separation measures how well items in different clusters are separated; Recall of a cluster is the fraction of the node pairs that are in the same cluster with that in a reference cluster; Speed-Up measures the algorithm runtime improvement when increasing the number of cores dedicated to the computation.

Since computing the above metrics is computationally hard, data points are picked randomly with uniform sampling and averaging independent runs for each data point.
**High compactness, less separation, less time taken** was observed in most of the NG-DBSCAN cases in comparison with other DBSCAN algorithms irrespective of the data which proves that clusters are more separated, dense and efficiently computable in case of NG-DBSCAN.

*2.3.1 Text Data Compatibility.* The NG-DBSCAN algorithm also supports text data compatibility. For this we used *Jaro-Winkler's Distance* algorithm as a distance measure between 2 text points.

Using Jaro-Winkler's Distance between two strings, epsilon graph was generated, further leading to generation of propagation tree. Then clusters were generated for the data set in similar fashion as with Euclidean distance measure.

Using dataset_generator.py user is asked to enter the type of data to generate. As per the evaluation measure, we used the fact; in the comparison of 2 strings, the lower the Jaro–Winkler distance for two strings, the more similar the strings are.

## 2.4 Working on Toy Dataset

15 two-dimensional points were taken as an input for the toy dataset. These points are present in file named toy_dataset.txt.

```
non_text 15 2
-11.004161206558486 -10.24530372973464
3.1768184552599394 -9.006254703405759
-8.357174823998038 -8.854391074446012
-8.435048195710767 -7.806762293952905
-7.393981404723236 1.028497940990529
-6.465388143821774 0.4870595871129323
1.7378505231503554 -9.575283879631659
1.1437903454896514 -9.697724553571327
-6.9145102974713275 3.033882068724556
-6.570426782705661 1.1091458038063127
2.6502349385029955 -8.78415509686841
-9.008866366830741 -8.939591350340011
-8.198548337708637 -9.689507438131614
2.294976018616592 -10.504253545393299
-8.244677296600988 3.2921021434741
```

**Figure 3: Toy Dataset Points**

The points of toy dataset are listed below. First line represents that it is non-textual data, has 15 points and is of 2 dimension. We took 2-dimensional for our simplicity in explaining.
The 1st column represents x-coordinate while 2nd column represents y-coordinate of the points. ith row represents the ith point of dataset.

The red edges between (i,j) indicates the edge between the ith and jth point in the epsilon graph. Each point in the epsilon graph is connected to atleast $M_{\max}$ more points in their neighbour (as seen in the figure) which are within $\epsilon$ distance here.

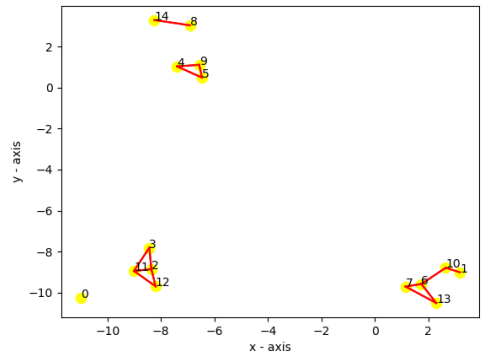


**Figure 4: Toy Dataset Epsilon Graph**

The propagation tree is having 2nd, 4th, 6th points as seeds of different clusters and hence 3 clusters are formed.
We found that the points 0th, 8th and 14th are noise since no core node is present in their neighbour within $\epsilon$ distance, while 1st point is border since only one core node (10th) is present in it's neighbour, while others are core points.

In the main algorithm for NG-DBSCAN, the following values of parameters were used for the toy dataset.

```
Want to change the default parameters?
Enter 1 for Yes and 0 for No
1
Enter Parameters(If you want to keep default value then enter -1
Enter x for Tn(Tn = x*n)
-1
Enter x for Tr(Tr = x*n)
-1
Enter k
2
Enter Mmax
3
Enter p
2
Enter iter
10
Enter epsilon
1.5
Enter Minpts
3
```

**Figure 5: Toy Dataset Parameters**

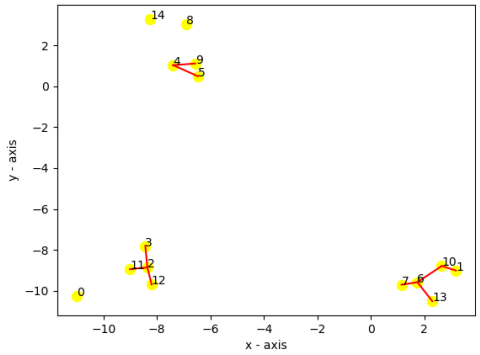


**Figure 6: Toy Dataset Propagation Tree**

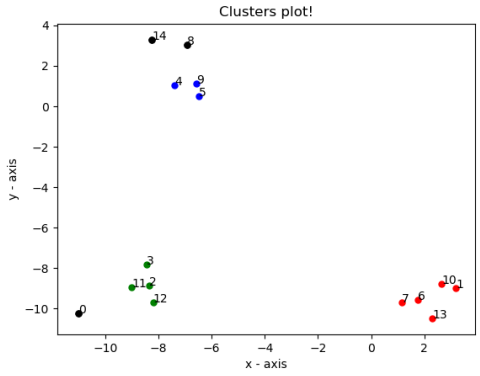


**Figure 7: Toy Dataset Clusters Identification**

In the toy dataset clusters identification, black points refer to the noise points while green points corresponds to 1st cluster and blue corresponds to 2nd cluster and red corresponds to 3rd cluster.

```
Compactness of each cluster:
 0.997319 0.997787 0.99219

Separation between (i,j)th clusters:
 0.997989 0.597068 0.540703
 0.597068 0.998525 -0.346914
 0.540703 -0.346914 0.993752
```

**Figure 8: Toy Dataset Metrics Evaluation**

## 2.5 Results

The results for the dimensional datasets used are shown in Figure 9,10,11.

*2.5.1 Blob Dataset.* The plot in Figure 9 contains 3 blobs in 3-dimension having 1000 points each. The following parameters were used: $xTn = 0.001$, $xTr = 0.0001$, $k = 20$, $M_{\max} = 20$, $\rho = 3$, $iter = 12$, $\epsilon = 1.0$, $Minpts = 8$.

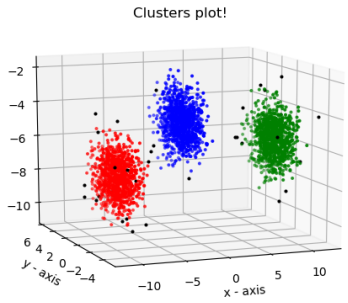


**Figure 9: 3-dimensional Blob Dataset**

```
Compactness of each cluster:
 0.988556 0.988927 0.919702

Separation between (i,j)th clusters:
 0.988567 0.134486 -0.195762
 0.134486 0.988938 -0.94748
 -0.195762 -0.94748 0.919784
```

**Figure 10: 3-dimensional Blob Dataset Evaluation**

*2.5.2 Moon Dataset.* The plot in Figure 10 contains 2 moons containing 2500 points each. The following parameters were used: $xTn = 0.001$, $xTr = 0.0001$, $k = 20$, $M_{\max} = 20$, $\rho = 2$, $iter = 10$, $\epsilon = 0.5$, $Minpts = 10$.

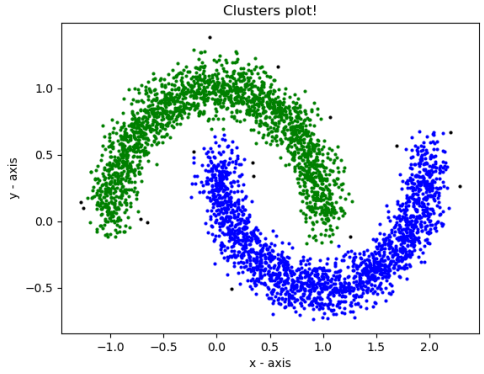


**Figure 11: 2-dimensional Moon Dataset**

*2.5.3 Circular Dataset.* The plot in Figure 11 contains 2 circles containing 2000 points each. The following parameters were used: $xTn = 0.001$, $xTr = 0.0001$, $k = 20$, $M_{\max} = 15$, $\rho = 2$, $iter = 15$, $\epsilon = 0.05$, $Minpts = 10$.

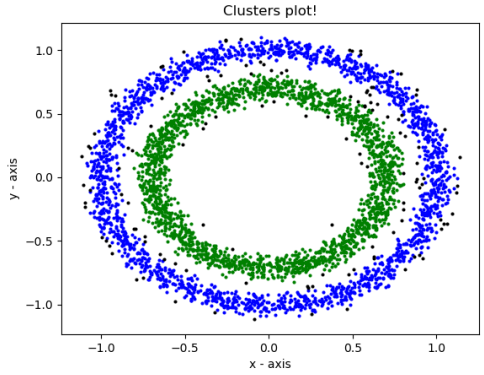


**Figure 12: 2-dimensional Moon Dataset**

## 3 RELATED INCREMENTAL ALGORITHMS

Some of the density based incremental algorithms which in some sense are related to NG-DBSCAN and DBSCAN referred by this paper are as follows:

(1) **MR-IDBSCAN: Efficient Parallel Incremental DBSCAN Algorithm using MapReduce: [12]** It is also a scalable, density based algorithm to find clusters of arbitrary shapes, size, and as well as filter out noise like NG-DBSCAN does. It also uses the Map Reduce method which NG-DBSCAN follows.
- **Intuition of proposed solution:** In this algorithm, new data points which intersect with old data points are determined. For each intersection point, the new dataset uses an incremental DBSCAN algorithm to determine new cluster membership. Cluster memberships of the remaining points are then updated. R*- tree data structure is used in this algorithm.

- **Results:** Time complexity of this algorithm is less than the original DBSCAN algorithm and it has also dealt with fault tolerance which makes it to compete with NG-DBSCAN.
- **Limitations:** In this algorithm it is difficult to delete clusters incrementally from an existing set of clusters.

(2) **Incremental Clustering for Mining in a Data Warehousing Environment: [3]** It is also a density based algorithm used in data warehouses applicable to any database containing data from a metric spacee, e.g., to a spatial database or to a WWW-log database.
- **Intuition of proposed solution:** In this algorithm, each data point to add or delete is considered seperately. $\text{UpdSeed}_{\text{ins}}$ and $\text{UpdSeed}_{\text{del}}$ are created which contains the affected nodes due to the incoming query of insertion or deletion.
- **Results:** Time complexity of this algorithm is less as compared to static DBSCAN as it only considers the points which are affected and due to density based nature of DBSCAN, only few points are affected.
- **Limitations:** In this algorithm, computational cost is high since queries are processed one at a time.

(3) **$\text{BISDB}_{\text{x}}$: Batch Incremental Clustering for Dynamic Datasets using SNN-DBSCAN: [1]** $\text{BISDB}_{\text{x}}$ is a batch incremental algorithm based on graph based clustering involving frequently changing dynamic datasets.
- **Intuition of proposed solution:** Incremental version of SNNDB, a graph-based clustering technique is modified to $\text{BISDB}_{\text{x}}$ since it was making the process extremely slow when updates are made on larger base dataset. $\text{BISDB}_{\text{add}}$ is used while adding points while $\text{BISDB}_{\text{add}}$ is used while deleting points dynamically. It computes k-nearest neighbour graph (like in the case of NG-DBSCAN), shared nearest neighbours graph, core and non-core points incrementally.
- **Results:** Experimental observations on real world and synthetic datasets showed that $\text{BISDB}_{\text{x}}$ are up to 4 orders of magnitude faster than the naive SNNDB algorithm and about 2 orders of magnitude faster than the pointwise incremental method.

(4) **A New Incremental Semi-Supervised Graph Based Clustering: [13]** It introduces a new incremental semi-supervised clustering which is based on a graph of k-nearest neighbour using seeds, namely incremental SSGC. In each incremental clustering algorithm, two processes including insertion and deletion for new data points are used for updating the current clusters.
- **Intuition of proposed solution:** Given a k-nearest neighbor graph presenting a data set X, this step uses a loop in which at each step, all edges which have weight less than a threshold $\theta$ will be removed. The value of $\theta$ is initialized by $\theta$ at first step and incremented by 1 after each step. This loop will stop when each connected component has at most one kind of seeds. The main clusters are identified by propagating label in each connected component that contains seeds. The further steps isolate the outliers.
- **Results:** Incremental SSGC obtains the good results compared with the incremental DBSCAN. It can be explained by the fact that the incremental DBSCAN can not detect clusters with different densities while incremental SSGC does and hence is a competitor for NG-DBSCAN.

# 4 PROPOSED ALGORITHM

Given a dataset D, initial clustering C and some sequence of changes (n queries of addition and deletion of points), our BING-DBSCAN will update the dataset D to D', clustering to C' that will be identical to the clustering created by the static NG-DBSCAN. Our incremental version BING-DBSCAN partitions the queries into set of points to be inserted ($\text{n}_{\text{add}}$) and deleted ($\text{n}_{\text{del}}$). First it performs all the additions in the existing dataset D and then updates the clustering and $\epsilon$-graph. Then it performs all the deletions and updates the clustering to C'. It requires 6 parameters: $k$, $M_{\max}$, $Minpts$, $\epsilon$, $iter$, *threshold*. The last parameter *threshold* is different as it was not present in the static version. Let us define a point to be *affected* if it changes it's type from core to non-core or from non-core to core. For each affected point due to addition and deletions, BING-DBSCAN computes it's approximate $\epsilon$ neighbourhood for finding it's new cluster (if it exists) and assigns it to that cluster.

## 4.1 Insertion Phase

Insertion phase of BING-DBSCAN algorithm converts D to $\text{D'}_{\text{add}}$ by adding all points in $\text{n}_{\text{add}}$. This phase is divided into 2 parts:

*4.1.1 Node Identification.* Each point present in D will be classified as core or non-core. For each point 'u' in $\text{n}_{\text{add}}$, it's approximate $\epsilon$-neighbourhood is calculated. From each cluster present in C, k points are assigned to the list of 'u' and this list is updated using the neighbours of these points and searching for the nearest neighbours for u is done till *iter* number of iterations only.

**Algorithm 1:** Random Neighbour Search Algorithm

**Result:** $\epsilon$-neighbourhood of the given set of points is updated

Let S be the set of points for which we have to find the $\epsilon$-neighbourhood over the set A;
**if** *A is Old Cluster Data* **then**
  Assign each point u of S with k random points from each cluster C (in old cluster);
**else**
  Assign each point u of S with k random points in A;
**end**
Store these random points in T;
For each vertex u in S: **while** *i < iter* **do**
  For each vertex v in T, check distance between u and v;
  **if** *dis(u,v) <= $\epsilon$* **then**
    Create an edge between u and v in $\epsilon$ Neighbourhood Graph;
    Insert neighbours of v in T;
  **else**
    Check for another vertex in T;
  **end**
  i++;
**end**

**Algorithm 2:** Node Identification in Insertion Phase

**Result:** The updated dataset D'$_{add}$ after addition of points will become labeled with core and non-core points

Let all the points to add into the current dataset form a set 'A'. Let all the points having degree less than *Minpts* form a set 'S';
**if** *A.size * S.size < threshold* **then**
  For each point in 'S', we will calculate distance with each point in 'A';
**else**
  Run the random_neighbour_search algorithm for all the points in 'S' over the set 'A';
**end**
To identify the type (core or non-core) of newly added points, store them in ***I***;
Run random_neighbour_search algorithm to find the approximate $\epsilon$-neighbourhood of the newly added points to label them;

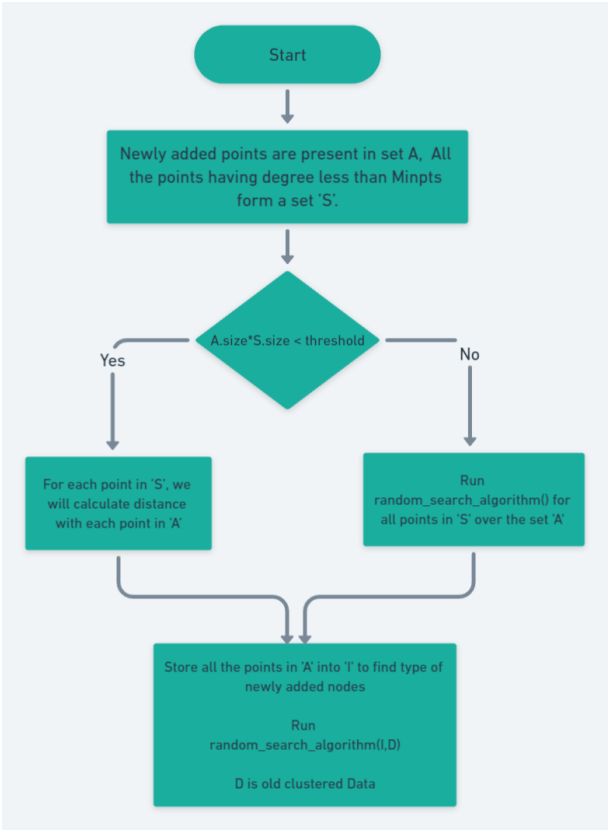


**Figure 13: Node Identification in Insertion Phase**

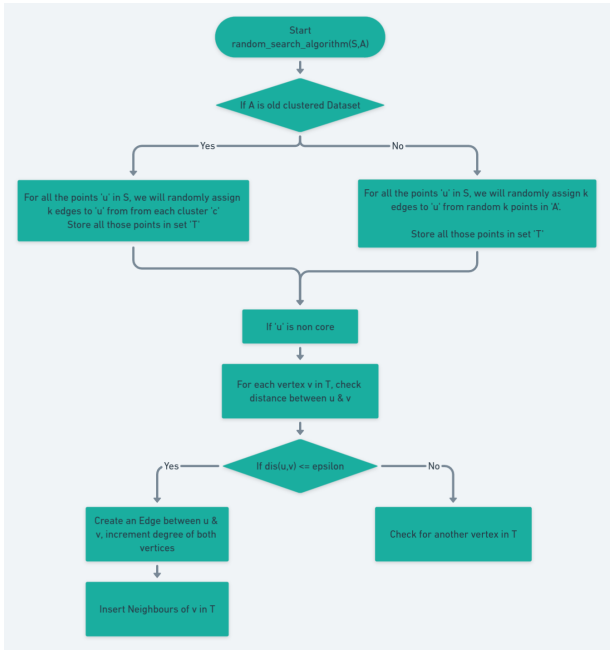


**Figure 14: Random Neighbour Search Algorithm**

*4.1.2 Cluster Membership.* Each node will get it's true label (core or non-core) in the updated dataset D'$_{add}$ and clusters are formed.

**Algorithm 3:** Cluster Membership in Insertion Phase

**Result:** The updated dataset D'$_{add}$ after addition of points will be clustered.

1 From the identification of nodes section, we can identify which nodes have changed their identity from non-core to core. So, store these points in a set 'Upd$_{ins}$';
2 After building this set, Depth First Search will run from all points in 'Upd$_{ins}$' and give each point the cluster number to which it belongs;
3 In each DFS, unique and same cluster name is given to all points which are reachable from that point;
4 Each point is visited once only in this process in the updated dataset D'$_{add}$;

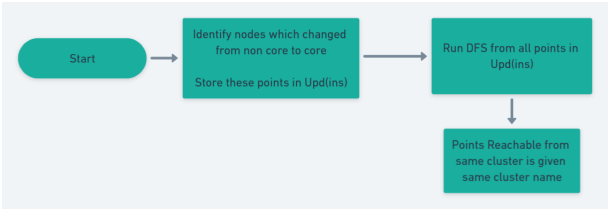


**Figure 15: Cluster Membership in Insertion Phase**

## 4.2 Deletion Phase

During the deletion phase, points listed in n$_{del}$ are removed from D'$_{add}$ to get the final dataset D'. This phase is divided into 2 parts:

*4.2.1 Node Identification.* $\epsilon$-neighbourhood of the nodes present in the neighbourhood of points which are present in n$_{del}$ is updated and hence will be classified as core or non-core.

**Algorithm 4:** Deletion Phase Node Identification

**Result:** The updated dataset D' after deletion of points from D'$_{add}$ will become labeled with core and non-core points

1 Let all the points to delete from the current dataset, form a set 'Del'. Let all the points having degree more than or equal to *Minpts* form a set 'R';
2 For all the points in 'R', calculate which all points will lose their degree to $< Minpts$ after deleting the points present in 'Del'. This can be done by going over the neighbour list of each point in 'R';
3 Store all these points which will have their final degree $< Minpts$ in 'I';
4 Store the neighbours for all the points present in 'Del' inside 'I'. Run random_neighbour_search algorithm for points in I over dataset D'$_{add}$ so that we could find whether they can become core or not again;

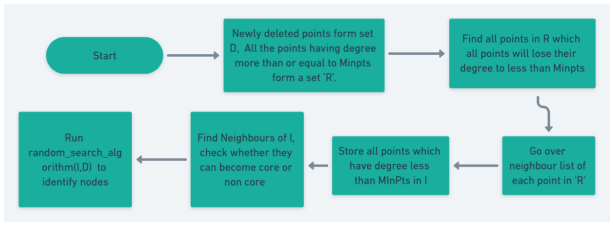


**Figure 16: Node Identification in Deletion Phase**

*4.2.2 Cluster Membership.* Each node will get it's true label (core or non-core) in the final dataset D' and final set of clusters are formed.

**Algorithm 5:** Cluster Membership in Deletion Phase

**Result:** The updated dataset D' after deletion of points from D'$_{add}$ will be clustered

1 From the identification of nodes section, we can identify which nodes have changed their identity from core to non-core. So, store these points in a set 'Upd$_{del}$';
2 After building this set, run Depth First Search from all points in 'Upd$_{del}$' and give each point the cluster number to which it belongs;
3 In each DFS, unique and same cluster name is given to all points which are reachable from that point;
4 Each point is visited once only in this process in the updated dataset D';

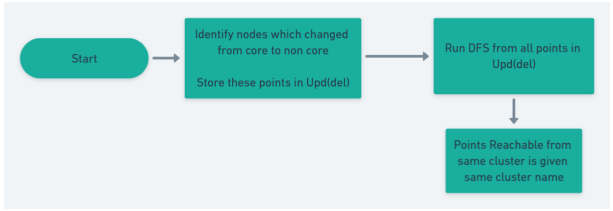


**Figure 17: Cluster Membership in Deletion Phase**

## 4.3 Complexity of Algorithm

- **Node Identification:** Time complexity will be linear to the number of points in the updated dataset.
  - **Insertion:** First step of insertion will take $O(n * a)$ where 'n' is the number of non-core points in the dataset D before update and 'a' is the number of points to be added in D. Second step of addition will take $O(a * c)$ where 'c' is the constant number of iterations for which we will search for the neighbours for each point in I.
  - **Deletion:** First step of deletion will take $O(m)$ where 'm' is the number of core points in the dataset D before update. Second step of deletion will take $O(m * c)$ where 'c' is the constant number of iterations for which we will search for the neighbours for each point in I.

- **Cluster Membership:** Both insertion and deletion steps will take $O(N * Minpts)$ time in worst case where 'N' is the number of points in the updated dataset.

## 4.4 Correctness of Algorithm

- **Node Identification:**
  - **Insertion:** For all the points present in dataset D, we will get it's approximate type whether it is core or non-core. Also for all the new added points, we found it's neighbours in each cluster hence we also get an approximation of the cluster to which it will belong along with it's type. Hence each node will have it's approximate type.
  - **Deletion:** All the points in I will be those which can change from core to non-core. But some of them can still become core by having some connections in the dataset and we found those nodes through random_neighbour_search. Hence here also each node will have it's approximate type in the updated dataset.
- **Cluster Membership:** Since we are running Depth First Search (DFS) from all those points which got their type changed, all the reachable nodes from these affected nodes are visited and their cluster numbers are updated. All the other remaining points will have their previous cluster numbers since addition and deletion of points didn't affected them.

# 5 DATASETS

Here we describe the datasets used in our experiments. We consider the following datasets:

## 5.1 Real Datasets

Here are the specifications of the datasets which we found.

- **Traffic Dataset:** We collected traffic data, It is a 2-dimensional dataset related to Traffic Signals. We found it here. Traffic volumes are collected by manual and mechanical means and are hosted at an independent site. They represent typical volumes likely to be found at a location.This data set contains information of traffic vehicles on the road. Our aim was to discover traffic hotspots in the city using our clustering algorithm and traffic volume data set. We pruned the main data to extract the coordinates of the location and removed duplicate coordinates. We extracted some data points from the same data set for the incremental queries. After pruning down the dataset and removing the duplicates, the dataset contains 2000 points in the existing dataset and 450 points as updates used for addition and deletions.

## 5.2 Synthetic Datasets

We also used synthetically generated input using the Sci-Kit library. We generated different types of input using circle, moon and blobs shapes. These graphs are usually considered as a baseline for testing clustering algorithms in a d-dimensional space and we also generated random shapes in 2D and 3D. We removed few points in a cluster so that it becomes two clusters and added few points between two clusters so they become one cluster and used these points as incremental queries. Here are the specifications of the used datasets.

- **Dataset_1:** It is a 2-dimensional dataset merging one blob and one circular cluster and removing points from a blob. It has 1500 points in the existing dataset and around 225 points for addition and deletions.
- **Dataset_2:** It is a 2-dimensional dataset covering the cases of merging of 2 clusters and deletion of points from an existing cluster. It has 2000 points in the existing dataset and around 550 points for addition and deletions.
- **Dataset_3:** It is a 3-dimensional dataset covering all the cases: Merging of 2 clusters, creation of a new cluster, deletion of points from a cluster and division of 2 clusters. It has 2800 points in the existing dataset and 2600 points for addition and deletion.

# 6 EXPERIMENTAL SETUP

We evaluate BING-DBSCAN through a comprehensive set of experiments, evaluating their clustering quality on real and synthetic datasets and comparing it to alternative approaches. In the following, we provide details about our setup.

## 6.1 Experimental Platform

All the experiments have been conducted on a cluster running Ubuntu Linux equipped with 8 GB of RAM, a 5-core CPU and a 1 Gigabit interconnect. Both the implementation of our approach and the alternative algorithms we use for our comparative analysis uses these specifications.

## 6.2 Evaluation Metrics

We have evaluated our algorithm some standard metrics like time, memory used, CPU usage, and clustering specific metrics like compactness and separation. Along with these metrics, we will also do manual investigations of the clusters using their plots(in 2-dimensional and 3-dimensional datasets).

- **Compactness:** It measures similarity between items in clusters. Compactness is obtained by calculating the average pairwise similarity among items in each cluster. We have taken cosine similarity for pairwise similarity. Higher values of compactness are preferred.
- **Separation:** It is a measure of how well clusters are separated. Separation is obtained by calculating the average pairwise similarity between items in different clusters. Lower values are preferred.
- **Time:** Time taken is a measure of how time-consuming our algorithm is. Algorithms that run in a shorter time are preferred.
- **Memory Used:** It measures how much memory an algorithm uses to execute its tasks. As storage resources are expensive, we prefer algorithms with lower memory usage.
- **CPU Usage:** It measures what percentage of the processing power of the system is used for the algorithm. Like other resources, we are greedy with processing power as well; thus, lower values are preferred.

### 6.3 Evaluation Strategy

Let say we have data set $D$ and some updates $U$ are processed on the data set $D$, after these updates data set changes to new data set, say it is $D'$. **Strategy:** We run the static NG-DBSCAN on the old datset $D$, we obtain clusters, say $C$, then we update the dataset to $D'$. We run static again on dataset $D'$. We calculate values of all metrics. Now, we run the incremental algorithm using old dataset $D$, old clustering $C$ and update queries $U$, we obtain new clusterings and calculates values of the all evaluation metrics.

## 7 EXPERIMENTAL RESULTS

We evaluate BING-DBSCAN by comparing its results with those obtained through the exact NG-DBSCAN algorithm and by comparing them using different metrics on datasets of points in Euclidean spaces.

In the following, we used Euclidean distance and k = 15, $M_{max}$ = k to 2k: in further Section 7.2, we provide an in-depth evaluation of those parameters and see that they provide a good trade-off between clustering quality and run-time. The parameters for both BING-DBSCAN and DBSCAN for each dataset are present in the Code Link.

### 7.1 Performance with Datasets

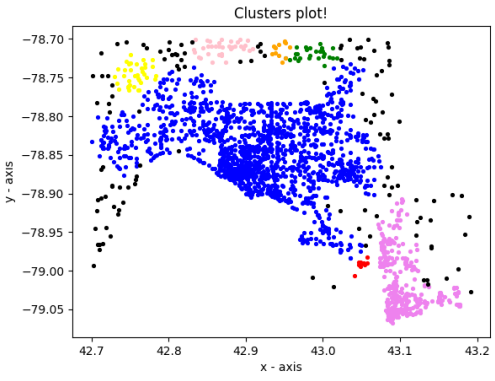


**Figure 18: Initial Traffic Dataset**

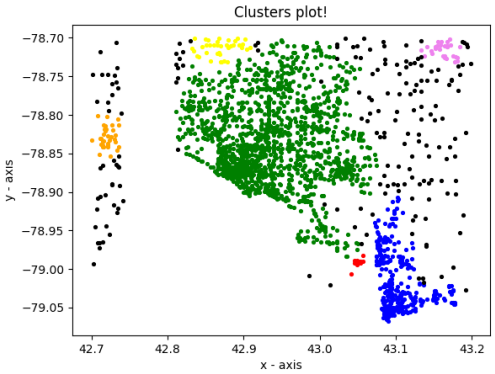


**Figure 19: Traffic Dataset after applying NG-DBSCAN**

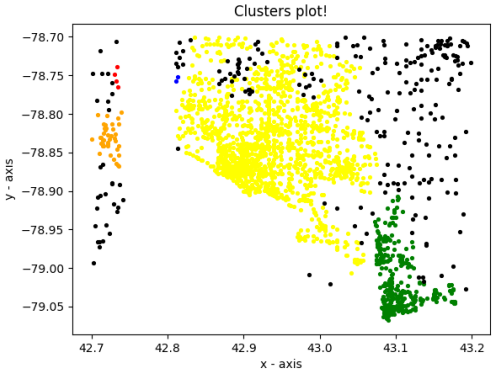


**Figure 20: Traffic Dataset after applying BING-DBSCAN**

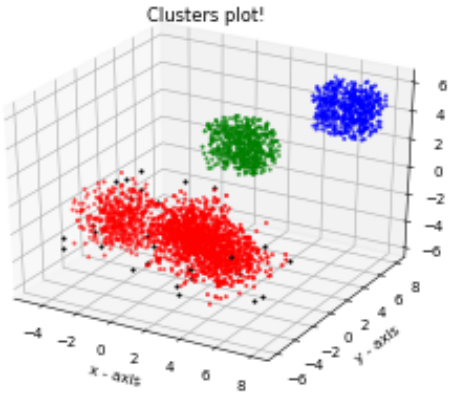


**Figure 21: Initial Dataset_3**

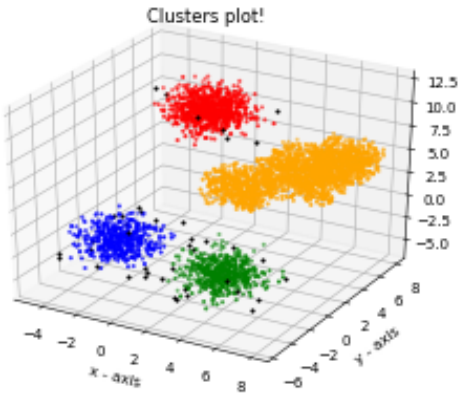


**Figure 22: Dataset_3 after applying NG-DBSCAN**

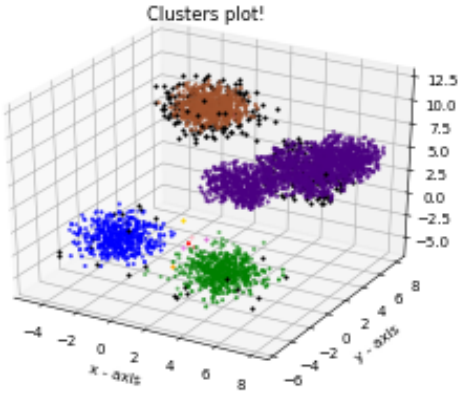


**Figure 23: Dataset_3 after applying BING-DBSCAN**

*7.1.1 Compactness.* The mean should be close to 1 while the standard deviation (S.D.) should be close to 0 for better accuracy.

*7.1.2 Seperation.* The mean should be close to -1 while the standard deviation (S.D.) should be close to 0 for better accuracy.

*7.1.3 Time.* Time taken in BING-DBSCAN is far lesser as compared to NG-DBSCAN in context to the accuracy achieved currently. There is a trade off between time and accuracy controlled by the parameters. On improving accuracy, time taken by incremental version increases.

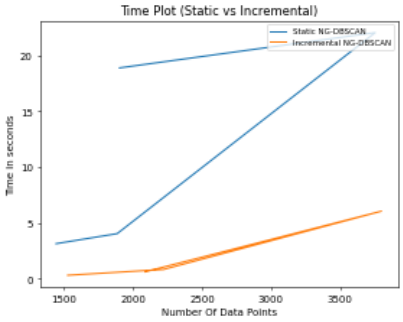


**Figure 24: Time Comparison**

*7.1.4 Memory.* Memory taken in BING-DBSCAN is more as compared to NG-DBSCAN due to maintenance of different data structures like set of core nodes, non-core nodes, different kinds of mappings.

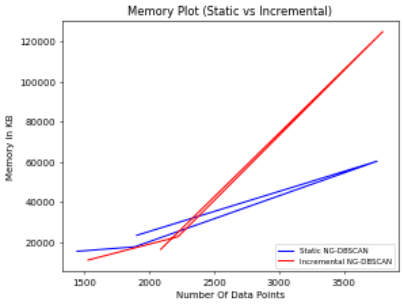


**Figure 25: Memory Comparison**

*7.1.5 CPU Usage.* It is less in BING-DBSCAN as compared to NG-DBSCAN due to less complexity in the former case. It is different for each dataset due to different activites occurring at same time on a CPU.

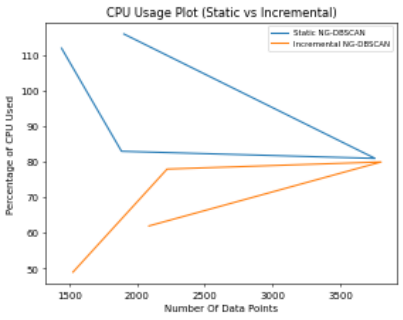


**Figure 26: CPU Usage Comparison**

## 7.2 Parameter Space

BING-DBSCAN has the following parameters: i) k, the number of neighbors per node in the neighbor graph; ii) $M_{max}$, the threshold of

**Table 1: Compactness Measure**

| Compactness Measure | | | | | |
|---|---|---|---|---|---|
| Algorithm | Value | Dataset_1 | Dataset_2 | Dataset_3 | Traffic Dataset |
| NG-DBSCAN | Mean | 0.802748 | 0.981389 | 0.929485 | 1.00000 |
| | S.D. | 0.116906 | 0.000109194 | 0.00244034 | 1.71786e-14 |
| BING-DBSCAN | Mean | 0.894894 | 0.992028 | 0.974308 | 1.00000 |
| | S.D. | 0.0305777 | 0.000135987 | 0.00214696 | 1.62239e-14 |

**Table 2: Seperation Measure**

| Seperation Measure | | | | | |
|---|---|---|---|---|---|
| Algorithm | Value | Dataset_1 | Dataset_2 | Dataset_3 | Traffic Dataset |
| NG-DBSCAN | Mean | -0.107518 | 0.71244 | -0.251028 | 0.999997 |
| | S.D. | 0.274237 | 0.0000 | 0.292897 | 8.802e-12 |
| BING-DBSCAN | Mean | 0.0127214 | 0.899173 | 0.264705 | 0.999997 |
| | S.D. | 0.466167 | 0.0135617 | 0.334339 | 1.07101e-11 |

**Table 3: Frequently Asked Questions**

| | |
|---|---|
| Link to base paper | [7] |
| Link to the code of the base paper | Code Link |
| Link to the datasets used | Datasets Link |
| Link to your code | Our Code Link |
| Is your code working | Yes |
| Maximum speed up you are getting | It depends upon the dataset used, but atleast 2-3 times speed-up we are getting as compared to static algorithm. |
| Extra amount of memory required by your incremental algorithm | For small datasets it is almost similar but for larger datasets the constant factor keeps on increasing. For n=5000, ING-DBSCAN takes twice memory as that taken by static version. |
| Accuracy of the incremental algorithm as compared to the static algorithm | It is slightly less accurate as compared to static version but can be improved more on tuning the parameters. |
| How did you measure the accuracy? | We measured the accuracy with the help of plots and metrics like seperation, compactness. |
| Are you interested in further improve your work during the summer? | We will try to generalize our algorithm on the parameters choosing and will try to work upon the textual datasets for incremental version. |

neighbors in the $\epsilon$-graph to remove nodes from the neighbor graph; iii) Minpts, a limit which specify whether a node is core or non-core; iv) iter, a constant which specifies upto what extent we will search for neighbours for each node; v) $\epsilon$, maximum distance upto which one node can be a neighbour of other node; vi) threshold, used to balance the trade-off between time and accuracy.

## 8 LESSONS LEARNT

The major learning was about managing how to design and implement such a clustering algorithm within the limited time frame. The other learning was how to build such synthetic datasets covering all the cases, tuning the parameters for better accuracy and how to divide work between different team members working remotely.

## 9 FUTURE WORK

We presented BING-DBSCAN algorithm that handles both addition and deletion to the dataset. We will further try to improve it's accuracy (by improving the seperation metric) and will try to find a proper way to tune the parameters for some representative datasets. Evaluating it on bigger textual datasets will be one of our major future works.

## REFERENCES

[1] P. Bhattacharjee and P. Mitra. 2019. BISDBx: towards batch-incremental clustering for dynamic datasets using SNN-DBSCAN. *Pattern Analysis and Applications* 23 (2019), 975–1009.

[2] Bi-Ru Dai and I-Chang Lin. 2012. Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition. *2012 IEEE Fifth International Conference on Cloud Computing* (2012), 59–66.

[3] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Michael Wimmer, and Xiaowei Xu. 1998. Incremental Clustering for Mining in a Data Warehousing Environment. In *Proceedings of the 24rd International Conference on Very Large Data Bases (VLDB '98)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 323–333.

[4] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 599–613.

[5] Yaobin He, Haoyu Tan, Wuman Luo, Huajian Mao, D. Ma, S. Feng, and Jianping Fan. 2011. MR-DBSCAN: An Efficient Parallel Density-Based Clustering Algorithm Using MapReduce. *2011 IEEE 17th International Conference on Parallel and Distributed Systems* (2011), 473–480.

[6] Mariam Khader and Ghazi Al-Naymat. 2020. Density-Based Algorithms for Big Data Clustering Using MapReduce Framework: A Comprehensive Study. *ACM Comput. Surv.* 53, 5, Article 93 (Sept. 2020), 38 pages. https://doi.org/10.1145/3403951

[7] Alessandro Lulli, Matteo Dell'Amico, Pietro Michiardi, and Laura Ricci. 2016. NG-DBSCAN: Scalable Density-Based Clustering for Arbitrary Data. *Proc. VLDB Endow.* 10, 3 (Nov. 2016), 157–168. https://doi.org/10.14778/3021924.3021932

[8] A. Lulli, L. Ricci, E. Carlini, P. Dazzi, and C. Lucchese. 2015. Cracker: Crumbling large graphs into connected components. In *2015 IEEE Symposium on Computers and Communication (ISCC)*. 574–581. https://doi.org/10.1109/ISCC.2015.7405576

[9] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[10] Robert Ryan McCune, Tim Weninger, and Greg Madey. 2015. Thinking Like a Vertex: A Survey of Vertex-Centric Frameworks for Large-Scale Distributed Graph Processing. *ACM Comput. Surv.* 48, 2, Article 25 (Oct. 2015), 39 pages. https://doi.org/10.1145/2818185

[11] L. Meng'Ao, M. Dongxue, G. Songyuan, and L. Shufen. 2015. Research and Improvement of DBSCAN Cluster Algorithm. In *2015 7th International Conference on Information Technology in Medicine and Education (ITME)*. 537–540. https://doi.org/10.1109/ITME.2015.100

[12] Maitry Noticewala and Dinesh Vaghela. 2014. MR-IDBSCAN: Efficient Parallel Incremental DBSCAN algorithm using MapReduce. *International Journal of Computer Applications* 93 (05 2014), 13–18. https://doi.org/10.5120/16202-5391

[13] V. V. Thang and F. F. Pashchenko. 2018. A New Incremental Semi-Supervised Graph Based Clustering. In *2018 Engineering and Telecommunication (EnT-MIPT)*. 210–214. https://doi.org/10.1109/EnT-MIPT.2018.00054