INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI

Density-Based Mining Algorithms for Dynamic Data: An Incremental Approach

by

Panthadeep Bhattacharjee
Roll no.- 136101011

A Thesis submitted in partial fulfillment of the requirement for the degree of Doctor of Philosophy

in the

Department of Computer Science and Engineering

Under the supervision of
Dr. Pinaki Mitra

Dedicated to

Almighty,

Smt. Gouri Bhattacharjee(my Mother),

Shri Janmejoy Bhattacharjee(my Father),

&

My Teachers, Friends and Well wishers.

Declaration of Authorship

I, Panthadeep Bhattacharjee, hereby confirm that:

- The contents of work contained in this thesis are original and have been done by myself under the general supervision of my thesis supervisor.
- This work has not been submitted to any other Institute for the award of any degree or diploma.
- I have followed the necessary guidelines provided by the Institute while preparing this thesis.
- Whenever I have used materials (data, theoretical analysis, results) from other sources, I have given due credit to the authors/researchers by citing them in the text of the thesis and giving their details in the Reference/Bibliography section. Whenever I have quoted from the work of others, the source is always given.

....

Panthadeep Bhattacharjee
Research Scholar,
Department of Computer Science and Engineeering,
Indian Institute of Technology Guwahati,
Guwahati, Assam, INDIA 781039,
Email: **panthadeep@iitg.ac.in, panthadeep.edu@gmail.com**

Date: February 8, 2021
Place: IIT Guwahati

Certificate

This is to certify that the thesis bearing title **"Density-Based Mining Algorithms for Dynamic Data: An Incremental Approach"**, being submitted by **Mr. Panthadeep Bhattacharjee** to the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, is a record of bonafide research work carried out by him under my supervision and is worthy of consideration for the award of the degree of Doctor of Philosophy in accordance with the regulation of the Institute.

....

Dr. Pinaki Mitra
(Thesis Supervisor),
Associate Professor,
Department of Computer Science and Engineeering,
Indian Institute of Technology Guwahati,
Guwahati, Assam, INDIA 781039,
Email: **pinaki@iitg.ac.in**
Website: https://www.iitg.ac.in/cse/internet-pages/pinaki

Date: February 8, 2021
Place: IIT Guwahati

Acknowledgements

I express my sense of thanks and gratitude to everybody whose support, guidance, wishes, blessings and prayers have enabled me to carry out my research. First and foremost, I would like to thank my PhD. thesis supervisor Dr. Pinaki Mitra for his guidance, patience and encouragement throughout. His in-depth understanding of Algorithms has enabled me to cover the important facets of my research with a greater detail. I would also like to extend the gratitude to my Doctoral Committee members: Dr. Pradip Kumar Das, Dr. V Vijaya Saradhi and Dr. Jitendriya Swain for their timely and constructive suggestions besides thanking the other faculty members of the CSE department.

I would also like to thank Mr. Monojit Bhattacharjee, Mr. Raktajit Pathak, Mr. Nanu Alan Kachari, Mr. Bhriguraj Borah and all other staff members of the CSE department who have helped me at different times. In addition, I convey my appreciation to all the hostel and canteen staffs, security guards, housekeeping staffs for ensuring an overall convenient life in the campus.

Life gets complemented with a set of good friends. Two of my best friends since the undergraduate days viz. Nilu Das and Nilotpal Chakraborty have had a huge positive influence in my career and otherwise. My association with them stretches to more than a decade now and I will always remain indebted to them for their support and co-operation at all times. I am also thankful to my other friends from the undergraduate and post-graduate days viz. Abhay, Rudradeep, Sambit, Anil, Bhaskar, Manoj, Jitender, Prasenjit, Subhendu, Kaustuv, Neeraj, Atanu, Sailesh, Ashok for always showing their sense of kindness and support throughout. Their inspiration and good wishes have enabled me to maintain a zeal of hard work and patience for countering some of the greater challenges that lie within and beyond academics. In addition, the love and affection showed by my beloved juniors like Prasun, Praveen, Mayank, Saprativa, Abhishek, Mrinmoy will also remain noteworthy.

I was also fortunate enough to be surrounded by an inspirational peer group during my PhD. duration. This includes Sandeep, Pradeep, Vasudevan, Achyut Mani, Rakesh Tripathi, Rakesh Pandey, Bala, Partha Sarathi, Piyoosh, Sadhu, Awnish, Basant, Surajit to name a few. They formed an integral part of all the academic or non-academic discussions that we have had in various times. Besides, I am also thankful to all the anonymous reviewers of my papers and thesis.

Last but not the least, I would like to convey my heartfelt thanks to my Parents: Smt. Gouri Bhattacharjee and Shri Janmejoy Bhattacharjee. Their constant support and motivation have always kept me focused in my research work and enabled me to maintain the virtue of perseverance. The mental and emotional support provided by them especially by my mother has been of highest order. It will be only fair to say that she has acted like an infinite source of support, blessings and guidance among others right from the beginning for which I will always remain grateful.

Abstract

Typically an algorithm designed for carrying data mining tasks is fed with a static set of input. This class of algorithms remain prone to certain disadvantages in scenarios where the input data and extracted results change temporally. The prominent bottlenecks may include redundant computation, high response time along with increased consumption of available resources. Given the importance of handling dynamic data in a real time environment eg: traffic monitoring, medical research, recommendation systems etc., this thesis focuses on developing incremental mining algorithms particularly in the field of density based clustering and outlier detection.

Density-based algorithms display robustness in extracting clusters of varying granularity or filtering outliers from variable density sub-spaces. In this thesis, we propose incremental extensions to two density based clustering algorithms: MBSCAN (Mass-based Clustering of Spatial Data with Application of Noise) and SNN-DBSCAN (Shared Nearest Neighbor Density Based Clustering of Large Spatial Data with Application of Noise). While dealing with outlier detection, an incremental density based approach is proposed for the K-Nearest Neighbor Outlier Detection algorithm known as KNNOD. The incremental extensions to MBSCAN and KNNOD are approximate in nature facilitating single point insertions. However for SNN-DBSCAN, we propose exact incremental solutions facilitating both addition and deletion of data in batch mode.

Our first contribution known as the *iMass* (Incremental Mass Based Clustering) clustering algorithm offers an approximate incremental solution to the static MBSCAN algorithm. The goal of this work is to identify the expensive building blocks of MBSCAN and reconstruct them incrementally post every new insertion. Observations combining six real world and two synthetic datasets showed that the proposed *iMass* algorithm outperformed the naive MBSCAN method by achieving a maximum efficiency upto an order of 2.28 ($\approx$ 191 times). Around 60.375% of mean clustering accuracy was observed post final insertion for three unlabeled datasets. The cluster quality evaluation through Normalized Mutual Information (NMI), Rand index (RI) measure and F1-score for five class labeled datasets showed similar or improved results for *iMass* as compared to MBSCAN. The efforts laid in our first contribution therefore motivated us to expand our research towards proposing exact incremental solutions.

The second contribution in form of our proposed clustering algorithm $BISDB_{add}$ (Batch Incremental Shared Nearest Neighbor Density Based Clustering Algorithm for addition) provides an exact incremental solution to the naive SNN-DBSCAN algorithm while adding points in batch mode. $BISDB_{add}$ comprises of two proposed sub-variant algorithms viz. $Batch - Inc1$, $Batch - Inc2$ and is the most efficient comparatively. $BISDB_{add}$ targets all the components of SNN-DBSCAN incrementally unlike its sub-variant methods. $BISDB_{add}$ achieved a maximum efficiency upto an order of 3 ($\approx$ 1000 times) over five (three real world and two synthetic) datasets. An identical cluster similarity was also observed with that of the SNN-DBSCAN algorithm.

Complementing addition of data, the third contribution proposes the algorithm $BISDB_{del}$ (Batch Incremental Shared Nearest Neighbor Density Based Clustering Algorithm for deletion) thereby providing an exact incremental solution to SNN-DBSCAN while deleting points in batch mode. Similar to $BISDB_{add}$, $BISDB_{del}$ comprises of two proposed sub-variant algorithms viz. $Batch-Dec1$, $Batch-Dec2$ and is the most efficient comparatively. $BISDB_{del}$ targets all the components of SNN-DBSCAN incrementally when points are deleted from the dataset unlike its sub-variant methods. On comparing with SNN-DBSCAN, the maximum efficiency achieved by $BISDB_{del}$ reached upto an order of 4 ($\approx$ 10000 times) over five (three real world and two synthetic) datasets. The set of clusters obtained were identical to the SNN-DBSCAN algorithm.

Moving from the paradigm of clustering, our fourth and final contribution focuses on dynamic extraction of at most top-N global outliers against single point insertions. Our proposed approximate incremental algorithm KAGO (Adaptive Grid Based Outlier Detection Approach using Kernel Density Estimate (KDE)) uses Gaussian kernel in a grid-partitioned space to determine the local density of a point. The local density obtained through KDE is used to filter the local outliers which are integrated to extract at most top-N global outliers. The KAGO algorithm outperformed KNNOD by achieving a maximum efficiency upto an order of 3.91 ($\approx$ 8304 times) over two intrusion detection datasets and a bidding data for market advertisement related to a search engine. Outliers' evaluation on these datasets using RI and F1-score showed a mean improved accuracy of around 3.3% in case of KAGO.

The thesis therefore strives towards developing approximate and exact incremental algorithms in the field of density-based clustering and outlier detection thereby facilitating real time data analysis.

Contents

List of Figures

List of Tables

List of Algorithms

Sl no.	**Algorithm**
1	$iMass$ (Incremental Mass Based Clustering) [Chapter 3, Section 3.7].
2	$Batch - Inc1$ [Chapter 4, Section 4.7.1].
3	$Batch - Inc2$ [Chapter 4, Section 4.7.2].
4	$BISDB_{add}$ (Batch Incremental Shared Nearest Neighbor Density Based Clustering Algorithm for addition) [Chapter 4, Section 4.7.3].
5	$Batch - Dec1$ [Chapter 5, Section 5.6.1].
6	$Batch - Dec2$ [Chapter 5, Section 5.6.2].
7	$BISDB_{del}$ (Batch Incremental Shared Nearest Neighbor Density Based Clustering Algorithm for deletion) [Chapter 5, Section 5.6.3].
8	KAGO (Adaptive Grid Based Outlier Detection Approach using Kernel Density Estimate) [Chapter 6, Section 6.7].

List of Abbreviations

Abbreviation	Full form
$BISDB_{add}$	Batch Incremental Shared Nearest Neighbor Density Based Clustering Algorithm for addition.
$BISDB_{del}$	Batch Incremental Shared Nearest Neighbor Density Based Clustering Algorithm for deletion.
DBCLAs	Density Based Clustering Algorithms.
$iMass$	Incremental Mass Based Clustering.
KAGO	Adaptive Grid Based Outlier Detection Approach using Kernel Density Estimate.
KDE	Kernel Density Estimate.
KNN	K-Nearest Neighbors
KNNOD	K-Nearest Neighbor Outlier Detection.
MBSCAN	Mass-based Clustering of Spatial Data with Application of Noise.
NMI	Normalized Mutual Information.
RI	Rand index.
SNN	Shared Nearest Neighbors.
SNNDB	Shared Nearest Neighbor Density Based Clustering of Large Spatial Data with Application of Noise.

List of Notations

Notation	Meaning
C	Set of Clusters.
C'	Set of Clusters after dataset is updated.
D	Original(Base) dataset.
D'	Changed dataset after a point or batch insertion/deletion.
n	Size of base dataset.
k	Number of new point insertions/deletions/batch size.
δ_{core}	Core point formation threshold.
$\mathcal{P}(.)$	Power set.
$Core(.)$	Set of core points of dataset.
$Non\text{-}Core(.)$	Set of non-core points of dataset.
$\lvert.\rvert$	Size of a set.
$R(.)$	Region of space.
$P(R(.))$	Probability mass of the region.
$\mathcal{H}(.)$	Set of hierarchical models for partitioning a region.
H	A particular hierarchical model.
B	Number of batches.
k'	Total no. of points to be inserted in batch mode.
δ_{sim}	Strong link formation threshold.
δ_{core}	Core point formation threshold.
$Sim\_Mat(.)$	Similarity matrix of dataset.

$adj(.)$	Set of adjacent points to a given point.
K	Size of the K-Nearest Neighbour list.
KNN(.)	KNN list of any data point.
O	Set of outliers in base dataset.
O'	Set of outliers after dataset is updated.
Lc_j	j^{th} kernel center.
p	No. of partitions per dimension.
d	No. of dimensions.
S	The set of points within a grid cell.
$\because$	Since.
$\therefore$	Therefore.
$\rightarrow$	Produces.

Citations to Published work

Chapter 1: Section 1.3.1; Chapter 2: Sections 2.3, 2.5 are based on the following paper(s):

- "A Survey of Density Based Clustering Algorithms", **Panthadeep Bhattacharjee**, Pinaki Mitra. *Frontiers of Computer Science* (SCI-e), Springer/Higher Education Press, September 2020. ISSN 2095-2236 (Online). *DOI:*10.1007/s11704-019-9059-3, Volume 15(1), Article number: 151308 (2021). URL: https://link.springer.com/article/10.1007/s11704-019-9059-3.

Chapter 3 is based on the following paper (s):

- "*iMass*: An Approximate Adaptive Clustering Algorithm for Dynamic Data Using Probability Based Dissimilarity", **Panthadeep Bhattacharjee**, Pinaki Mitra. *Frontiers of Computer Science* (SCI-e), Springer/Higher Education Press, October 2020. ISSN 2095-2236 (Online). *DOI:*10.1007/s11704-019-9116-y, Volume 15(2), Article number: 152314 (2021). URL: https://link.springer.com/article/10.1007/s11704-019-9116-y.

Chapter 4, Chapter 5 are based on the following paper (s):

- "BISDBx: towards batch-incremental clustering for dynamic datasets using SNN-DBSCAN", **Panthadeep Bhattacharjee**, Pinaki Mitra. *Pattern Analysis and Applications* (SCI-e), Springer, May 2020. ISSN 1433-755X (Electronic), *DOI* : 10.1007/s10044-019-00831-1, Volume 23(2), pages 975-1009. URL: link.springer.com/article/10.1007/s10044-019-00831-1.
- "Incremental Mining Algorithms: Adapting to Dynamic Data", **Panthadeep Bhattacharjee**, Pinaki Mitra. *24th International Conference on Advanced Computing and Communications* (ADCOM 2018), ACCS, PhD Forum, Bangalore, India, September 2018, ISBN 978-93-5321-421-0, pages 110-113. URL: https://accsindia.org/adcom-2018-conference-proceedings/.

Chapter 6 is based on the following paper(s):

- "KAGO: An Approximate Adaptive Grid Based Outlier Detection Approach using Kernel Density Estimate", **Panthadeep Bhattacharjee**, Ankur Garg, Pinaki Mitra. *Pattern Analysis and Applications*, Springer, (**Under Review**).

Chapter 1

Introduction

1.1 Incremental Algorithms

Given a set of input sequence X on an existing dataset D, an incremental algorithm A_{inc} produces a set of output sequence O_{inc} by intelligently adapting to the changes in input. On the contrary, when the same input sequence X is processed by a non-incremental (naive) algorithm A_{naive}, the efficiency, latency and resource consumption may be compromised while producing an exact or approximate output O_{naive} with respect to (w.r.t.,) A_{inc}.

Formally, let $X = \{i_1, i_2, i_3, \dots, i_k\}$ [$k \in \mathbf{N}$ (set of natural numbers), $|X| = k$] be the set of data items inserted to or deleted from an existing dataset D. The elements of X may be fed to algorithm A_{inc} in a point-wise manner or batch mode. Let an update sequence of data sets $x_1, x_2, x_3, \dots, x_m$ be processed by A_{inc}. After processing each set x_r [$1 \le r \le m$, $m \in \mathbf{Z}^+$ (set of positive integers)] where:

- $x_r \subseteq X$, $\bigcup_{r=1}^{m} x_r = X$.
- $1 \le |x_r| \le k$, $1 \le m \le k$.
- For $p, q \in \mathbf{Z}^+, p \ne q$ and $1 \le p, q \le m$, we have $x_p \cap x_q = \phi$.

let A_{inc} produce a set of output sequence $O_{inc} = \{o_{i1}, o_{i2}, o_{i3}, \dots, o_{im}\}$. $\forall o_{ij} \in O_{inc}[1 \le j \le m]$, $o_{ij} = \{y | y \in f_{inc}(x_j \cup D)\}$ where function $f_{inc}(.)$ is a part of A_{inc}. Let the time taken to execute A_{inc} be T_{inc}.

Assuming that the algorithm A_{naive} is fed with an identical input sequence to A_{inc}, then **instead of selectively dealing with set x_i, the entire updated dataset $x_i \cup D$ is processed**. The output sequence produced by A_{naive} may be represented as $O_{naive} = \{o_{n1}, o_{n2}, o_{n3}, \ldots, o_{nm}\}$. $\forall o_{nj} \in O_{naive}[1 \leq j \leq m]$, $o_{nj} = \{y | y \in f_{naive}(x_j \cup D)\}$ where function $f_{naive}(.)$ is a part of A_{naive}. Let the time taken towards completion of A_{naive} be T_{naive}. Therefore, **upon execution of the presented sequence to both the algorithms A_{inc} and A_{naive}, we have** $T_{inc} < T_{naive}$.

Based on the comparison between output sequence O_{inc} and O_{naive}, A_{inc} can be said to produce either an approximate or an exact incremental solution to A_{naive}. The following conditions categorize the incremental algorithm A_{inc} accordingly:

- $\exists o_{ij} \in O_{inc}, \exists o_{nj} \in O_{naive}$ where $1 \leq j \leq m$, if $o_{ij} \approx o_{nj}$, then A_{inc} is an approximate incremental extension to A_{naive}.
- $\forall o_{ij} \in O_{inc}, \forall o_{nj} \in O_{naive}$ where $1 \leq j \leq m$, if $o_{ij} = o_{nj}$, then A_{inc} is an exact incremental extension to A_{naive}.

1.1.1 Some examples of incremental algorithms in data mining

We mention some of the incremental algorithms of choice which include tasks like clustering, pattern extraction, network intrusion detection to name a few.

1. **Incremental K-means clustering algorithm [6]:** An incremental extension of K-means clustering algorithm involves addition of cluster centers singularly in course of cluster formation. The scheme adopted in this work has been designed to reduce the cluster distortion by initiating movement of cluster centers.
2. **Incremental document clustering [7]:** In context of clustering web documents, an incremental approach becomes important as tasks like accessing, browsing and searching of large repositories are involved. In this algorithm, a pair-wise document similarity information is utilized where a similarity histogram is used for representing the clusters.
3. **Dynamic Information Retrieval [8]:** In applications related to document and image classification, the task of clustering points in a dynamic

environment becomes critical. The algorithm attempts to maintain clusters of smaller diameter while new points are being inserted.

4. **Incremental Algorithm Based on FP-growth for Big Data [9]:** The rapid use of big data has led to an increase in new transactions on a steady basis. Therefore the results of association rule mining called frequent item-sets should be updated with change in time. This technique relies on the concept of heap tree while addressing the issue of updating frequent item-sets incrementally.

5. **Real time network intrusion detection [10]:** The work proposes a technique to detect intrusions in network on a real time basis. An approach of incremental mining for fuzzy association rules is adopted in this case. Two sets of rules mined from online source and training of attack-free packets are compared. The proposed approach is able to produce a result every two seconds. This leads to a significant improvement in efficiency as compared to static approaches of mining.

6. **On-line failure prediction [11]:** An architecture has been proposed by the said work for predicting online failures in air traffic control system. The proposed architecture named as CASPER [11] exhibits a greater degree of accuracy in forecasting failures. The proposed scheme utilizes no prior knowledge about internal applications, information about CPU utilization or amount of memory consumed.

1.2 Dynamic data and its applications

Dynamic data changes over time in an asynchronous manner with the availability of new information [12]. The rapid use of dynamic data by various applications in domains such as online recommender systems, health care, traffic monitoring, cyber security, web-crawling etc. have increased significantly over the years. In this section we mention certain applications of choice to highlight the relevance of dynamic data.

1. **Recommender systems:** A recommendation system involves information filtering in order to predict the rating or preference of a user. Recommender systems make use of either content based filtering or collaborative filtering [13]. A user's past choices eg: items purchased, videos watched, channels

subscribed, content browsed etc. play a major role in collaborative filtering to make future recommendations. However, content based filtering make use of prior ratings or tagged characteristics to recommend items with similar features.

Every choice by a user may be treated as a change in data and hence the predictions can be made accordingly. Applications such as YouTube [14] involves content based filtering for recommendations depending on the user's profile and the description of video content. Similarly e-commerce sites like Amazon [15] use collaborative filtering based on customer's interests for generating a list of recommended items. Therefore, we observe the prevalence of dynamic data in form of ordered items or chosen contents on such platforms for future recommendations.

2. **Traffic monitoring:** With growth of population in urban areas, the road networks are expanding rapidly. The traffic congestion are a direct result of the frequent movement of people on the road [16]. A GPS service provider may gather data from individual users about their locations, speeds, mobility, etc. The dynamic data such as the number of users at different regions during each time period, can be mined for commercial interest like determining congestion patterns on the roads.

3. **Medical field:** In the field of medical research, the application of dynamic data can be mapped to individual patients. Every new addition of symptoms like cough, cold, fever, demographic attributes can be taken as the features pertaining to a given patient. The information may be shared with researchers for discovery, seasonal epidemic outbreak monitoring [17]. Any epidemic monitoring system involved in precise discovery of infected individuals ensure a proper understanding of the characteristics of epidemics. This may help in further prevention of diseases. Moreover, physical participation of patients can be potentially avoided. Use of social media in this regard may also turn out to be of great help while developing a computational epidemic model.

4. **Eye movement dataset:** For the task of visual action recognition, dynamic eye movement datasets are used [18]. Large quantity of eye movement data are collected in order to study and construct automatic, end-to-end computer vision systems. These systems can be trained based on human eye movements.

5. **Web-crawlers:** Dynamic-content web crawlers are installed at locations between user and the server. The task of these crawlers is to monitor contents from points where it is placed, eg: being a proxy to the web traffic or sniffing the web traffic as the contents go by. The parsing of web page content is done recursively into sub-components [19]. Sub-components go through a fingerprint check enabled by a cyclic redundancy checking code so that it is able to determine the repeat occurrence of the sub-component in pages to follow.

6. **Cyber security:** In order to detect changes in real time, the velocity of data streams and other network characteristics, an ensemble-based approach may be used for generating functions. These functions are utilized for combining the classifiers and strategies to respond to data changes [20].

1.3 Data mining tasks of choice

The two data mining tasks of choice leading towards our contributions in this thesis are:

- Density-based clustering.
- Density-based outlier detection.

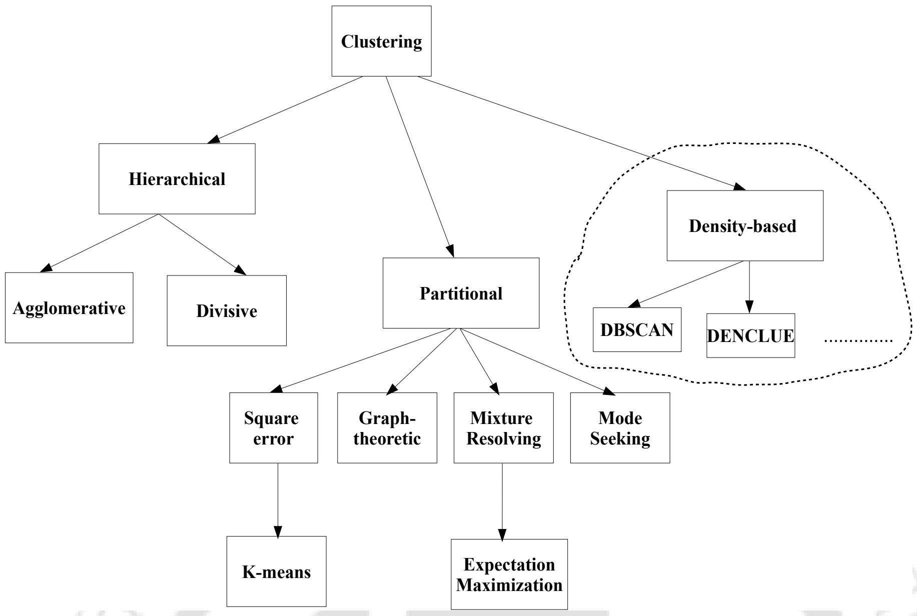
Next we mention the essential characteristics of density based clustering and outlier detection.

1.3.1 Density-based clustering

Clustering is a learning task that groups data objects or patterns based on similarity measures. Such objects may exist as data points in a $\mathcal{R}^d$ (d-dimensional) space. Entities belonging to a certain cluster have greater similarity between them than that with an entity belonging to a different cluster [21, 22].

Out of all the clustering domains (Figure 1.1), in this thesis we particularly focus on providing incremental extensions in approximate or exact form to density based clustering algorithms (DBCLAs) of choice. The objective of DBCLAs is to find clusters at different levels of granularity with appropriate noise filtering. The

Figure 1.1: Representation of various clustering domains.



notion of density used by the DBCLAs enables segregation of compact regions in the data space from noise. With use of DBCLAs, clusters are identified as areas of higher density than the remainder of the data space [23]. Several DBCLAs eg:SNN-DBSCAN [24] also facilitate detecting clusters of arbitrary shapes and densities. Over a period of last two decades, numerous density based clustering techniques have been proposed. A majority of these methods aim to extract clusters of relatively uniform densities lying across the data space.

The motivation behind advent of DBCLAs results from the drawbacks present in other clustering domains viz. partitional clustering, hierarchical clustering, fuzzy clustering, nearest neighbor based clustering or evolutionary approaches. Depending on the properties of data and the mechanism adopted to form clusters, we mention the flaws in other areas of clustering as compared to DBCLAs:

1. **Inability to detect clusters correctly in high dimensional space:** Most of the challenges faced by clustering algorithms are particularly related to the quality of clusters obtained in voluminous high dimensional space. The phenomenon of "curse of dimensionality" [25] is one of the major bottlenecks due to which quality of clusters cannot be guaranteed among data points having numerous attributes.

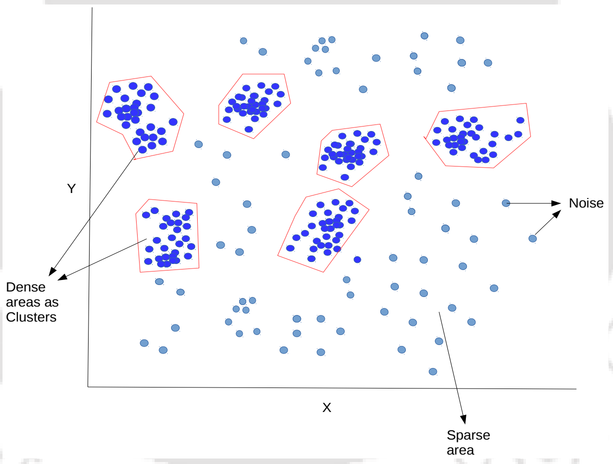
2. **Resource constraint:** With the use of very large datasets, there also exists an issue regarding availability of computational resources. Clustering

algorithm such as CLARANS [26] is based on the assumption that its data reside within memory, an approach that fails for very large datasets.

3. **Inability to detect non-globular clusters:** Existing literature have also pointed out the inability of various methods to extract arbitrarily shaped clusters having variable densities. Partitional method such as K-means [27] finds convex shaped clusters. It fails to identify clusters of non-globular shapes. The algorithm also shows its limitations while dealing with noisy data. Agglomerative clustering techniques [28, 22, 21] are not as affected by noise, but they have a tendency towards identifying globular clusters.

4. **Overlooking clusters in areas of uniform density:** For spatial databases, we can often find regions of uniform density located at a remote area. Such regions usually go undetected and the probable clusters existing in those regions are potentially overlooked by many clustering algorithms.

FIGURE 1.2: Dense areas represent the clusters and subsequently the noises are filtered out. DBCLAs aslo enable cluster detection in areas of uniform density irrespective of the size of region.



In order to address these challenges, the class of DBCLAs came into existence. The following points underline the reasons for robustness of the DBCLAs.

1. DBCLAs discover clusters as dense regions separated from the areas of lower density [23] (Figure 1.2). The density of any region is given as the number of points within that region or in terms of its kernel density estimate [29].

2. DBCLAs enable appropriate noise filtering [23] (Figure 1.2).

3. DBCLAs aim at exploring the data space with varied levels of granularity [30]. This enables detection clusters in regions of uniform density.

FIGURE 1.3: Detecting clusters of arbitrary shapes and densities.

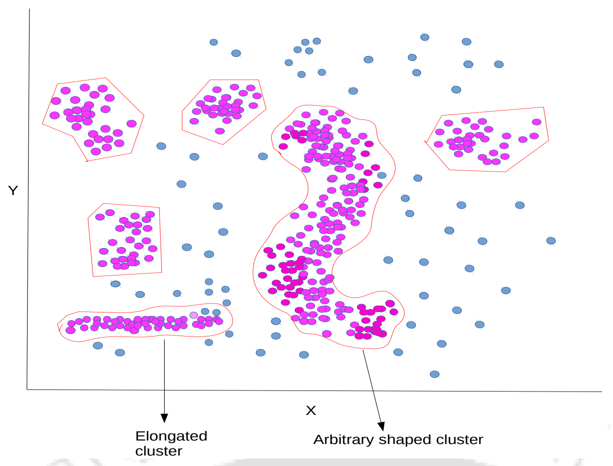
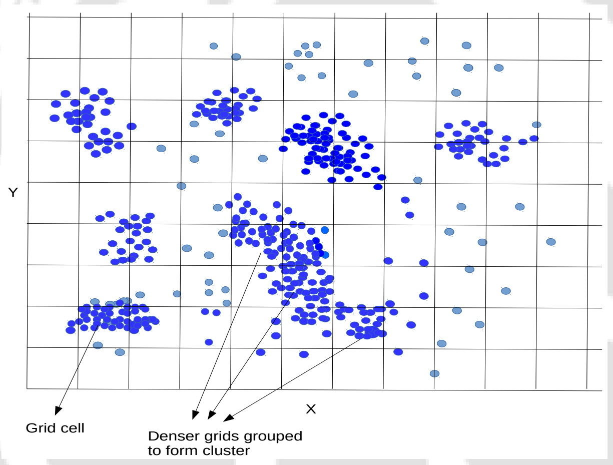


FIGURE 1.4: Dense grid cells accumulate to form clusters.



4. Exploring data at higher levels of granularity enables DBCLAs to reconstruct the entire shape of the data distribution [31].

5. DBCLAs facilitate detection of arbitrary shaped clusters with varying sizes and densities [24]. A post processing phase is considered in order to accumulate the dense region into an arbitrary shape (Figure 1.3).

6. DBCLAs also use grid based methods to explore individual regions of the data space and form clusters [30] (Figure 1.4).

1.3.1.1 Applications of density-based clustering algorithms

We mention some of the relevant applications where DBCLAs find their importance.

1. **Earth sciences:** In the field of earth sciences, spectral space obtained from satellite images is clustered. This is a task commonly used in remote sensing image analysis. In this application, the said algorithm (GDBSCAN [32]) uses a 5-D space gathered from the satellite imagery of a region on earth's surface. Clusters obtained through this procedure finds importance in analysis of digital images in remote sensing. Algorithms such as SNN-DBSCAN [24] also finds its application in earth sciences data. In particular, researchers are interested in finding the areas of ocean whose behavior correlates well to climate events on the earth's land surface.

2. **Molecular biology:** Bio-molecules, proteins, DNA and other cell components consist of millions of atoms within them. These cell components interact with each other eg: protein protein interaction (PPI). PPI contains dense regions that can be identified based on the data connectivity [33]. The spatial arrangement of the molecular structures at the site of interaction holds importance in addition to physio-chemical molecular behavior highlighting the application of clustering.

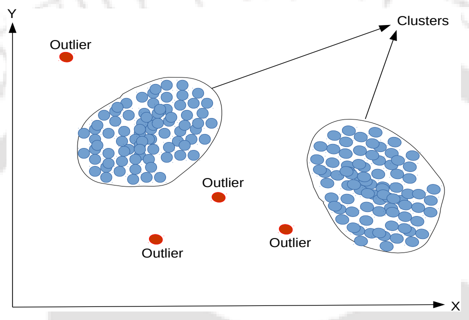
3. **Astronomy:** In the field of astronomy, the data acquired from celestial objects may lead to the discovery of various patterns. Grouping of such data is necessary for detecting the presence of any pattern or any other relevant information for mining. Surveys detecting celestial objects of interest perform their statistical studies. Such observations may extract sporadic or irregular objects [32]. One of the main reasons to adopt DBCLAs for such purpose is its ability to segregate compact regions from the sparse areas in the data space. This enables detection of clusters in the galactic survey along with appropriate noise filtering.

4. **Geography:** DBCLAs can be used to retrieve two-dimensional polygons based on the similarity measure of non-spatial attributes. In order to characterize the proximity between attributes, the domain is segregated to some distinct classes. The values lying within the same class are said to be similar. Sets that have the least or highest feature values become the influence regions. GDBSCAN [32] algorithm extracts such regions of influence. In the area of geospatial clustering, DBSCAN [23] also plays an important role in GIS (Geographic Information System) spatial analysis techniques such as polygon overlay [34].

1.3.2 Density-based outlier detection

Anomaly or outlier detection relates to the task of filtering patterns in data that deviate from normal behavior. These non-conforming or deviating patterns are often designated as anomalies, outliers or exceptions [35]. Figure 1.5 demonstrates the outliers and normal patterns in a 2-D data. The points which appear in isolation from the expected patterns are shown as outliers while the two groups of accumulated points in close neighborhood of each other form the clusters.

FIGURE 1.5: Illustration of outliers in a 2-D data.



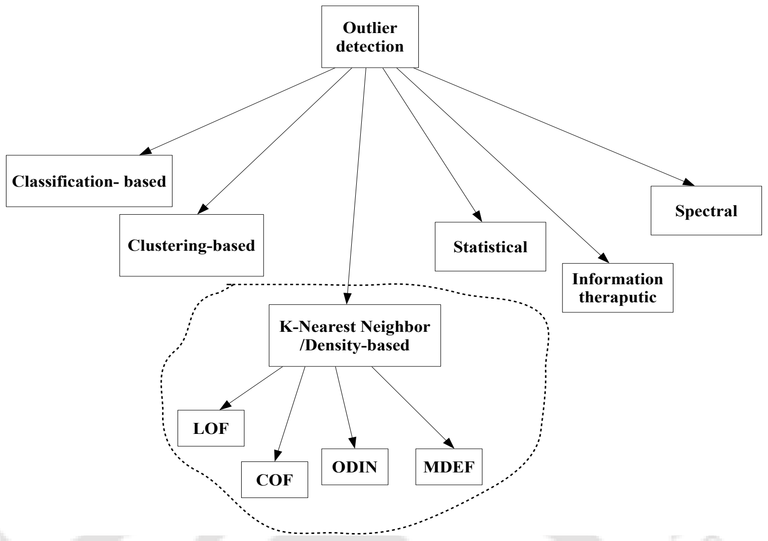
In this thesis, we focus on density based outlier detection approach out of all the outlier extraction domains (Figure 1.6). The K-Nearest-Neighbor (KNN) approach is considered as one of the basic density based techniques to detect anomalies [36]. However the KNN approach shows its limitations while extracting outliers from regions of variable densities. In order to deal with this issue, the idea of finding local density came into existence. The local density of a point is computed based on the relative density of its neighboring points. Any instance of data lying within a neighborhood of lower density may be called as an anomalous object.

Evaluation of local density pertaining to any data item is accompanied by its assigned anomaly score known as *Local Outlier Factor* (LOF) [37]. In case of any normal point residing within a region of higher density, the local density of that point is expected to be similar to its neighboring points. For any outlier, an estimated lower local density may be obtained compared to its K nearest neighbors. Most density based anomaly detection techniques are based on the variations of LOF which essentially involves the KNN of a data instance.

The key advantages of density-based anomaly detection approach are:

1. No prior assumptions are made regarding the distribution of data. The approach is data dependent.

FIGURE 1.6: Representation of various outlier detection paradigms.



2. Adapting to KNN-based approach for determining density is simpler as it primarily involves defining a suitable measure of distance for the data item.

3. Finding local density of data points can be more meaningful and advantageous in data space involving skewed distribution.

Next, we mention some of the relevant applications where the use of outlier detection has been made for the required purpose.

1.3.2.1 Some applications of outlier detection algorithms

1. **Network intrusion detection:** For many years, anomaly detection in networks have been a challenging area of research. Challenges lie in detecting a high false alarm rate along with issues in obtaining a cleaner data for modeling patterns that are normal. Anomaly detection approach based on TCM-KNN (Transductive Confidence Machines for KNN) [38] enable detection of outliers with a high true positive rate.

2. **Credit card transactions:** The operation sequence in credit card transactions is modeled using a Hidden Markov Model (HMM) [39] for fraud detection. Initially, the HMM is trained with a normal cardholder. However if a new transaction doesn't comply with the acceptance level of the trained HMM, the perception of fraud tends to happen.

   Use of artificial immune system (AIS) [40] is also termed as beneficial while detecting frauds in credit card transactions. The reason may be attributed to the fact that AIS can be built for flagging off dubious transactions without any knowledge of prior examples.

3. **Healthcare:** Wireless sensor networks (WSN) remain susceptible to numerous faults and anomalous calculations. In the domain of healthcare, any uncalled for alarm may involve unnecessary involvement of medical professionals. As a result, enabling proper differentiation between actual health issues and a false indication improve the task of patient monitoring [41]. Using medical sensors to collect physiological data, true alarms can be picked out neglecting the anomalous signals.

4. **Safety critical system:** A behavior-rule based technique has been proposed to detect intrusions (BRIDS) [42] for providing security to distribution access points, energy meters of subscribers in modern electrical grids where continuity in performance is important. The proposed scheme investigates the consequence of behaviors by attackers on the strength of the BRIDS method.

1.4 Challenges: Why do algorithms adapt?

An algorithm may adapt in order to deal with following challenges:

1. **Efficiency:** Increase in disproportional amount of run time by naive algorithm inflicted due to continuous changes in input data.

2. **Latency:** Due to speedy changes in the input, producing a correct and timely output without any delay may be a challenge.

3. **Resource usage:** Use of excessive computing resource due to involvement of entire dataset is also a major bottleneck.

Next we mention the role of incremental algorithm in addressing the aforementioned challenges.

1.4.1 Improve efficiency:

Whenever data resorts to a differential amount of change, an incremental algorithm A (say)[1] attempts to save time by reconstructing output that depends on the changed data. Upon successful execution, algorithm A can be substantially faster as compared to any naive approach. In order to illustrate this idea, let us consider the following scenario:

Let I_1 be the input to any naive algorithm N (say) producing output O_1. When a differential change δI is inflicted upon I_1, I_1 changes to I_2. Let the changed output be represented by O_2. If $T(.)$ represents the time required to process a given input by an algorithm, then we may have the following scenario:

$$\begin{aligned} N(I_1) &\rightarrow O_1 \\ I_1 \pm \delta I &\rightarrow I_2 \\ N(I_2) &\rightarrow O_2 \\ |O_2 - O_1| &\rightarrow \delta O \\ \because \mathit{size}(\delta I) < \mathit{size}(I_2); A(\delta I) &\rightarrow O_2 \\ \therefore T(A(\delta I)) &< T(N(I_2)) \end{aligned} \tag{1.1}$$

Since a minor change δI is made upon I_1, the change towards output O_2 is also expected to be minimal and let it be δO (Equation 1.1). As a result the necessity to involve I_2 in its entirety against every update may not be a feasible approach for algorithm N, as it may involve redundant computation. The efficiency is therefore achieved by effectively involving only δI amount of change as input instead of I_2 (Equation 1.1)$\therefore A(\delta I) \rightarrow O_2$.

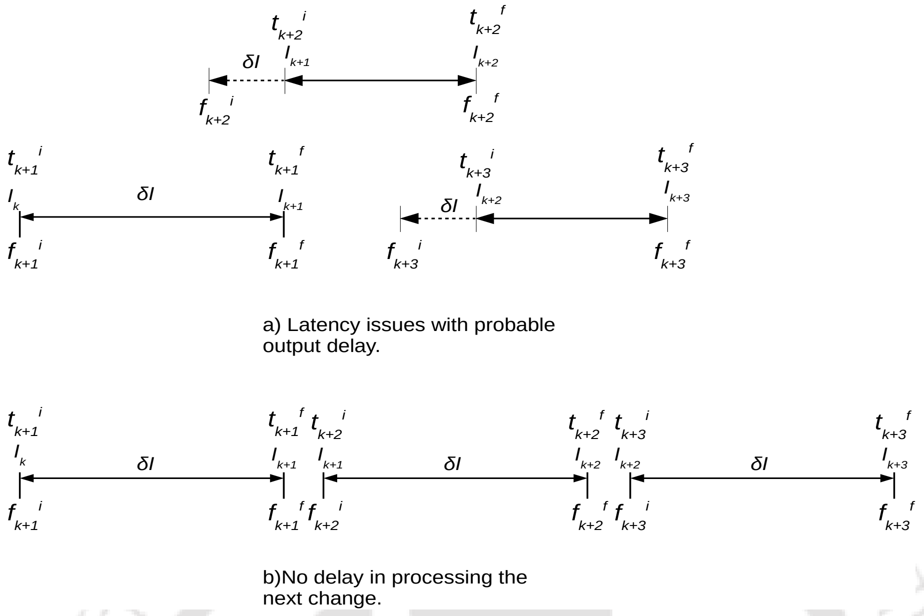
1.4.2 Reduce latency:

Expecting frequent changes being made to input data, after every δI amount of change, the task of finding correct output becomes critical before the next set of changes take place. Let I_k be the current input after $k-1$ number of changes on I_1. Therefore we have:

$$I_1 + (k-1)\delta I \rightarrow I_k \tag{1.2}$$

[1]The word "say" within parentheses denoted as (say) in this thesis signifies an assumption made regarding the name of an algorithm, variable or a value assigned to any parameter required for illustration.

Figure 1.7: Latency scenarios due to changes in the input data.



As demonstrated in Figure 1.7 (a (top figure)), let the input I_k be subjected to three differential changes through δI leading towards input I_{k+1}, I_{k+2} and I_{k+3}. Let f_x^i be the timestamp of initiating the x^{th} change on I_k. Let f_x^f be the timestamp at which the algorithm N finishes processing the x^{th} change while producing the output. Correspondingly, let t_x^i be the timestamp at which the execution of N in processing the change δI begins and t_x^f be its ending timestamp.

On close observation, we notice that the input I_k has been affected by δI changing to I_{k+1}. Let f_{k+1}^i, f_{k+1}^f be the timestamps of initiation and completion of processing the change. In this case, the starting time of execution of $N(I_k + \delta I)$ i.e., t_{k+1}^i is identical to f_{k+1}^i. While the ending timestamp of execution t_{k+1}^f coincides with f_{k+1}^f. When the next amount of change in form of δI initiates at timestamp f_{k+2}^i, the execution process of $N(I_k + \delta I)$ is still in continuation. As a result we have $t_{k+1}^f > f_{k+2}^i$. This leads to an unnecessary waiting time of $(t_{k+2}^i - f_{k+2}^i)$ for execution of $N(I_{k+1} + \delta I)$ to begin, which effectively delays the desired output. Similarly, by the time $N(I_{k+1}+\delta I)$ leads towards completion, a new change in form of δI is already inflicted. The initiation of change takes place at timestamp f_{k+3}^i which happens before t_{k+2}^f. Therefore an undesirable waiting time of $(t_{k+3}^i - f_{k+3}^i)$ is consumed prior to execution of $N(I_{k+2} + \delta I)$.

However, on observing Figure 1.7 (b (bottom figure)), the incremental algorithm A computes the output due to changes made upon input I_k dynamically. Therefore instead of processing the entire data $I_k + \delta I$, a selective handling of changes being made are taken into account. For the next element of change, we have $t_{k+1}^f <$

f^i_{k+2}. This establishes the fact that no additional waiting time is involved before processing the next update. Similarly, $t^f_{k+2} < f^i_{k+3}$ resulting in no waiting time before dealing with the third element of change.

1.4.3 Limit resource usage:

A minimal CPU utilization can be ensured by intelligently handling the frequent updates. Moreover, with selective processing of the data, limit to the memory consumption may also become possible.

1.5 Motivation

The motivation behind work done in this thesis is primarily driven by two major factors. Firstly, the inefficiency of naive algorithms in a dynamic environment and secondly, the robustness of data mining tasks to be incrementally extended. The following reasons may precisely depict the motivating factors:

1.5.1 Reasons inclining towards inefficiency of naive algorithms

1. The inability of naive or static algorithms to process dynamic datasets may be hazardous in terms of efficiency (Section 1.4) while dealing with real time applications [13, 14].

2. A high response time that can be incurred by a naive method may delay the result. Applications related to healthcare [41, 17], safety critical systems [42] can ill afford to allow the adoption of any naive algorithm with latency issues.

3. For any minor change in input from $I \rightarrow I'$ (say), a change in output from $O \rightarrow O'$ (say) is also expected to be minimal, $\therefore O \cap O' \approx \delta$ (a small entity). To extract the changed result, execution of naive algorithm involving data in its entirety may involve unnecessary redundant computation. This situation can be avoided with the use of incremental algorithms (Section 1.4).

4. Excessive consumption of computing resource eg:CPU usage, additional buffer requirements due to non-intelligent handling of continuous updates may turn out to be disadvantageous.

1.5.2 Reasons inclining towards robustness of data mining tasks to be incrementally extended

1. Density based algorithms enable finding of clusters with arbitrary shapes, sizes or filtering outliers from sub-spaces with variable densities (Section 1.3.1, Section 1.3.2). Moreover numerous real time applications related to cyber security [38, 36], fraudulent transactions [39], astronomy, earth sciences [24], healthcare [17] etc. fall under the area of density based clustering or outlier detection.

2. Providing incremental extension to a robust DBCLA known as MBSCAN [2] (Mass-based Clustering of Spatial Data with Application of Noise) may be justified as MBSCAN has the ability to detect clusters of variable densities. Moreover MBSCAN eliminates the use of any distance based density computation. Instead, it relies on a data dependent probability measure for evaluating the neighborhood strength of a point. This enables extraction of clusters with arbitrary shapes lying in different regions of the data space.

3. Dynamic handling of point-wise updates by any incremental algorithm based on SNN-DBSCAN [24] (Shared Nearest Neighbor Density Based Clustering of Large Spatial Data with Application of Noise) is indeed more efficient. However with increase in the size of base dataset[2], re-construction of expensive algorithmic components may lead towards inefficiency with every update. As a result, batch mode updates are facilitated in two of our contributions supporting both insertion and deletion of data.

4. Upon observing the importance of detecting outliers in applications related to intrusion detection [38], safety critical systems [42], fraud in credit card transactions [39] or health care monitoring [41] (Refer Section 1.3.2), the task of detecting outliers in a dynamic setup cannot be overlooked. As a result we chose to provide a density based incremental extension to a commonly used outlier detection algorithm known as KNNOD [5] (K-Nearest Neighbor outlier detection). Due to quadratic time complexity, KNNOD

[2]Dataset upon which updates are inflicted.

may incur heavy cost in terms of CPU execution time against frequent updates. Therefore framing an incremental extension to limit the use of KNN in density computation may lead towards greater effectiveness.

1.6 Objectives

Let us assume the following:

- T_{naive} = Time required to find output by naive method A_{naive}.
- T_{inc} = Time required to find output by incremental method A_{inc}.
- $|Mem_{inc} - Mem_{naive}|$ be the difference in percentage of average memory consumed.

Here $A_{inc} = A_{point-inc} \cup A_{batch-inc}$, where $A_{point-inc}$ and $A_{batch-inc}$ refer to incremental algorithms facilitating single point and batch-mode updates respectively.

- O_{naive} = Output obtained by naive method A_{naive}.
- O_{inc} = Output obtained by incremental method A_{inc}.

For a given data mining algorithm (clustering or outlier detection), we establish the following objectives:

- $T_{inc} < T_{naive}$
- $O_{naive} \approx O_{inc}$ (approximate incremental solution)
- $O_{naive} = O_{inc}$ (exact incremental solution)
- $|Mem_{inc} - Mem_{naive}| \leq \delta$ where δ is a small real number.

1.7 Primary Contributions

The primary contributions of this thesis are summarized as follows (in order):

1.7.1 The $iMass$ clustering algorithm providing an approximate incremental solution supporting point-wise insertion

The goal of this work is to identify the most expensive building blocks of MBSCAN [2] and re-construct them incrementally after each new insertion. Instead of using any geometrical model eg:distance, $iMass$ (Incremental Mass Based Clustering) relies on a probability measure to evaluate the neighborhood mass based density of any data point. The term "mass" w.r.t., a pair of points refers to the weight or strength of the smallest local region containing the concerned pair (See details in Chapter 3).

A hierarchical structure in form of isolation-Forest ($iForest$) is considered while evaluating the mass or probability mean mass [2] for any given pair of points. An $iForest$ is a combination of multiple isolation-trees ($iTrees$) with each $iTree$ representing a particular hierarchical model of any region (See details in Chapter 3). The mass value gives a measure of dissimilarity score between any two points. A dissimilarity mass matrix is constructed based on which the dense or non-dense points are determined. The dense points group to form clusters.

Experimental observations upon six real world and two synthetic datasets showed that the proposed $iMass$ algorithm outperformed the naive MBSCAN method by achieving a maximum efficiency upto an order of 2.28 ($\approx$ 191 times). Around 60.375% of mean clustering accuracy was obtained post final insertion for three unlabeled datasets. The cluster quality evaluation through NMI, RI and F1-score measures for five class labeled datasets showed similar or improved results for $iMass$ as compared to MBSCAN.

1.7.2 The $BISDB_{add}$ clustering algorithm providing an exact incremental solution in batch-mode for insertion

$BISDB_{add}$ (Batch Incremental Shared Nearest Neighbor Density Based Clustering Algorithm for addition) leads towards incremental addition of data points in batch-mode extending SNN-DBSCAN [24]. $BISDB_{add}$ comprises of two proposed sub-variant algorithms viz. $Batch - Inc1$, $Batch - Inc2$ and is the most efficient comparatively. $BISDB_{add}$ builds all the components of SNN-DBSCAN [24] incrementally when points are added to the dataset unlike its sub-variant algorithms.

Contrary to point based scheme, the batch-mode insertion policy eliminates repeated re-construction of expensive SNN-DBSCAN components (See details in Chapter 4).

$BISDB_{add}$ outperformed SNN-DBSCAN by more than an order of 3 ($\approx$ 1000 times) as observed across three real world and two synthetic datasets. An identical cluster similarity was observed with the SNN-DBSCAN algorithm.

1.7.3 The $BISDB_{del}$ clustering algorithm providing an exact incremental solution in batch-mode for deletion

$BISDB_{del}$ (Batch Incremental Shared Nearest Neighbor Density Based Clustering Algorithm for deletion) incrementally extends SNN-DBSCAN by supporting deletion of data points in batch-mode. $BISDB_{del}$ comprises of two other proposed sub-variant algorithms viz. $Batch-Dec1$, $Batch-Dec2$ and is the most efficient comparatively. $BISDB_{del}$ builds all the components of SNN-DBSCAN incrementally when points are deleted from the dataset unlike its sub-variant algorithms. The efficiency obtained by $BISDB_{del}$ reached upto an order of 4 ($\approx$ 10000 times) across three real world and two synthetic datasets. $BISDB_{del}$ achieved an identical cluster similarity with SNN-DBSCAN.

1.7.4 The KAGO outlier detection algorithm providing an approximate incremental solution supporting point based insertion

The proposed algorithm KAGO (Adaptive Grid Based Outlier Detection Approach using Kernel Density Estimate (KDE)) uses the Gaussian kernel in a grid partitioned space to compute the local density of a point. The local density obtained through KDE is used to find the local outliers. The local outliers are integrated to determine at most top-N global outliers against every insertion. Upon entry of each data point, a previously sparse region may become dense due to which a maximum of top-N outliers or less may be identified in course of multiple point insertions.

The KAGO algorithm outperformed KNNOD [5] by more than an order of 3.91 ($\approx$ 8304 times) across two intrusion detection datasets and a bidding data for market advertisement of a search engine. KAGO had a reduced memory consumption of about 51.57% on an average. Outlier evaluation on these datasets using Rand index and F1-score showed a mean improved accuracy of around 3.3% as compared to KNNOD.

1.8 Summary

The thesis primarily considers two data mining tasks: density based clustering and outlier detection for which incremental algorithms have been proposed. Out of four contributions made in this thesis, three are related to clustering while one contribution has been dedicated to outlier detection. A total of four incremental algorithms have been proposed out of which two algorithms, one each for second and third contribution comprises of two sub-variant methods. Two algorithms provide exact while two other provide approximate incremental solutions to the baseline algorithm. The second and third contribution deal with both insertion and deletion of points in batch node, while the remaining contributions handle only single point insertions.

1.9 Organization of the Thesis

1. Chapter 1: Introduces the idea of incremental algorithms and its applications in relevant domains. The chapter also highlights the importance of selected data mining tasks along with the motivation behind our contributions.

2. Chapter 2: Presents a study of related incremental algorithms in the field of density based clustering and outlier detection. It also provides a background of the methods that form the basis of our proposed incremental algorithms in this thesis.

3. Chapter 3: Presents the first contribution in form of the $iMass$ clustering algorithm. The proposed algorithm is an approximate incremental extension to the static MBSCAN [2] method supporting point-wise insertion.

4. Chapter 4: Contributes to an exact incremental extension of the SNN-DBSCAN [24] clustering algorithm. The proposed algorithm known as $BISD_{add}$ facilitates addition of data points in batch mode. It is a combination two

sub-variant algorithms: $Batch-Inc1$, $Batch-Inc2$ and is the most efficient comparatively.

5. Chapter 5: In this chapter, an exact incremental solution to SNN-DBSCAN [24] is proposed in form of the $BISD_{del}$ algorithm. $BISD_{del}$ facilitates deletion of points in batch mode. It is a combination two sub-variant algorithms: $Batch - Dec1$, $Batch - Dec2$ and is the most efficient comparatively.

6. Chapter 6: Presents the KAGO algorithm which leverages the idea of KDE to find local outliers. These outliers are aggregated to find at most top-N global outliers against every point insertion.

7. Chapter 7: Provides the concluding remarks along with the future scopes of pertaining contributions made in this thesis.

Chapter 2

Literature Survey

In this chapter, first we present our study of previous works on incremental density based clustering and outlier detection. Besides we also provide a background study of some naive algorithms that are related to the contributions made in this thesis.

2.1 Related density-based incremental clustering algorithms

1. **Incremental DBSCAN:** Inc-DBSCAN [43] is the incremental version of the DBSCAN [23] clustering algorithm. Patterns in database eg:log database alters temporally with new logs being added to and previous records are deleted from the database. The algorithm identifies the affected parts of existing clusters caused by an update in the database. Based on this underlying idea of selective handling of updated dataset, Inc-DBSCAN proves to be more efficient than DBSCAN. Post insertion of new points, some non-core (non-dense) objects may turn into core (dense) forming novel density connections. Points which were not density reachable [23] earlier might become density reachable. Similarly upon performing deletion, some core objects may turn into non-core resulting in removal of existing connections.

   If an object p is inserted or deleted, then $N_{Eps}(p)$ [23] (Eps neighborhood of p) becomes the affected region. The unaffected points retain their old cluster membership. The number of region queries performed by Inc-DBSCAN is determined experimentally. Let r_i and r_d denote the mean number of region queries during incremental insertion and deletion respectively. Let f_i and

f_d be the percentage of insertions and deletions. Then the cost incurred by Inc-DBSCAN for making r updates to the dataset incrementally is given as $r \times (f_i \times r_i + f_d \times r_d)$.

Inc-DBSCAN shows its inability in handling bulk insertion or deletion of data objects. The algorithm is also sensitive to changes in parameter values.

2. **IncSNN-DBSCAN:** IncSNN-DBSCAN [1] (InSDB) is an extension of the SNN-DBSCAN [24] clustering algorithm. InSDB detects clusters dynamically while points are added to the base dataset D one at a time. InSDB identifies each data point $p \in D$ with the following properties: KNN (K-Nearest Neighbors) list, strengths of shared links [1], number of adjacent links, dense or non-dense status. When a new data point arrives, InSDB identifies only those points which undergo changes in values of their properties. The algorithm targets only the affected points while rest of the points are allowed to exist in their previous state. This selective handling of data points ensures that the reconstruction time of the updated KNN lists and the shared nearest neighbor (SNN) graph [1] is significantly reduced. InSDB shows that a very small percentage of points ultimately gets affected due to which it becomes more efficient than SNN-DBSCAN [24].

   Since InSDB is a point-based insertion technique, it might slow down as the size of base dataset increases. It is also sensitive to change in parameter values.

3. **IGDCA:** The incremental grid density-based clustering algorithm (IGDCA) [44] enables discovery of clusters having arbitrary shapes. IGDCA is an incremental extension of the GDCA [44] algorithm. The clusters obtained through GDCA are modified after a sequence of insertions δ_{add} (say) and deletions δ_{del} (say) of data points. Let D be the base dataset and D' be the updated dataset where $D' = D + \delta_{add} - \delta_{del}$. Since the data space is partitioned into grid cells, a cell gets updated only when a data point is added to or removed from it. Once the affected grids are identified, the updated clusters are subjected to modification. New points obtain cluster membership followed by the modification of existing clusters.

   The algorithm is unable to determine the threshold parameters automatically. Moreover, the task of deletion also involves efficiency issues.

4. **Dynamic density based clustering:** This work [45] investigates the principles of dynamic clustering by DBSCAN [23] and the ρ-*approximate* version

of DBSCAN. The work proves that the *ρ-approximate* version suffers from the same computational hardness as the one when the dataset is fully dynamic in nature. However, it also shows that this hardness disappears when a tiny relaxation is made. The quality of clusters obtained is same as that while handling static data. This phenomenon is known as the "sandwich guarantee" of $\rho - approximate$ DBSCAN. The algorithms guarantee near-constant update processing. The approximate version takes $\mathcal{O}(N)$ (N is the data size) time while the unit spherical emptiness checking ($USEC$) method consumes $o(N)^{4/3}$ time in worst case. A factor which may go against this approach is the involvement of multiple theoretical concepts within it.

5. **DBCLASD:** DBCLASD (Distribution-Based Clustering of LArge Spatial Databases) [46] assumes that the objects within a cluster are distributed uniformly. The algorithm dynamically determines the quantity and conformation of clusters in a database without involving any input parameter. DBCLASD incrementally augments an initial cluster with points in its neighborhood. This procedure continues till the set of nearest neighbor distances of the resultant cluster fits the estimated distance distribution. A point which is not yet a part of the current cluster but needs to be examined for possible cluster membership is a candidate point. Candidates failing the cluster membership test in their first attempt are called unsuccessful candidates. Unsuccessful candidates are not overlooked. They are considered at a later time. Objects belonging to any cluster might shift to another cluster later. The running time of DBCLASD is approximately twice that of the DBSCAN.

2.2 Related density-based incremental outlier detection algorithms

1. **iLOF:** The incremental local outlier factor (iLOF) [47] algorithm facilitates dynamic updation of the properties of data points. The performance regarding outlier detection due to iLOF is equivalent to that of the LOF [37] algorithm handling static data. The performance comparison is made against insertion of each data record. The work also provides an evidence of the fact that only a handful amount of data points lying within the neighborhood of an inserted or deleted point gets affected. Therefore the overall approach remains independent of the total number of data points involved.

2. **I-IncLOF:** The Improved Incremental LOF (I-IncLOF) [48] algorithm takes into consideration a sliding window and allows the data properties a requisite updation within the window. The data items are then categorized as outliers or inliers. I-IncLOF fails to clearly define a scheme for performing deletion of data points efficiently.

3. **MiLOF:** The iLOF [47] algorithm limits itself to keep a check on the memory usage while storing old points. In order to address this issue, a memory efficient incremental outlier detection algorithm MiLOF [49] has been proposed. The memory efficient MiLOF has better efficiency than iLOF. Within the bounds of limited memory, the algorithm is able to detect outliers from data streams with high volume. However, the algorithm may be sensitive to parameter changes.

4. **DILOF:** The Density summarizing Incremental LOF [50] (DILOF) algorithm detects outliers within the limited bounds of memory. The algorithm is a two step process. The first step keeps a track of the neighbors of past data in the memory. The second step performs the summarization task by sampling previous data and keeping a record of their density. The underlying approach of this method is a density-based algorithm used for sampling old data records followed by detecting the outlier sequence. The algorithm assumes no previous knowledge of data distribution. DILOF may be sensitive to parameter changes.

5. **KELOS:** KELOS [51] (KDE-based Local Outliers over Streams) provides a scheme to detect the top-N KDE (Kernel Density Estimate) [51, 29] based local outliers from streaming data. KELOS was proposed to resolve the issues faced by algorithms having high time complexity and volatility while dealing with data updates. The algorithm introduced the concept of abstract kernel center (*a*KDE) [51] for accurately estimating the local data density. However KELOS employs a clustering scheme for deciding the abstract kernel which may not be a feasible option for voluminous datasets.

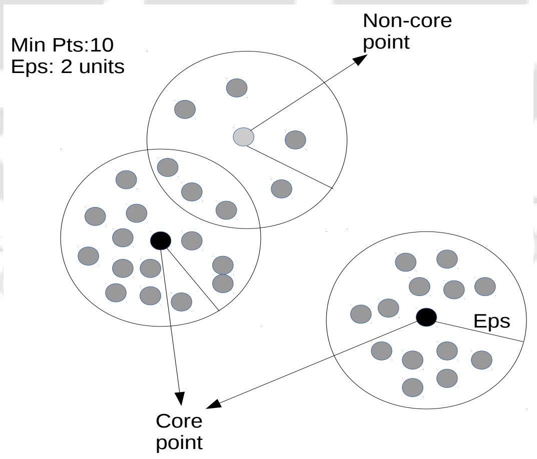
2.3 Other related naive algorithms

1. **DBSCAN:** Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Application of Noise (DBSCAN) [23] was proposed in 1996 as the first DBCLA. DBSCAN detects clusters in spatial databases

along with filtration of noise. DBSCAN takes two parameters viz. *Eps* and *Minpts*. For a given point p, *Eps* signifies the radius of its surrounding region known as the *Eps* neighborhood of p. The literature [23] denotes *Eps* neighborhood of p as $N_{Eps}\ (p)$. Let D denote the dataset, then $\forall p \in D$, its *Eps* neighborhood is given as $N_{Eps}\ (p) = \{q \in D \mid \text{dist}\ (p,q) \leq Eps\}$. If $|N_{Eps}(p)| \geq Minpts$, then p is a dense or core point otherwise p is a border (non-core) point. For a core point p w.r.t., *Eps* and *Minpts*, if there exists a point $q \in N_{Eps}\ (p)$, then q is directly density reachable from p. However q is density reachable from p w.r.t., *Eps* and *Minpts* only if there exists a chain of points $q_1,q_2,q_3,\ldots..,q_n$, where $q_1{=}p,q_n{=}q$ and $q_{i+1}[1 \leq i \leq n-1]$ is directly density reachable from q_i. Density-reachability is an extension of direct density-reachability. The relation follows transitivity but it is not symmetric.

DBSCAN classifies each point as either core or non-core (Figure 2.1). Two core points have the same cluster membership if they are directly density reachable. A point in cluster C is density reachable from any other core point belonging to C. The cluster expansion takes place by merging the density reachable core points. The algorithm assigns the border points to a cluster of their nearest core point. Points not belonging to any cluster qualify as noise points.

FIGURE 2.1: Cluster formation scheme in DBSCAN.



DBSCAN detects clusters of arbitrary shapes and sizes with appropriate noise filtering in spatial databases. However, the preset value of the parameter *Minpts* does not allow the algorithm to detect clusters of variable densities. The method also shows its limitations while dealing with high dimensional data sets. DBSCAN consumes $\mathcal{O}(N \log N)$ time, with N as the size of the dataset. In context of this thesis, DBSCAN is related to three of our primary contributions.

2. **Shared Nearest Neighbors clustering:** As per the Shared Nearest Neighbors (SNN) [52] clustering algorithm, two points p and q are said to be similar if they have sufficient number of common points between their corresponding KNN list. The set of shared points denotes the degree of similarity or proximity score between the involved pair of points. As per the SNN clustering algorithm, if p and q are significantly similar, then there exists a high probability that both points have the same cluster membership. The following illustration may explain the degree of similarity between p and q: Let KNN $(p) = \{q, n_1, n_2, n_3, n_4\}$ and KNN $(q) = \{p, n_1, n_2, n_3, n_5\}$, then KNN $(p) \cap$ KNN $(q) = \{n_1, n_2, n_3\}$. Therefore the degree of similarity between p and q is 3. However, the similarity measure is considered to be valid if the concerned points p, q are present in each others KNN list. Therefore, $q \in$ KNN (p) and $p \in$ KNN (q).

   A SNN graph is constructed by considering the data points as nodes and shared links between any pair of points as edges. The link strength provides a measure of the edge weight between any two involved nodes. If either of the points p or q are not present in each others' KNN lists, an edge is not formed between them. Post construction of the SNN graph, the connected components are designated as clusters.

   The SNN algorithm finds its usage in our second and third contributions respectively where we incrementally extend the SNN-DBSCAN [24] algorithm supporting batch mode updates.

3. **LOF:** Local Outlier Factor (LOF) [37] is an anomaly score associated with an instance of data. LOF value can be computed by taking the ratio of average local density of K nearest neighbors for any data point to the local density of the concerned point. The local density is evaluated as follows: first the radius of smallest hyper-sphere (distance to Kth nearest neighbor), the center of which lies with the point itself having its own KNN list is found. Then the value K is divided by the volume of the hyper-sphere. For a given point, higher the LOF score, greater is its degree of outlierness. The idea of finding local density is utilized in our final contribution where instead of relying on KNN, a kernel function is used.

2.4 Mixture of other related naive and incremental clustering algorithms

In order to cluster binary data, Mirkin and Kramarenko [53] proposed a model and algorithm related to bi-clustering boxes. The work was extended to tri-clustering of binary data objects. A combination of both bi and tri-clustering was additionally applied to real world datasets. Clustering may also be used for reducing the number of wrongly labelled items in a dataset. The said work [54] relies on the cluster membership criterion to assign appropriate labels of the entities within the dataset. For other multi-modal clustering algorithms adopting density based approaches, tri-concepts have been used for tri-clustering of binary data [55]. The proposed scheme generalized the method that had been previously introduced for concept based bi-clustering. It was also shown that finding an optimal tri-cluster cover is an NP-Complete problem [56].

Further more there are contributions made towards identifying various similarity measures involved within the cluster [57]. An efficient online tri-clustering algorithm [58] involving binary data have also been proposed to deal with big data. The algorithm runs in linear time and space complexity. A convenient parallelization of the said scheme is also possible resulting in analysis of big datasets. Algorithms generating formal concepts and graphs related to concept lattices [59] have also been studied and compared within the domain of incremental clustering approaches. In addition, an incremental algorithm [60] for construction of lattice have been proposed which is related to methods [61, 62] identified previously for building concept lattices. The constructed lattice is derived by closing the group of sets with reference to intersection of sets.

Another relevant paradigm for dense and noise points formalism is the rough clustering. Conventionally, the data points which are at identical distances from multiple cluster centers are placed into a single cluster. With rough set clustering [63], the idea of overlapping clusters is also taken into consideration.

2.5 Background

In this section we present a brief study of necessary algorithms that form the basis of our contributions in this thesis. First, we present the MBSCAN [2] clustering algorithm that has been incrementally extended to support point-based insertion (first contribution). Next we provide the details of clustering scheme followed by SNN-DBSCAN [24]. This relates to our proposed dynamic insertion and deletion algorithms in batch mode (second and third contribution). Finally, we highlight the KNNOD [5] algorithm, which is compared with our proposed incremental outlier detection approach (fourth contribution) facilitating single point insertion.

2.5.1 MBSCAN

MBSCAN [2] is the first DBCLA to introduce a generic interpretation of dissimilarity based on data distribution. The algorithm proves that the data dependent dissimilarity is better than any distance based approaches used for clustering. MBSCAN replaces the distance function with a probability measure. The dissimilarity estimate depends on the probability mass of the smallest region covering two data instances. Formally, let D be the dataset from a probability density function F and $H \in \mathcal{H}(D)$ be a hierarchical partitioning model of the space into non-overlapping regions. For two data instances a and b, the smallest local region r (Equation 2.1) covering a, b w.r.t., H and D is given as:

$$R(a, b|H; D) = \underset{r \subset H | \{a,b\} \in r}{\operatorname{argmin}} \sum_{z \in D}^{n} 1(z \in r) \tag{2.1}$$

where 1 (.) is an indicator function and n represents the strength of smallest local region. The mass based dissimilarity or probability mean mass estimated from a finite number ($t \in \mathbf{Z}^+$) of models $H_i \in \mathcal{H}(D)$, $i = 1, 2, 3, \ldots, t$ is given as:

$$m_e(a, b|D) = \frac{1}{t} \sum_{i=1}^{t} P(R(a, b|H_i; D)) \tag{2.2}$$

where $P(R(a, b|H_i; D) = \frac{1}{|D|} \sum_{z \in D} 1\ (z \in R)$ represents the probability mass w.r.t., a given hierarchical model and $|D|$ being the size of dataset. Instead of any conventional distance metric to measure the proximity, the dissimilarity between two points m_e (a,b) is measured by using Equation 2.2. Unlike DBSCAN, instead of using ϵ-neighborhood, μ-neighborhood is used for estimating the neighborhood strength or mass based point density. The definition of μ-neighborhood mass is given as:

$$M_\mu(a) = \#\{b \in D | m_e(a,b) \leq \mu\} \tag{2.3}$$

From Equation 2.3, it is evident that only those points b are included as seed points for a, if the mass based dissimilarity between a and b is less than a certain threshold μ. Therefore, unlike DBSCAN which involves a radius to decide the neighborhood while finding point density, MBSCAN relies on a probability measure. The algorithm runs to completion in $\mathcal{O}(N \log N)$ time.

Key advantage (s) of MBSCAN: The algorithm introduces a generic data dependent dissimilarity measure to find proximity between data objects. Moreover, the use of μ- neighborhood [2] enables MBSCAN to address the hard density problem [2, 23] and the associated challenges related to distance based clustering [2, 24].

2.5.2 SNN-DBSCAN

The SNN-DBSCAN [24] clustering algorithm is an amalgamation of two clustering techniques viz. SNN (Shared nearest neighbor) [52] (Refer Section 2.3) and DBSCAN [23] (Refer Section 2.3). In case of SNN [52] algorithm, a SNN-graph is constructed with data points as nodes and the shared links as edges. The connected components within the SNN graph are treated as clusters.

However, the scheme adopted by SNN-DBSCAN slightly differs from that of the SNN clustering algorithm. As per the new scheme, if for any pair of points (p, q), their shared link strength SNN (p, q) (Refer SNN algorithm in Section 2.3) is greater than a certain threshold δ_{sim}, the link is said to be a strong link, otherwise a weak link. The strong links in the SNN graph are retained as edges while the weak links are removed. The residual graph that is obtained in this process is known as the K-Nearest Neighbor Sparsified SNN [24] or K-SNN graph. Unlike the SNN algorithm, the connected components from the K-SNN graph are not considered as clusters. Instead the cluster formation policy is similar to that of the DBSCAN algorithm.

Data points which have a sufficient number of adjacent strong links are designated as core (dense) points while rest of the points are non-core (non-dense). Two core points connected by a shared strong link obtain the same cluster membership. Non-core points are assigned to a cluster of their nearest core point. Points which fail to obtain any cluster membership qualify as noise points. The algorithm runs to completion in $\mathcal{O}(N^2)$ time. The construction of SNN graph in the form of a similarity matrix results in its quadratic time complexity.

Key advantages of SNN-DBSCAN: Uses the concept of SNN to evaluate proximity score between data objects. This removes the hazards associated with distance based clustering in a high dimensional space. SNN-DBSCAN is also able to find clusters of varying shapes and densities.

2.5.3 KNNOD

K-Nearest Neighbors Outlier Detection [5] (KNNOD) algorithm relies on the measure of distance to extract outliers. For each point $x \in D$ (base dataset), the algorithm identifies the distance of x with its Kth nearest neighbor d_{K_x}. If $d_{K_{th}}$ is considered as a threshold value, then all the data points whose d_{K_x} value is greater than $d_{K_{th}}$ are considered as outliers while rest of the points remain as inliers.

Key advantages of KNNOD: No prior assumptions about data distribution is required.

Chapter 3

iMass: An Approximate Incremental Clustering Algorithm using Probability Measure

The previous chapter provided with a study of related incremental algorithms in the field of density based clustering and outlier detection. In course of our coverage of necessary literature, we observed that the Background (Section 2.5) comprised of a robust clustering algorithm known as MBSCAN [2]. MBSCAN introduced a data dependent dissimilarity measure in order to find the proximity between data points. This tends to overcome the limitations associated with distance based clustering [24, 64]. To the best of our observation, no incremental extension exists for MBSCAN. Realizing the importance of MBSCAN in finding clusters of arbitrary shapes and densities, in this chapter we propose a novel approximate incremental extension to MBSCAN known as the *iMass* (Incremental Mass Based Clustering).

3.1 Motivation

MBSCAN relies on a concept called probability mass [2] instead of any geometric model to compute the dissimilarity between any pair of data points. For any given pair of points (x, y), instead of measuring the distance $|x - y|$ along a dimension, MBSCAN measures the pairwise dissimilarity by finding the mass of smallest local region covering x and y. The mass of a region is determined by the number of points within that region. The algorithm maps a region in the form of a hierarchical partitioning model (isolation-Tree or *iTree* [65]). Multiple such hierarchical models are considered to build an isolation Forest (*iForest* [65]). The expectation of probability masses obtained from each model is treated as the dissimilarity value between x and y. Each pairwise dissimilarity is stored in a structure called mass-based dissimilarity matrix or mass-matrix. By determining the dissimilarity measure based on distribution of data, MBSCAN removes the weaknesses associated with distance based clustering [24, 64, 2].

Table 3.1: Motivation behind developing the *iMass* clustering algorithm.

Motivation	**Description**
Redundant computation	Naive or non-incremental algorithms fail to address the issue of redundant computation while handling dynamic datasets. They involve the entire set of data points against every new update.
Small frequent Updates	When minimal number of insertions are made to a larger base dataset, the changes in clustering are also expected to be small. As a result, there is a need for designing intelligent algorithms to handle such frequent updates efficiently without redundant computation.
MBSCAN is unable to handle dynamic insertion	MBSCAN is a naive clustering algorithm. Due to addition of data points one at a time, the cluster extraction process may get slower as the size of base dataset increases. As a result there is a need to process updates intelligently to quicken the cluster detection process.

However, MBSCAN fails to detect clusters dynamically upon insertion of new data points. To the best of our observation, no incremental version of MBSCAN exists. Therefore in order to extract clusters efficiently post new updates, we provide an approximate incremental extension to the MBSCAN clustering algorithm known as *iMass*. Through *iMass*, we strive to obtain a nearly accurate set of clusters at the benefit of a highly efficient point insertion technique. Table 3.1 provides a brief description about the motivation behind our work.

3.1.1 Chapter contributions

The key contribution(s) made in this chapter may be summarized as follows:

1. Identify the most expensive components of the MBSCAN algorithm.

2. Propose an incremental clustering algorithm *iMass* by targeting the expensive components of MBSCAN.

3. Mathematically deduce a relation to compute the dissimilarity score between any old pair of points for $iMass$ after a new insertion has been made.

4. Propose Lemmas based on the $iMass$ clustering algorithm. The is done in order to depict the properties of algorithmic components when updates are made to the dataset.

3.2 Related work and background

In this section, we provide a brief description about the algorithms and concepts that prelude $iMass$.

DBSCAN [23] is the pioneer density based clustering algorithm proposed by Ester $et'al$. The algorithm enables clustering of spatial data with appropriate noise filtering. DBSCAN employs ϵ-neighborhood density estimation to compute the point densities. The term ϵ or Eps determines the neighborhood radius of a given point $x \in D$ (dataset). Let $N_\epsilon(x)$ denote the set of points within the ϵ neighborhood of x, and $MinPts$ be the number of points belonging to $N_\epsilon(x)$, then according to DBSCAN we have the following interpretations:

- If $|N_\epsilon(x)| > MipPts$, then x is a core or dense point ($x \in Core(D)$).
- If $|N_\epsilon(x)| \leq MipPts$, then x is a non-core or non-dense point ($x \in Non - Core(D)$).
- $\forall\, x, y \in D$, if $x \in Core(D), y \in Core(D)$ and $y \in N_\epsilon(x), x \in N_\epsilon(y)$, the points x and y obtain the same cluster membership.
- $\forall\, x, y \in D$, if $x \in Core(D), y \in Non - Core(D)$ and $y \in N_\epsilon(x), x \in N_\epsilon(y)$, the points x and y obtain the same cluster membership.
- $\forall\, x, y \in D$, if $x \in Core(D), y \in Non - Core(D)$ and $y \notin N_\epsilon(x)$, then if $\exists z \in Core(D)$ and $y \in N_\epsilon(z)$, the points z and y obtain the same cluster membership.
- $\forall\, x \in D$, if $x \in Non - Core(D)$ and $\nexists z \in Core(D)$ where $z \in N_\epsilon(x)$, then x qualifies to be a noise point.

However, due to prior fixation of parameters $MinPts$ and ϵ, it becomes improbable for DBSCAN to find clusters of varying densities. Moreover, the algorithm manages to extract only clusters of globular shapes in lower dimensions. In order to address these limitations associated with distance based clustering technique like DBSCAN [23], a mass-based dissimilarity oriented algorithm viz. MBSCAN was proposed. MBSCAN acts as a building block for its incremental version $iMass$. The next set of related works presents concepts that lead to the use of data dependent dissimilarity measure in MBSCAN as well as the $iMass$ algorithm.

Existing studies [66, 67] have pointed out the issues over use of geometric model for computing dissimilarity between data points. It has been indicated that on the basis of human perception, two instances in a dense region are less similar than two instances of identical geometric distance in a region of lesser density [66]. The pioneer data dependent dissimilarity measure [68] called as m_p- dissimilarity provided a better match in identifying the nearest neighbors for K-nearest neighbors classifiers. According to this measure, the dissimilarity in i^{th} dimension for any pair of points is computed by finding the probability mass of a region $P(R_i(x, y))$ instead of using the distance metric $|x_i - y_i|$. The m_p-dissimilarity [68] approach investigates about the ways of exploiting the data distribution for finding dissimilarity between two data items. The said scheme computes the proximity between two objects in each dimension as a probability mass in a region enclosing the objects. The dissimilarity of a pair of data objects can be computed in $\mathcal{O}(d \log n)$ time where d represents the number of dimensions.

In another study related to image retreival [69], a novel dissimilarity measurement technique was proposed which can calculate both the distance and perceptual similarity of two images in a multi-dimensional space. It combines the properties of both m_p [68] and l_p [68] dissimilarity having a $\mathcal{O}(rd)$ time with d being the number of dimensions and r is the number of points in a given dimension used for finding the mass of any pair(x, y).

MBSCAN [2] introduced the generic implementation of data dependent dissimilarity in which the m_p-dissimilarity is a special case. While proposing $iMass$ algorithm, the concept of dissimilarity laid by MBSCAN is used as a measure to calculate the proximity score between any pair of points in the updated dataset D' post new insertions. The algorithm replaces the density definition of DBSCAN [23]

Table 3.2: Major notations used in this chapter (first contribution).

Notation	Description
C	Set of Clusters
C'	Set of Clusters after dataset is updated
D	Original (Base) dataset
D'	Changed dataset after a point insertion
n	size of base dataset
k	number of new insertions
δ_{core}	core point formation threshold
$\mathcal{P}(.)$	Power set
$Core\ (.)$	Set of core points of dataset
$Non\text{-}Core\ (.)$	Set of non-core points of dataset
$\|.\|$	Size of a set
$R(.)$	Region of space
$P(R(.))$	Probability mass of the region
$\mathcal{H}(.)$	Set of hierarchical models for partitioning a region
H	A particular hierarchical model

with a mass-based neighborhood or μ-neighborhood mass definition[1] (Refer Section 3.3) while retaining the clustering scheme of DBSCAN. The μ-neighborhood mass of any point x viz. $M_u(x)$ [2] relies on a certain value μ determined by MBSCAN in course of its execution. Similar to ϵ parameter in DBSCAN, μ controls the size of a region around x. Denser regions are small while sparse regions are large. The use of μ-neighborhood mass enables MBSCAN to extract clusters of arbitrary densities and shapes unlike DBSCAN.

3.3 Preliminaries and Definitions

In this section, we present the definitions of different terms and concepts that are used in this chapter (See Table 3.2 for identifying the notations henceforth).

Let D be a sample of data from probability density function F; and let $H \in \mathcal{H}$ be a hierarchical model that partitions the data space into non-empty regions, then the following concepts may be defined as follows:

[1]See details in Section 3.5 for explanation of mass-based neighborhood or μ-neighborhood mass as a part of the MBSCAN algorithm.

3.3.1 Modeling a region

A recursive partitioning methodology known as $iForest$ (isolation Forest) [65] is used to depict regions. Existing study [70] has shown that $iForest$ is a special case of mass estimation technique. MBSCAN [2] uses a method based on completely random trees to construct an $iTree$ (isolation Tree) (Refer Section 3.5 for details). An $iForest$ is a combination of multiple such $iTrees$. Each $iTree$ is a binary tree that represents a particular hierarchical partitioning model $H_i, i = 1, 2, 3..., t$ where t denotes the maximum number of $iTrees$.

Let R represent a region, then we have the following interpretations:

- $iTree_i, i = 1, 2, 3, \ldots, t$ models a sub-region $r_j \subset R, j = 1, 2, 3, \ldots, t$
- $\bigcup_{j=1}^{t} r_j = R; r_j \neq \phi, j = 1, 2, 3, ..., t$
- $\forall\, i, j, i \neq j, r_i \cap r_j = \phi$ where $1 \leq i, j \leq t$.
- If the number of points within any $r_j, j = 1, 2, 3, \ldots, t$ belong to a set $\mathcal{D}$, then the root node of corresponding $iTree_j, j = 1, 2, 3, \ldots, t$ may contain the elements of $\mathcal{D}$ and based on certain split condition (See details in Section 3.5), the internal nodes are created.
- The root node of any $iTree_j, j = 1, 2, 3, \ldots, t$ effectively represents the whole sub-region $r_j, j = 1, 2, 3, \ldots, t$ and the internal nodes denote r_j's division into smaller sub-regions.

3.3.2 Mass of a region

The mass of a region is defined as the number of data points within that region. The following relation (Equation 3.1) defines the mass of a region containing a and b $\forall\, a, b \in D$:

$$M_r(a, b|H; D) = \sum_{r \subseteq H \ s.t. \{a,b\} \in r,\ c \in D} 1(c \in r) \tag{3.1}$$

where $1(.)$ is an indicator function, r is any region, H is any hierarchical partitioning model represented by an $iTree$, D is set of elements involved. If any node of $iTree$ modeling the region r represents a sub-region within r containing data points a and b, then the number of elements within that node gives the mass of that sub-region inclusive of the pair of points. The number of elements in the root node of the *iTree* provides the mass of the whole region r.

3.3.3 Mass of smallest local region

We have the following relation (Equation 3.2) defining the mass of smallest local region [2] containing points a and b $\forall\, a, b \in D$:

$$R(a,b|H;D) = argmin_{r \subset H\ s.t. \{a,b\} \in r} \sum_{c \in D} 1(c \in r) \tag{3.2}$$

where 1 (.) is an indicator function, r is the smallest local region, H is any hierarchical partitioning model represented by an *iTree*. The smallest local region covering a, b is represented by the lowest leveled node of the *iTree* containing the same pair of points. The mass of smallest local region r is the number of elements in the lowest leveled node inclusive of the pair of points a and b.

3.3.4 Mass-based dissimilarity

Mass based dissimilarity [2] or mass or probability mean mass of a and b w.r.t., D and F is defined as the expected probability of $R\ (a,b \mid H;D)$ (Equation 3.3) and is given as:

$$m_e(a,b|H;D) = E_{\mathcal{H}(D)}[P_F(R(a,b|H;D))] \tag{3.3}$$

where P_F (.) is the probability w.r.t., F and $E_{\mathcal{H}(D)}$ is the expectation taken over all hierarchical models in $\mathcal{H}$ (D). In practice the mass based dissimilarity would be estimated from a finite number of hierarchical models (*iTrees*) $H_i \in \mathcal{H}$ (D), $i = 1, 2, 3...., t$ as follows (Equation 3.4):

$$m_e(a,b|H;D) = \frac{1}{t} \sum_{i=1}^{t} \tilde{P}(R(a,b|H_i;D)) \tag{3.4}$$

where $\tilde{P}$ (R) = $\frac{R(a,b|H;D)}{|D|}$ denotes the probability mass w.r.t., a given H_i. It is to be noted that R $(a,b \mid H;D)$ is the mass of smallest local region covering a and b. It is analogous to the shortest distance between a and b used in the geometric model.

3.3.5 Mass-based neighborhood

For a real value μ (determined by the MBSCAN algorithm), the mass-based neighborhood or μ-neighborhood mass [2] for a point $a \in D$ is given as:

$$M_u(a) = |\{b \in D | m_e(a,b) \leq \mu\}| \tag{3.5}$$

Equation 3.5 states that the size of μ-neighborhood mass for a point a is the number of points with which its probability mean-mass is less than or equal to μ.

3.3.6 Clustering

Given a dataset D, $\forall\, a, b \in D$ if there exists a mass-based dissimilarity function $m_e(a,b)$ and a mass-based neighborhood set $M_\mu(a)$, then we define clustering by a mapping $f : D \to C$, where $C \subseteq \mathcal{P}\ (D)$. If $a \neq b$ and there exists a threshold δ_{core}, then we may have the following interpretations:

1. If $|M_u(a)| > \delta_{core}$ and $|M_u(b)| > \delta_{core}$, where $b \in M_u(a)$ and $a \in M_u(b)$, then $f(a) = f(b)$.
2. If $|M_u(a)| > \delta_{core}$ and $|M_u(b)| <= \delta_{core}$, where $b \in M_u(a)$ and $a \in M_u(b)$, and $\exists c \in D$ where $a \neq b \neq c$, $b \in M_u(c)$ and $|M_u(c)| > \delta_{core}$. Then if $m_e(a,b) < m_e(b,c)$, then $f(b) = f(a)$ else $f(b) = f(c)$.
3. If $|M_u(a)| <= \delta_{core}$, and $\nexists c \in D$ where $a \neq b \neq c$ and $|M_u(c)| > \delta_{core}$, then $\{a\} \notin C$.

According to the above definitions, the first point states that if two data points a and b are dense or core[2], and they both belong to each other's mass-based neighborhood (μ-neighborhood), then they are a part of the same cluster.

The second point states that if two data points a and b are a part of each other's mass based neighborhood such that a is core while b is non-core (non-dense), then b is associated with a cluster of its nearest core point.

The third point states that if any non-core point eg: a fails to find any core point within its mass-based neighborhood, it does not obtain any cluster membership.

[2]The detailed explanation of the core or non-core points is presented in Section 3.5 as part of the MBSCAN algorithm.

3.3.7 Approximate Incremental Clustering

Let the initial clustering defined by a mapping $f : D \rightarrow C$ represents the set of clusters obtained from the static algorithm. Let an insertion sequence of k points be made over a base dataset D ($|D| = n, k \ll n$). After k insertions let D' be the updated dataset, and an incremental clustering be defined as a mapping $h : D' \rightarrow C'$, where $C' \subseteq \mathcal{P}\ (D')$ represents the clusters produced from the incremental version. Now, if the updated dataset D' is fed to the naive algorithm in its entirety and the clustering is given by a mapping $f : D' \rightarrow C''$, then in case of approximate incremental clustering we have $C' \approx C''$.

3.3.8 Core and Non-core points

For any point $a \in D$, if the size of μ-neighborhood mass $M_u(a)$ exceeds a core point formation threshold δ_{core}, then a is designated as a core point or else it is a non-core point.

3.3.9 Noise points

For a non-core point $a \in D$, if it fails to obtain any cluster membership, then that point qualifies as a noise point.

3.4 Problem formulation

For k number of insertions where k $\in \mathbf{N}$, $\mathcal{R}^d$, let T_{naive} be the total time taken by the naive algorithm, $T_{point-ins}$ be the time taken by the point based approximate incremental method against every insertion. Let C_{naive} and $C_{point-ins}$ be the respective set of clusters obtained after k number of updates and $|Mem_{point-ins} - Mem_{naive}|$ be the difference in percentage of average memory consumed, then we establish the following objectives:

- $T_{point-ins} < T_{naive}$
- $C_{point-ins} \approx C_{naive}$
- $|Mem_{point-ins} - Mem_{naive}| \leq \delta$, where δ is a small real number.

In the next section, we present a detailed description of the MBSCAN [2] clustering algorithm and identify its most expensive components so that we can target these components incrementally while designing the incremental $iMass$ algorithm.

3.5 The MBSCAN Clustering Algorithm

The components of MBSCAN [2] algorithm are: base dataset (D), $iForest$, mass-matrix, core and non-core points, clusters and outliers. The MBSCAN clustering algorithm constitutes of the following steps:

1. **Step 1 - Construction of $iForest$ to model a region:** Isolation Forest or $iForest$ is used by MBSCAN to model a region. An $iForest$ is a combination of multiple number of *iTrees* (isolation Trees). Each *iTree* represents a particular hierarchical partitioning model $H_i, i = 1, 2, 3, ..., t$ of a sub-region, where t is the maximum number of chosen *iTrees*.

   Individual *iTree* construction method: A random tree construction policy is used to build a single *iTree*. Individual *iTree* is constructed using the axis-parallel split [2] technique as follows:

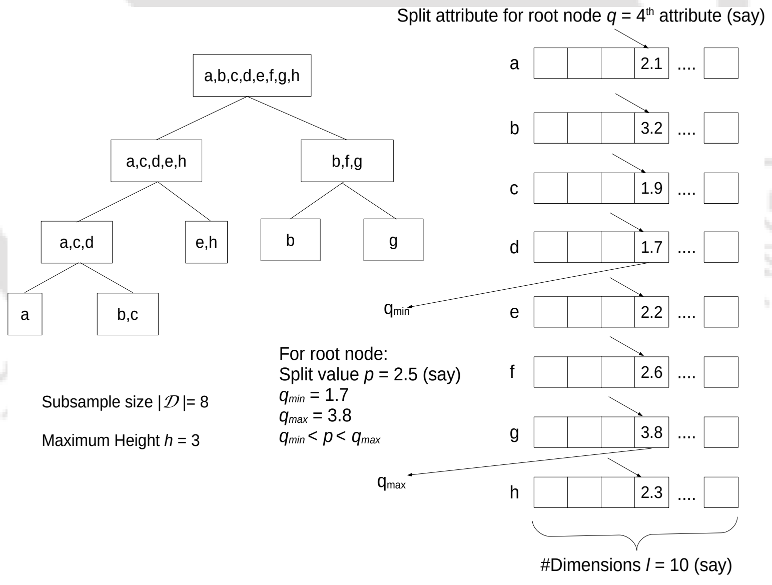
   ALGORITHM 1: *iTree($\mathcal{D}$,e,h)*

```
1  Input: $\mathcal{D}$-input data; $e$-current height of $iTree$; $h$-maximum height of $iTree$;
2  Output: $iTree$;
3  if $e \geq h$ $OR$ $|\mathcal{D}| \leq 1$ then
4      $exNode\{\text{Size} \leftarrow |\mathcal{D}|\}$;
5  else
6      Randomly select an attribute $q$;
7      Randomly select a split point $p$ between $min$ and $max$ values of attribute
         $q$ in $\mathcal{D}$.;
8      $\mathcal{D}_l \leftarrow filter(\mathcal{D}, q < p), \mathcal{D}_r \leftarrow filter(\mathcal{D}, q \geq p)$;
9      return $inNode$
10     $Left \leftarrow iTree(\mathcal{D}_l, e + 1, h)$,
11     $Right \leftarrow iTree(\mathcal{D}_r, e + 1, h)$,
12     $SplitAttr \leftarrow q, SplitValue \leftarrow p$
13 end
```

   A subset $\mathcal{D} \subset D$ where $|\mathcal{D}| = \Psi$ is sampled without replacement from D to build an *iTree* independently. The maximum height attained by an *iTree* is $h = \lceil \log_2 \Psi \rceil$. The parameter e representing the initial height is set to 0 while the tree building process starts. Algorithm 1 presents the axis-parallel split procedure [2] for recursively constructing an *iTree*.

Description of Algorithm 1: The root node of an *iTree* consists of the data elements in $\mathcal{D} \subset D$ (whole base dataset) where each $x \in \mathcal{D}$ is a d-dimensional point. As per the tree construction algorithm, a split attribute (say q^{th} attribute, where $1 \leq q \leq d$) is randomly selected [Line 6]. Now $\forall\, x \in \mathcal{D}$, the q^{th} attribute values are compared and subsequently the minimum (q_{min}) and maximum (q_{max}) q^{th} attribute values are found [Line 7]. A random split value p is chosen such that $q_{min} < p < q_{max}$ [Line 7]. If the q^{th} attribute value for any $x \in \mathcal{D}$ (for root node) is less than p, then x becomes a part of the left child ($\mathcal{D}_l$) otherwise it goes to right child ($\mathcal{D}_r$) [Line 8]. This procedure is performed recursively for the newly created internal nodes ($\mathcal{D}_l$,$\mathcal{D}_r$) [Line 10,11]. The *iTree* construction continues till the maximum height is reached or each point gets isolated [Line 3].

Figure 3.1: *iTree* construction procedure based on axis-parallel split algorithm [2].



Running example: Through Figure 3.1 we demonstrate the *iTree* building process. Let us consider that the subsample set $\mathcal{D} \subset D$ comprises of eight elements such that $\mathcal{D} = \{a, b, c, d, e, f, g, h\}$ and hence $\Psi = 8$. Initially, the elements in $\mathcal{D}$ represent the elements of root node of an *iTree*. Let us consider that each of the points in $\mathcal{D}$ are having l number of attributes (say $l = 10$).

Focusing on the root node first, we assume that the axis-parallel split algorithm (Algorithm 1) for construction of an $iTree$ randomly selects the 4^{th} dimension as its split attribute. As a result, the value of random split attribute q for root node is 4. From Figure 3.1, we observe that on comparing the 4^{th} attribute values of all the eight data points within root node, points d and g have the minimum and maximum 4^{th} attribute value. Therefore we have q_{min}=1.7 and q_{max}=3.8 respectively.

The next task is to select a random split value p lying between q_{min} and q_{max}. Assuming that p takes a value of 2.5 (say), the algorithm compares the 4^{th} attribute value (since $q = 4$) of individual data points with p. If the corresponding 4^{th} attribute value of a data point in $\mathcal{D}$ is less than 2.5, then that point becomes a part of the left child ($\mathcal{D}_l$), otherwise it goes to the right child ($\mathcal{D}_r$) of the $iTree$. From Figure 3.1, it is evident that the newly created nodes: ($\mathcal{D}_l$) and ($\mathcal{D}_r$) consists of a fraction of elements that were present in the root node due to the split. Based on the split criterion, the left child $\mathcal{D}_l$ contains $\{a, c, d, e, h\}$ while $\mathcal{D}_r$ comprises of $\{b, f, g\}$. The whole procedure is executed for splitting the nodes $\mathcal{D}_l$ and $\mathcal{D}_r$ in a recursive manner. The split attribute (q) and split value (p) selection for bifurcating the internal nodes are performed randomly. Two nodes of an $iTree$ may not have same q and p values. The $iTree$ building process continues till it either reaches a maximum height of 3 ($\because \Psi = 8, h = \lceil \log_2 \Psi \rceil$) or all the elements in $\mathcal{D}$ are isolated.

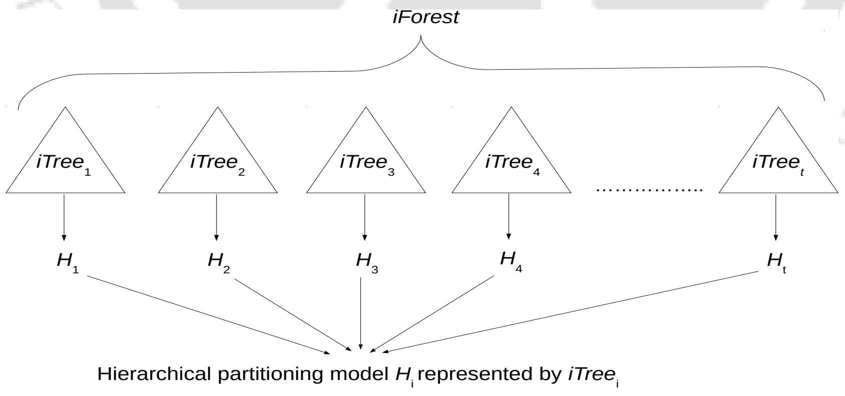


FIGURE 3.2: $iForest$ as a collection of t $iTrees$ [2].

Every $iTree$ represents a hierarchical partitioning model $H_i, i = 1, 2, 3, ..., t$ of a sub-region. In a similar manner, t number of $iTrees$ are constructed to build the $iForest$ for modeling a whole region. Figure 3.2 represents the $iForest$ as a combination of t such $iTrees$ each of which are built by the axis-parallel split tree construction algorithm (Algorithm 1).

Post construction of the $iForest$, every data object in D is positioned into the respective nodes of an $iTree$ as per the q, p values (node-split criterion). This enables to compute the mass of a node containing any pair of points.

2. **Step 2 - Construct the mass-based dissimilarity matrix:** The mass-based dissimilarity matrix or mass-matrix provides a measure of dissimilarity between each pair of data points belonging to D. The mass-matrix is constructed by computing the probability mean mass between all the pair of points. The following list is adhered to while computing the mass-matrix:

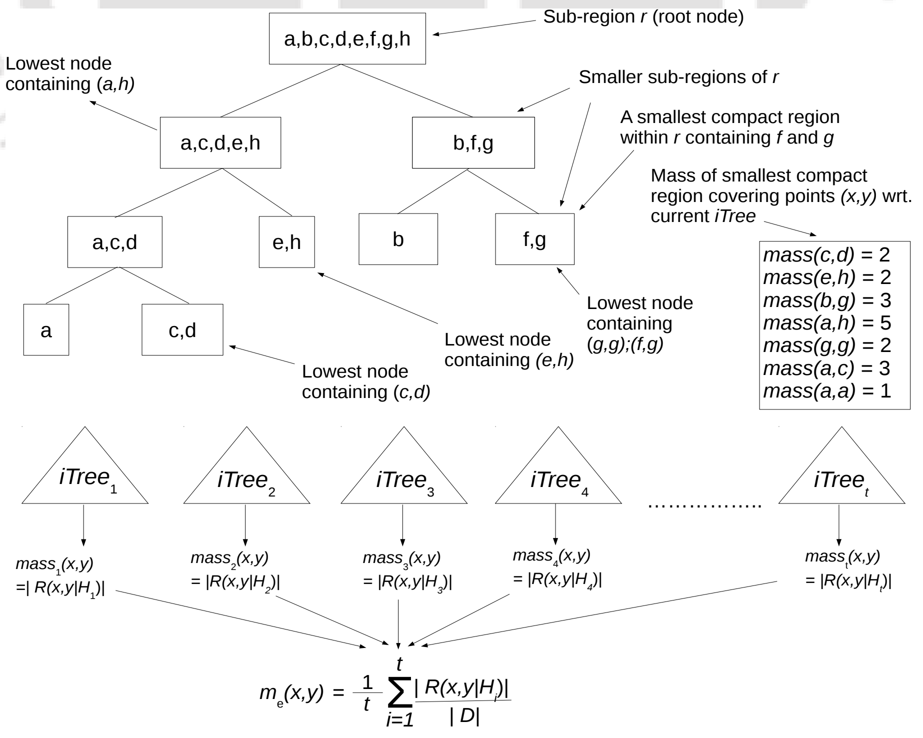
   (a) Select test points $x, y \in D$ (base dataset).

   (b) $\forall\ iTree_i, i = 1, 2, 3, ..., t$, identify the lowest leveled node containing x, y. The lowest leveled node represents the smallest compact region covering the pair of points (x, y).

   (c) If $|R(x, y|H_i)|$ denotes the mass of the lowest leveled node from $iTree_i$ representing hierarchical model H_i, then the probability mass w.r.t., $iTree_i$ is given as $\frac{|R(x,y|H_i)|}{|D|}, i = 1, 2, 3, ..., t$.

   (d) $\forall\ x, y \in D$, the probability mean mass across all $iTrees$ is given as: $m_e(x, y) = \frac{1}{t}\sum_{i=1}^{t} \frac{|R(x,y|H_i)|}{|D|}$.

FIGURE 3.3: $iForest$ as a collection of t number of $iTrees$.



Running example: Let us consider the subsample $\mathcal{D} = \{a, b, c, d, e, f, g, h\}$ where $\mathcal{D} \subset D$ (Figure 3.3). From the *iTree* present in the figure, we observe that the mass of smallest local region containing any pair of points is the total count of data points present in the lowest leveled node inclusive of the pair of points. Eg: if we consider the pair of points (c, d) (Figure 3.3), both the points are present in four nodes (including root node) of the *iTree*. Out of four nodes, the node at (*Level* 3, Node 9) [Node numbering convention: If a node is numbered as m, it's left child is numbered as $2m$ while the right child's node number is $2m + 1$. We assume that the root node of the *iTree* in Figure 3.3 is numbered as 1 having *Level* 0.] also contains (c, d) beyond which the points are not found. As a result the mass of the smallest local region containing the pair of points (c, d) is 2. For leaf nodes containing only a single point eg: a (*Level* 3, Node 8), the mass is taken to be 1. However, for point g, the self-dissimilarity score is 2. This is because the lowest leveled node (*Level* 2, Node 7) containing g has only two elements f, g within it. For the pair of points $(b, g), (a, c)$, the mass of lowest leveled node is 3 as there are three data points present in that node. Similarly, the mass of lowest leveled node for the pair (a, h) is 5 (*Level* 1, Node 2).

The subsample $\mathcal{D} \subset D$ for individual *iTree* will be different. However, if we position the whole dataset D in an *iTree* already constructed according to its node split-criterion, we shall obtain the probability mass w.r.t., that *iTree* for any given pair of points $(x, y) \in D$. For any *iTree*, the mass of lowest leveled node (lowest node)[3] containing pair of points (x, y) divided by the size of the whole dataset D gives a measure of its probability mass w.r.t., that *iTree*. The summation of individual probability masses obtained $\forall\, iTree_i, i = 1, 2, 3, ..., t$ divided by t (the total number of *iTrees*) gives the probability mean mass or dissimilarity score between x and y (Figure 3.3). In this way the mass-matrix $\forall\, x, y \in D$ is constructed (Figure 3.4).

3. **Step 3 - Find the μ-neighborhood mass $M_u(a)$:** The mass-based neighborhood or μ-neighborhood mass (Equation 3.5) for any point $a \in D$ is equivalent to the number of other points b with which a's probability mean mass is less than a real value μ. The value of μ is chosen such that it is greater than the maximum self dissimilarity score between any pair of points. The self dissimilarity score $\forall\, a \in D$ or $m_e(a, a)$ can be identified from the diagonal elements of the mass-matrix (Figure 3.4). The value of μ decides the size

[3]We use the terms 'lowest leveled node' or 'lowest node' interchangeably.

FIGURE 3.4: Mass based dissimilarity matrix.

$x \backslash y$	1	2	3	4	.....	n
1	$m_e(1,1)$	$m_e(1,2)$	$m_e(1,3)$	$m_e(1,4)$	.....	$m_e(1,n)$
2	$m_e(2,1)$	$m_e(2,2)$	$m_e(2,3)$	$m_e(2,4)$	.....	$m_e(2,n)$
3	$m_e(3,1)$	$m_e(3,2)$	$m_e(3,3)$	$m_e(3,4)$	.....	$m_e(3,n)$
4	$m_e(4,1)$	$m_e(4,2)$	$m_e(4,3)$	$m_e(4,4)$	.....	$m_e(4,n)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
n	$m_e(n,1)$	$m_e(n,2)$	$m_e(n,3)$	$m_e(n,4)$	.....	$m_e(n,n)$

of the mass-based neighborhood for any point. The μ-neighborhood mass replaces the distance based definition of finding the point density.

4. **Step 4 - Core and Non-core points:** Let δ_{core} be the core point formation threshold, then for any point x if $|M_u(x)| > \delta_{core}$, then x is designated as a core point, otherwise a non-core point.

5. **Step 5 - Cluster formation and noise points:** Two core points within the same mass-based neighborhood aggregate to form a cluster. A non-core point attaches itself to a cluster of its nearest[4] core point. Any non-core point which fails to obtain a cluster membership is classified as a noise point.

3.6 Experimental evaluation of MBSCAN in brief

In order to find out the cost incurred by the MBSCAN algorithm, we conducted experiments on some real world and synthetic datasets taken from the UCI Machine Learning repository [3] and clustering benchmark datasets site [4]. A brief description of the datasets is provided in Table 3.3. Datasets S1 and S2 are synthetic in nature while rest of them are real world.

Table 3.4 shows the results that were obtained on executing the MBSCAN clustering algorithm on datasets (Table 3.3) used in the experiments. We observed that out of all the constituent components of MBSCAN, the construction of mass-matrix consumes a greater share of the total time (Table 3.5). Noticeably, the

[4]For a point, its nearest object is the one with which it has lesser dissimilarity score.

Table 3.3: Datasets used for evaluating the MBSCAN algorithm [3, 4].

Dataset	Size	Dimensions	Description
Libras	360	91	Hand movement type
Segment	2310	19	Image segmentation data
Wine	178	14	Chemical analysis of wine
Seeds	210	7	Varieties of wheat
Aggregation	788	2	Clustering Aggregation
Iris	150	4	Iris plants database
S1	900	2	Synthetic data (Gaussian Clusters)
S2	1500	2	Synthetic data (Gaussian Clusters)

Table 3.4: Results of MBSCAN on various datasets.

Dataset	*#iTrees* t	δ_{core}	μ	***iForest*** **built time (sec)**	**Mass-matrix built time (sec)**	**Core/Non-core, Clusters, outliers time (sec)**	**MBSCAN time (excl. *iForest*) (sec)**
Libras	20	5	0.31694	15	16.751	0.004702	16.7695
Segment	20	10	0.52816	25	1130.08	0.136521	1130.26
Wine	20	5	0.370225	16	2.08052	0.001545	2.0915
Seeds	21	7	0.44195	16	3.03881	0.004456	3.05231
Aggregation	24	9	0.426301	17	145.055	0.033157	145.118
Iris	20	7	0.405	18	1.361	0.002541	1.36982
S1	20	9	0.344	18	244.346	0.080631	244.476
S2	20	10	0.23113	16	951.056	0.107407	952.131

time required to build the $iForest$ is also significant, even though it is directly not involved in the cluster extraction process.

Table 3.5: Percentage of total time required to construct the mass-matrix

Dataset	Libras	Segment	Wine	Seeds	Aggregation	Iris	S1	S2
Percentage of total time for constructing the mass-matrix	99.88	99.98	99.47	99.55	99.95	99.42	99.94	99.88

Table 3.6: Memory consumed on MBSCAN run.

Dataset	Libras	Segment	Wine	Seeds	Aggregation	Iris	S1	S2
Memory	15.76 MB	508.86 MB	6.38 MB	7.53 MB	63.04 MB	5.47 MB	81.49 MB	215.91 MB

3.6.1 Inferences drawn from experiments on MBSCAN

The following inferences may be drawn from the experiments that were conducted for evaluating MBSCAN:

1. The mass-matrix turns out to be the most expensive component of the MBSCAN algorithm.

2. Apart from building mass-matrix, the construction of *iForest* also contributes significantly to the whole process.

3. Extraction of clusters along with core and non-core points do not make a substantial contribution to the final execution time.

While designing the *iMass* algorithm, we target building of mass-matrix incrementally since it happens to be one of the heavier components of MBSCAN. We also construct the updated *iForest* incrementally for further enhancing the effectiveness of *iMass*.

3.7 The *iMass* Clustering Algorithm

3.7.1 Theoretical Model

Let $D = \{O_1, O_2, O_3, \ldots, O_n\}$ be the set of objects clustered by the non-incremental MBSCAN algorithm (let it be denoted by $\mathcal{M}$). Each object in D is characterized by a set of d attributes and is represented by a d-dimensional vector. Therefore $O_i = \{O_{i1}, O_{i2}, \ldots, O_{id}\}$, where $O_{im} \in \mathcal{R}^+, 1 \leq i \leq n, 1 \leq m \leq d$.

Let O_{n+1} be the new object added upon the base dataset D. D changes to D' where $D' = \{O_1, O_2, O_3, \ldots, O_n, O_{n+1}\}$ is the current set of objects to be clustered. We intend to address the problem of recomputing clusters post dataset expansion. The new set of clusters can be obtained by applying $\mathcal{M}$ on D'. However, we aim to avoid this process by developing a less expensive method called *iMass* (let it be denoted by $\mathcal{IM}$).

Let $C_{\mathcal{M}}$ and $C'_{\mathcal{M}}$ be the set of clusters obtained by executing algorithm $\mathcal{M}$ upon dataset D and D', then we may have the following interpretations:

- $C_{\mathcal{M}} = \mathcal{M}(D)$;
- $C'_{\mathcal{M}} = \mathcal{M}(D')$;
- $C'_{\mathcal{IM}} = \mathcal{IM}(D')$;
- $C'_{\mathcal{M}} \approx C'_{\mathcal{IM}}$, where $C'_{\mathcal{IM}}$ is the set of clusters obtained by executing algorithm $\mathcal{IM}$ upon dataset D'.

3.7.2 Assumptions made for the $iMass$ clustering algorithm:

For the incremental algorithm $iMass$, we make the following assumptions:

1. The total number of $iTrees$ (t) used by MBSCAN is retained by the $iMass$ algorithm for its execution.

2. The split-attribute (q) and split-value (p) for any node belonging to $iTree_i, i = 1, 2, 3, \ldots, t$, does not change after the addition of new object $O_{n+j}, 1 \leq j \leq k$ [71]. Here k is the maximum number of insertions to be made and $k \ll |D|$.

3. While computing the nodal mass post insertion of a new object O_{n+j} by traversing all instances in D', the height of any $iTree_i, i = 1, 2, 3, ..., t$ obtained in the prior run of MBSCAN remains unchanged.

3.7.3 Retain the components of MBSCAN algorithm

A prior execution of MBSCAN is performed in order to produce the information related to components viz. base dataset D, $iForest$, mass-matrix along with a lowest node identifier[5] (*lowest node_id*) $\forall$ $iTree_i, i = 1, 2, 3, ..., t$ and $\forall$ $(x, y) \in D$. The values of these components are retained and utilized for an efficient design of the $iMass$ algorithm. Figure 3.5 shows the sequence of computation performed while executing $iMass$.

3.7.4 Steps of the $iMass$ clustering algorithm

1. Load D, $iForest$, mass-matrix, the *lowest node_id* from each $iTree$ for every pair of points $(x, y) \in D$ produced from the MBSCAN run.

2. Insert a new point; D changes to D'.

3. Compute the mass of lowest leveled nodes $\forall$ $(x, y) \in D$ in each $iTree$ within the $iForest$ incrementally.

4. Build the mass-matrix incrementally.

5. Compute the updated μ-neighborhood mass $M_u(x)$ $\forall$ $x \in D'$.

[5]The lowest node identifier for any pair (x, y) in an $iTree$ provides the node number of the lowest leveled node containing x and y.

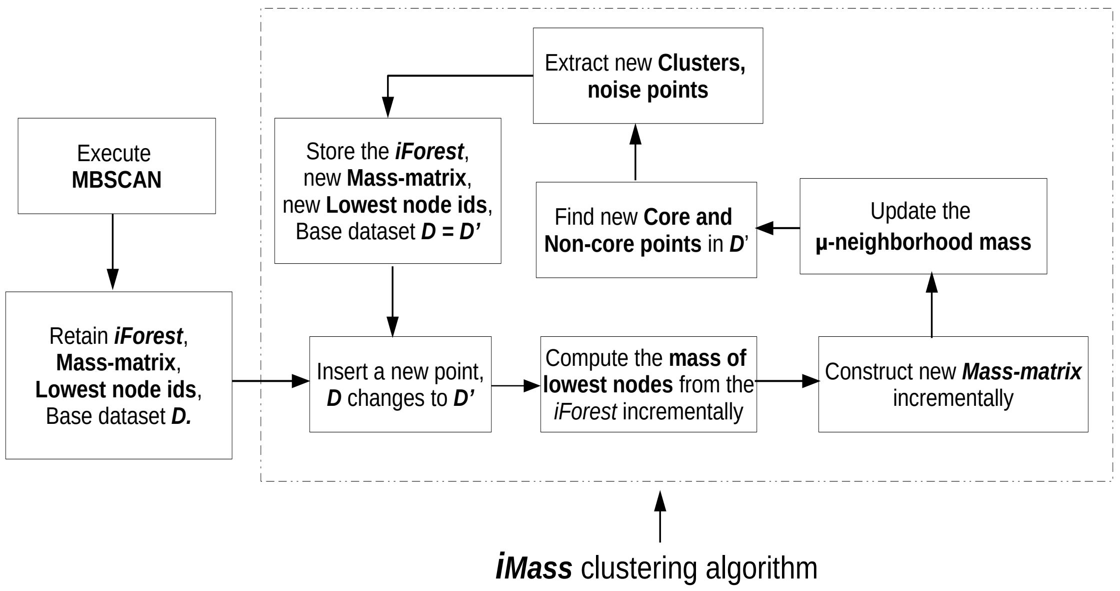

Execute MBSCAN
Retain iForest, Mass-matrix, Lowest node ids, Base dataset D.
Insert a new point, D changes to D'
Compute the mass of lowest nodes from the iForest incrementally
Construct new Mass-matrix incrementally
Update the μ-neighborhood mass
Find new Core and Non-core points in D'
Extract new Clusters, noise points
Store the iForest, new Mass-matrix, new Lowest node ids, Base dataset D = D'
iMass clustering algorithm

FIGURE 3.5: Sequence of execution for the *iMass* clustering algorithm.

6. Identify the set of core and non-core points from D' based on the updated μ-neighborhood mass.

7. Cluster the objects of D' and filter out the noise points.

8. D' becomes the new base dataset D. Repeat steps 2 to 8 till all the insertions are made [71].

Next we describe each of the steps involved in the *iMass* clustering algorithm.

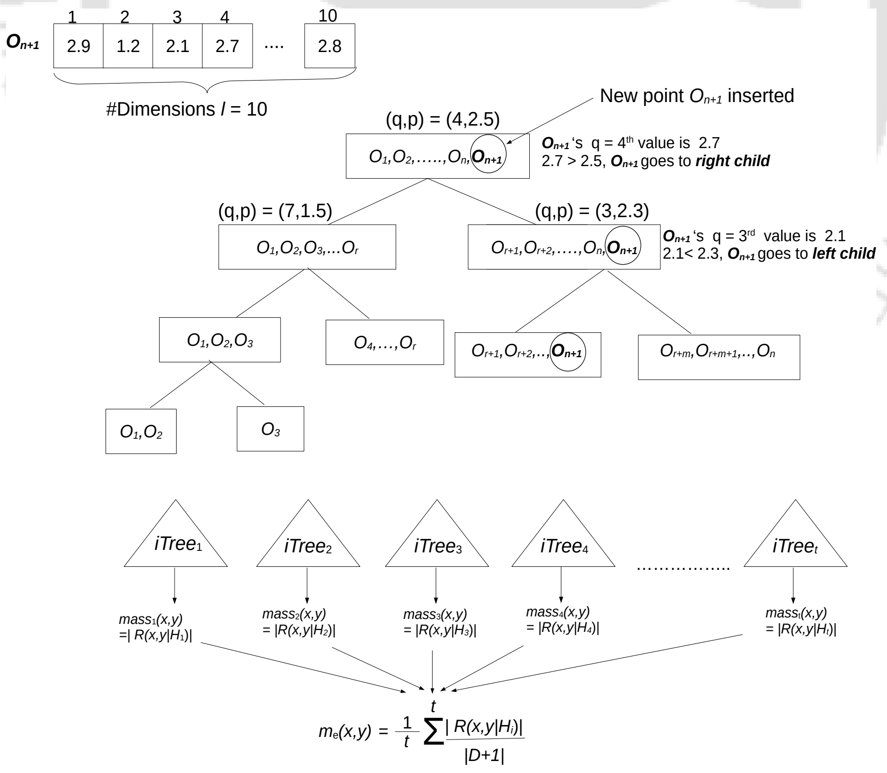
1. **Step 1 - Load** D, $iForest$, **mass-matrix,** *lowest node_id*: In this step the values of components obtained from the initial run of MBSCAN: base dataset D where $|D| = n(say)$, the $iForest$, mass-matrix, the *lowest node_id* for any pair of points (x, y) from each of the $iTrees$ are loaded. The *lowest node_ids* are stored in order to quicken the mass-matrix building process incrementally.

   For our purpose we initialize a variable i to 1. The variable i keeps a count of the number of new insertions made to D.

2. **Step 2 - Insert a new point,** D **changes to** D'**:** Upon entry of a new point O_{n+i}, the base dataset D increases its size by one and changes to $D'(|D'| = |D| + 1)$.

3. **Step 3 - Compute the mass of lowest leveled nodes in each $iTree$ within the $iForest$ incrementally:** $\forall\ iTree_j [1 \leq j \leq t]$, the newly entered point O_{n+i} uses the splitting condition for each node as per the axis-parallel split algorithm [2] (Section 3.5) and finds the appropriate nodes for positioning itself within that $iTree$.

   The $iMass$ algorithm scans the presence of a newly entered point O_{n+i} in each $iTree_j$. Therefore for every node within an $iTree$, the algorithm uses the same split-attribute (q) and split-value (p) figures obtained from prior MBSCAN execution to decide whether the new point O_{n+i} traces to left-child or right-child of that concerned node. The point O_{n+i} compares the split-attribute (q) value of the concerned node with its own q^{th} attribute value. If the value is less than the node's q^{th} attribute value, then O_{n+i} goes to the left child of the node, otherwise it goes to the right child. The old points in any node of an $iTree$ do not alter their position and therefore such points retain their previous lowest node identifier value in any $iTree$. This procedure is repeated for each $iTree_j$ within the $iForest$. The algorithm therefore efficiently scans the presence $\forall\ x \in D'$ from the $iForest$ and computes the nodal mass (including lowest leveled node) incrementally.



O_{n+1}
1 2 3 4 10
2.9 1.2 2.1 2.7 .... 2.8
#Dimensions l = 10
New point O_{n+1} inserted
(q,p) = (4,2.5)
$O_1,O_2,\ldots,O_n,O_{n+1}$
O_{n+1}'s q = 4th value is 2.7
2.7 > 2.5, O_{n+1} goes to ***right child***
(q,p) = (7,1.5)
(q,p) = (3,2.3)
$O_1,O_2,O_3,\ldots O_r$
$O_{r+1},O_{r+2},\ldots,O_n,O_{n+1}$
O_{n+1}'s q = 3rd value is 2.1
2.1< 2.3, O_{n+1} goes to ***left child***
O_1,O_2,O_3
$O_4,\ldots,O_r$
$O_{r+1},O_{r+2},\ldots,O_{n+1}$
$O_{r+m},O_{r+m+1},\ldots,O_n$
O_1,O_2
O_3
$iTree_1$
$iTree_2$
$iTree_3$
$iTree_4$
$iTree_t$
$mass_1(x,y) = |R(x,y|H_1)|$
$mass_2(x,y) = |R(x,y|H_2)|$
$mass_3(x,y) = |R(x,y|H_3)|$
$mass_4(x,y) = |R(x,y|H_4)|$
$mass_t(x,y) = |R(x,y|H_t)|$
$m_e(x,y) = \frac{1}{t}\sum^{t}\frac{|R(x,y|H_i)|}{|D+1|}$

FIGURE 3.6: Compute the node mass incrementally upon new point insertion.

Running example: Let us assume that the *iTree* in Figure 3.6 has been constructed while executing the MBSCAN algorithm. Prior to any insertion, let there be n objects in the base dataset D. where $D = \{O_1, O_2, O_3, \ldots, O_n\}$. All the n instances of D are traversed through each of the t *iTrees* while computing the mass of individual node during MBSCAN execution.

In this current example, we highlight the positioning of a newly entered point O_{n+1} w.r.t., the *iTree* in Figure 3.6. Upon entry of point O_{n+1}, D updates to $D' = \{O_1, O_2, O_3, ..., O_n, O_{n+1}\}$ where $|D'| = n + 1$. Let us consider that initially, the point O_{n+1} places itself in the root node of the *iTree*. From prior execution of MBSCAN, let us assume that the (q, p) values of root node are 4 and 2.5 respectively. According to the splitting criterion of a node (Refer Algorithm 1), the point O_{n+1} compares its $q = 4^{th}$ attribute value *i.e.*, 2.7 with the split-value (p) of the root node *i.e.*, 2.5. Based on this comparison, the new point O_{n+1} traces itself to the right child of the root node ($\because 2.7 \geq 2.5$).

Noticeably, point O_{n+1} finds itself placed at the 3^{rd} node lying in *Level* 1 of the *iTree* (Figure 3.6). The 3^{rd} node of *iTree* which currently holds O_{n+1} has a pre-estimated (q, p) values of 3 and 2.3 respectively. Upon comparing the $q = 3^{rd}$ attribute value of O_{n+1} *i.e.*, 2.1 with that of the 3^{rd} node *i.e.*, 2.3, O_{n+1} identifies the 6^{th} node lying in *Level* 2 of the *iTree* as its next destination. If we consider the 6^{th} node as one of the leaf nodes, then no further split happens w.r.t., point O_{n+1}.

The updated nodal mass can be found out by simply keeping a count of the number of elements within that node post insertion of new point. This procedure of accommodating the new point O_{n+1} is performed for all the t *iTrees*. The updated nodal mass for all the lowest leveled nodes $\forall\,(x, y) \in D$ in individual *iTree* can therefore be computed incrementally.

4. **Step 4 - Build the mass-matrix incrementally:** In this step, the probability mean-mass or mass $\forall\,(x, y) \in D$ (base dataset) is updated incrementally. The mass value for new point $O_{n+i} \in D'$ (updated dataset) with each of the old points $x \in D$ is computed similar to the MBSCAN method.

   Mathematically deduce the updated mass computation of lowest leveled node $\forall\, x, y \in D$: $\forall\,(x, y) \in D$, let m_u be the mass of lowest leveled node obtained from MBSCAN for u^{th} *iTree* $[1 \leq u \leq t]$. Given that t number of *iTrees* are involved in building the *iForest*, then as per the MBSCAN [2] algorithm the probability mean-mass between x and y is formulated as:

$$
\begin{aligned}
m_e(x,y) &= \frac{1}{t \cdot |D|}[m_1 + m_2 + m_3 + ... + m_t] \\
\implies m_e(x,y) &= \frac{\sum_{u=1}^{t} m_u}{t \cdot |D|} \\
\implies \frac{\sum_{u=1}^{t} m_u}{t} &= |D| \cdot m_e(x,y)
\end{aligned} \tag{3.6}
$$

When the new point O_{n+i} enters upon the base dataset D, each $iTree_u, u = 1, 2, 3, ..., t$ will initially have O_{n+i} in its root node. As observed in the previous step (Step 3), any newly inserted point O_{n+i} will position itself in the appropriate nodes of an $iTree$ using the pre-estimated (q, p) values of that node obtained during execution of MBSCAN. We may therefore have the following scenario:

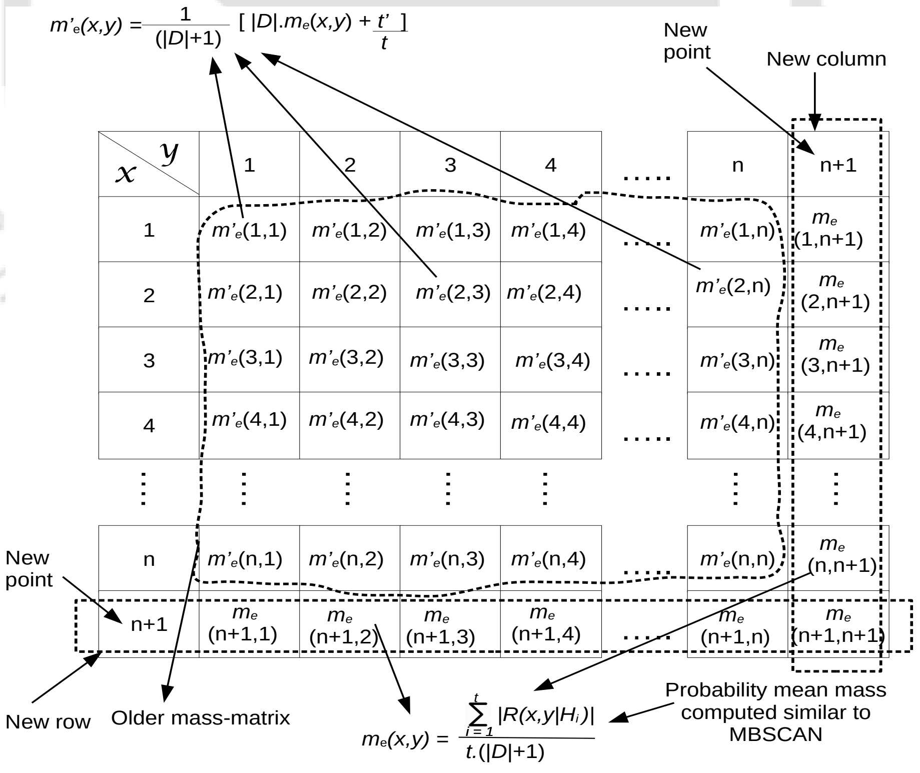
For any pair of points (x, y), the new point O_{n+i} on its entry may penetrate into the lowest leveled node of an $iTree$ containing x and y. As a result, the contributory mass $m_u, u = 1, 2, ..., t$ for the pair (x, y) w.r.t., the current $iTree$ will increase by 1. In worst case scenario, if the lowest leveled node containing the pair of points (x, y) in all the $iTrees$ are affected due to entry of O_{n+i}, then the contributory mass for x and y will increase by 1 for each of the $iTrees$. Let $m'_e(x, y)$ be the updated probability mean-mass between x and y, then we may represent the above scenario as follows:

$$
\begin{aligned}
m'_e(x,y) &= \frac{1}{t \cdot (|D|+1)}[(m_1+1) + (m_2+1) + (m_3+1) + ... + (m_t+1)] \\
\implies m'_e(x,y) &= \frac{(\sum_{u=1}^{t} m_u) + t'}{t \cdot (|D|+1)} \quad [Here \quad t' = t]
\end{aligned} \tag{3.7}
$$

Equation 3.7 enables evaluation of updated probability mean-mass between any pair of old points in the new mass matrix when the lowest leveled node containing the involved points gets affected by the entry of a new point. Proceeding further, we shall generalize the updated mass computation for any $(x, y) \in D$ when a fraction (maximum t) of $iTrees$ are affected. From Equation 3.7, we therefore have:

$$
\begin{aligned}
m'_e(x,y) &= \frac{\left(\sum_{u=1}^{t} m_u\right) + t'}{t \cdot (|D|+1)} \\
\implies m'_e(x,y) &= \frac{\sum_{u=1}^{t} m_u}{t \cdot (|D|+1)} + \frac{t'}{t \cdot (|D|+1)} \\
\implies m'_e(x,y) &= \frac{|D| \cdot m_e(x,y)}{(|D|+1)} + \frac{t'}{t \cdot (|D|+1)} \quad [Using\ \ Equation\ 3.6] \\
\implies m'_e(x,y) &= \frac{1}{(|D|+1)}[|D| \cdot m_e(x,y) + \frac{t'}{t}] \ [Here\ 0 \le t' \le t]
\end{aligned} \tag{3.8}
$$

From Equation 3.8, we observe that in order to calculate the updated mass[6] $\forall\,(x,y) \in D$ post insertion of a data point, the following variables are involved: $D, m_e(x,y), t'$, and t. Apart from t' (number of affected *iTrees*), the value of remaining variables are already estimated during execution of MBSCAN. Therefore on finding the number of affected *iTrees* due to entry of new point O_{n+i}, we can estimate the value of t' and compute the updated probability mean-mass $\forall\,(x,y) \in D$ (base dataset) in the new mass-matrix.



m'ₑ(x,y) = 1/(|D|+1) [|D|.mₑ(x,y) + t'/t]
New point
New column
x y
1 2 3 4 ..... n n+1
m'ₑ(1,1) m'ₑ(1,2) m'ₑ(1,3) m'ₑ(1,4) m'ₑ(1,n) mₑ (1,n+1)
m'ₑ(2,1) m'ₑ(2,2) m'ₑ(2,3) m'ₑ(2,4) m'ₑ(2,n) mₑ (2,n+1)
m'ₑ(3,1) m'ₑ(3,2) m'ₑ(3,3) m'ₑ(3,4) m'ₑ(3,n) mₑ (3,n+1)
m'ₑ(4,1) m'ₑ(4,2) m'ₑ(4,3) m'ₑ(4,4) m'ₑ(4,n) mₑ (4,n+1)
m'ₑ(n,1) m'ₑ(n,2) m'ₑ(n,3) m'ₑ(n,4) m'ₑ(n,n) mₑ (n,n+1)
n+1 mₑ (n+1,1) mₑ (n+1,2) mₑ (n+1,3) mₑ (n+1,4) mₑ (n+1,n) mₑ (n+1,n+1)
New point
New row
Older mass-matrix
mₑ(x,y) = Σ_{i=1}^{t} |R(x,y|Hᵢ)| / t.(|D|+1)
Probability mean mass computed similar to MBSCAN

FIGURE 3.7: Updated mass-matrix post insertion of a new point.

[6]We use the terms 'probability mean-mass' or 'mass' interchangeably.

However, for computing the probability mean-mass between newly inserted point O_{n+i} and each of the older points, we adopt the methodology identical to the MBSCAN algorithm. An additional row and a column is added to the old mass-matrix for storing the new mass values. Figure 3.7 gives a representation of the updated mass-matrix when a new point is added.

5. **Step 5 - Compute the updated μ-neighborhood mass $M_u(x) \forall\, x \in D'$.**

   Upon entry of new point O_{n+i}, D changes to D'. Now $\forall\, x \in D'$, the μ-neighborhood mass $M_u(x)$ is computed similar to MBSCAN using Equation 3.5.

6. **Step 6 - Identify the set of core and non-core points from D':** Given that we retain the core-point formation threshold δ_{core}, the set of core and non-core points from D' is determined similar to the MBSCAN algorithm.

7. **Step 7 - Extract clusters from D' and filter out noise points:** This step is also similar to MBSCAN.

8. **Step 8 - Update the base dataset and store the component values:** Set $D = D'$, store the new mass-matrix and new *lowest node_ids* $\forall\, (x, y) \in D$. Set $i = i + 1$, repeat Steps 2 to 8 till $i = k$.

3.8 Time Complexity comparison between MBSCAN and *iMass*

For MBSCAN [2] algorithm, the construction of *iForest* takes $\mathcal{O}(t\Psi \log_2 \Psi)$ time with t being the number of *iTrees* and Ψ is the subsample size (Refer Section 3.5). However while computing the mass of nodes (including lowest leveled nodes), all the objects from base dataset $D\,(|D| = n\,(say))$ need to be traversed through each *iTree*. The necessary nodal mass calculation from the *iForest* involves a cost of $\mathcal{O}(tn \log_2 \Psi)$. The non-incremental mass-matrix builds to completion in $\mathcal{O}(tn^2 \log_2 \Psi)$ time. The core and non-core points along with clusters and noise points are extracted in linear time.

In case of our proposed *iMass* algorithm, we need not construct the *iForest* post insertion of a new point. Only the lowest leveled nodal mass is computed incrementally from each *iTree*. This step invokes a running time of $\mathcal{O}(t \log_2 \Psi)$ against every insertion. Upon comparing with MBSCAN, the construction of

$iForest$ along with the nodal mass computation for $iMass$ achieves a gain factor of $\mathcal{O}(n)$.

TABLE 3.7: Time complexity comparison between MBSCAN and $iMass$.

Components	MBSCAN	*iMass*	Gain factor
***iForest* + nodal mass computation**	$\mathcal{O}(t(n+\Psi)\log_2\Psi)$	$\mathcal{O}(t\log_2\Psi)$	$\mathcal{O}(n)$
Mass-matrix	$\mathcal{O}(tn^2\log_2\Psi)$	$\mathcal{O}(tn^2+t(n+1)\log_2\Psi)$	$\mathcal{O}(log_2\Psi)$
Core and non-core points	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Cluster and noise points	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$

While executing the mass-matrix for MBSCAN, we store the *lowest node_ids* $\forall\,(x,y)\in D$. When any new point enters the base dataset, $iMass$ refers to the stored *lowest node_id* to check whether the new point has penetrated into the lowest leveled node for that concerned pair of points w.r.t., a given $iTree$. In order to find the number of affected $iTrees$ t' (Refer Equation 3.8) due to entry of a new point, at most t number of scans are required in the whole $iForest$. Since the old mass-matrix consists of n^2 number of cells (prior to first point insertion), therefore the updation of probability mean mass values $\forall\,(x,y)\in D$ (base dataset) in the new mass-matrix consumes $\mathcal{O}(tn^2)$ time. The calculation of lowest node masses between the new point and each of the old points take $\mathcal{O}(t(n+1)\log_2\Psi)$ time. On comparing with MBSCAN, we estimate a gain factor of $\mathcal{O}(\log_2\Psi)$.

Given that $t,\Psi \ll n$, we present a tabular comparison (Table 3.7) of the time complexities of both MBSCAN and the incremental $iMass$ algorithm when a single point is added upon base dataset.

3.9 Theoretical analysis of the $iMass$ clustering algorithm

In this section, we analyze the possible scenarios that may exist while updating the old mass matrix incrementally. We also present few lemmas related to the $iMass$ algorithm.

3.9.1 Cases related to updated mass-matrix:

From Equation 3.8 we have:
$m'_e(x,y) = \frac{1}{(|D|+1)}[|D| \cdot m_e(x,y) + \frac{t'}{t}]$ [where $0 \leq t' \leq t$]

1. **Case 1: $t' = 0$:** For any given pair of points (x, y), if the lowest leveled node in none of the *iTrees* are affected due to entry of new point, then $t' = 0$. In such a case:
   $m'_e(x,y) = \frac{1}{(|D|+1)}[|D| \cdot m_e(x,y)]$
   The updated mass becomes independent of the number of *iTrees* and is reduced to a function of D and old mass value $m_e(x, y)$.

2. **Case 2: $0 < t' < t$:** If the number of affected *iTrees* is greater than zero but less than t, then the updated probability mean-mass is derived directly from Equation 3.8.

3. **Case 3: $t' = t$:** If all the *iTrees* are affected, then $t' = t$. In such a case Equation 3.8 reduces to $m'_e(x,y) = \frac{1}{(|D|+1)}[|D| * m_e(x,y) + 1]$. Since the value of the term $m_e(x, y)$ is already known from execution of MBSCAN, similar to Case 1, the updated mass matrix becomes independent of the number of *iTrees* involved.

3.9.2 Lemmas related to the *iMass* clustering algorithm:

Lemma 3.1. $\forall\, x, y \in D, m'_e(x,y) > 0$.

Proof: The minimum value of $m_e(x, y)$ happens when the lowest node contains only x and y ($x \neq y$). In this case the probability mass w.r.t., a given *iTree* is $\frac{2}{|D|}$. Therefore, the minimum probability mean-mass from MBSCAN becomes $\frac{2t}{t*|D|} = \frac{2}{|D|} > 0[\because |D| > 0]$. When $x = y$, the minimum mass turns out to be $\frac{1}{|D|} > 0$. Since the updated mass $m'_e(x, y)$ is a function of $m_e(x, y), D, t$ and t' (Refer Equation 3.8), therefore $\forall\, x, y \in D, m'_e(x,y) > 0$ for all the three cases related to updated mass matrix.

Lemma 3.2. *$\forall\, x, y \in D$, when no iTree is affected, $m'_e(x,y) < m_e(x,y)$.*

Proof: When no *iTree* is affected, $t' = 0$. Therefore from Equation 3.8 we have $\frac{m'_e(x,y)}{m_e(x,y)} = \frac{|D|}{|D+1|}$. Therefore $m'_e(x,y) < m_e(x,y)$ $[\because \frac{|D|}{|D+1|} < 1]$.

Lemma 3.3. *$\forall\, x \in D \cap D'$, x may or may not retain its core or non-core status obtained during previous insertion.*

Proof: The μ threshold (Refer Equation 3.5) may change after a point insertion, but in spite of the decrease in pair wise probability mean-mass values (Lemma 3.2), it cannot be guaranteed that the size of μ-neighborhood mass for any point will continue to increase or decrease.

3.10 Experimental evaluation

We performed experiments on six real world and two synthetic datasets (Refer Table 3.3 in Section 3.6 for datasets' description) and simulated our proposed $iMass$ clustering algorithm in C++ on a Linux machine with 4GB RAM. Through the experiments, we aim to find the following results while proving the efficiency of $iMass$ over MBSCAN:

1. Compare the CPU execution times of MBSCAN and $iMass$.
2. The percentage of affected nodes in the $iForest$ against every point insertion made.
3. The extent of reduction achieved while building the mass-matrix in an incremental manner for $iMass$ as compared to MBSCAN.
4. The extent of reduction achieved due to $iMass$ while finding the lowest node masses incrementally from the $iForest$ post any insertion.

3.10.1 Experimental procedure

For our experimental purpose, we defined a new term called Algorithm-Components (Algo-Comp). The Algo-Comp consists of following components:

- Base dataset D.
- $iForest$.
- Mass-matrix.

- Lowest node identifier (*lowest node_id*) in each *iTree* for any pair of points.

A base dataset is one taking which MBSCAN is executed in order to set the values of other components in Algo-Comp. Initially, the base dataset is fed as input to the *iMass* algorithm. Upon entry of a new point, the base dataset alters its size. The new set of points becomes a part of the updated dataset D'. D', nodes with updated masses from the *iForest*, new mass-matrix and *lowest node_id*s become a part of the updated Algo-Comp upon execution of *iMass*. The next arriving point is processed based on the updated Algo-Comp values from the previous iteration.

Table 3.8: Parameters and Data division for experiments related to *iMass* algorithm.

Dataset	**Size**	**Base dataset size**	**#Points to be added**	**#*iTrees* *t***	δ_{core}
Libras	360	240	120	20	5
Segment	2310	1540	770	20	10
Wine	178	118	60	20	5
Seeds	210	140	70	21	7
Aggregation	788	525	263	24	9
Iris	150	100	50	20	7
S1	900	600	300	20	9
S2	1500	1000	500	20	10

For carrying out experiments, we divided the actual dataset into a 2:1 ratio (Table 3.8) where the larger share was taken as the base dataset (D) and the smaller share was the set of points to be added upon D one at a time. We initially executed the MBSCAN algorithm upon D and stored the Algo-Comp values, and then allowed the set of points within smaller share to be inserted one at a time. For each dataset, we measured the CPU time required due to execution of *iMass* for detecting clusters after inserting a single point. Correspondingly, we also executed MBSCAN on the whole updated dataset (D'). Experimentally we compared $CPU_{MBSCAN(D')}$ and $CPU_{iMass(D')}$ after every insertion to show that:

$$CPU_{iMass(D')} < CPU_{MBSCAN(D')} \tag{3.9}$$

The next evaluation metric used was the percentage of affected nodes in the *iForest* against every new insertion. Any node within an *iTree* which accommodates a newly entered point is designated as an affected node.

$$Affected\,nodes(\%) = \frac{\#Nodes\ with\ new\ point}{\#Total\ Nodes\ in\ iForest} \cdot 100 \tag{3.10}$$

Along with aforementioned metrics used for evaluation of $iMass$ viz. CPU execution times and percentage of affected nodes (Equation 3.9, Equation 3.10), we also determined the extent of reduction achieved by $iMass$ in time required for mass-matrix construction and nodal-mass computation from the $iForest$.

3.10.2 Experimental results:

In this phase, we present the key experimental results that were obtained based on the metrics described in previous phase (experimental procedure). We also present certain observations and reasons pertaining to the experimental results.

Key Results: For every dataset, there are two horizontally placed plots (Figures 3.8(a), 3.8(b), 3.8(c), 3.8(d), 3.8(e), 3.8(f), 3.9(a), 3.9(b), 3.9(c), 3.9(d), 3.9(e), 3.9(f), 3.10(a), 3.10(b), 3.10(c) and 3.10(d)) viz. CPU execution times of $iMass$ and MBSCAN and the percentage of affected nodes from the $iForest$ due to execution of $iMass$. Both the algorithms show an upward trend in execution times with increase in the size of base dataset due to repeated insertions. We observed that for Iris dataset, the $iMass$ algorithm achieved a highest speedup upto an order of 2.28 across all datasets which is about 191 times faster than the MBSCAN method. The average speedup obtained for Iris was upto an order of 2.11 ($\approx$ 128.82 times) post insertion of the 50^{th} point. For another dataset Wine, a highest speedup of about 134 times was achieved. The mean speedup for Wine was around 83.17 ($\approx$ order of 1.92). Among the datasets achieving a lesser mean speedup, it was about 23.44 times faster ($\approx$ order of 1.37) for Aggregation dataset while for S1 the mean speedup was about 27.54 ($\approx$ order of 1.44).

While computing the percentage of affected nodes in the $iForest$, for Iris dataset we observed that an average of 26.54% of nodes were affected after all the 50 points had been inserted. For the highest sized dataset Segment with a insertion size of 770 points, the mean number of affected nodes were as low as 8.94%. The datasets with lesser size viz. Wine and Seeds had a higher mean percentage of affected nodes with 24.5% and 24.61% respectively. This is due to the fact that $iForest$s that were constructed with datasets of higher size had a greater number of nodes per $iTree$ contrary to datasets with lesser size.

Out of all datasets, Segment consumed the highest amount of memory space because of its greater size. The average memory consumption for $iMass$ stood at 164.72 MB while MBSCAN took about 113.055 MB of space. The mean memory consumption due to $iMass$ algorithm was about 66.77% more than that of MBSCAN (See Tables 3.6, 3.9), $\therefore |Mem_{point-ins} - Mem_{naive}| \approx 0.67$. The higher memory consumption for $iMass$ can be attributed to the maintenance of the $lowest\ node\_id$ for every pair of points in each of the t $iTrees$. Moreover, certain additional memory was also required for storing the parameters carried from the MBSCAN algorithm for all insertions.

TABLE 3.9: Memory consumed due to $iMass$.

Dataset	**Libras**	**Segment**	**Wine**	**Seeds**	**Aggregation**	**Iris**	**S1**	**S2**
Memory	28.67 MB	632.84 MB	9.42 MB	12.76 MB	137.33 MB	7.95 MB	158.67 MB	330.132 MB

Next we mention certain observations and reasons based on the experimental results that were obtained.

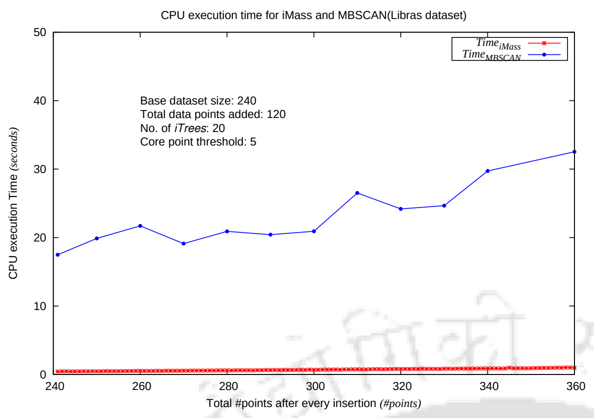
1. **Efficiency comparison of $iMass$ and MBSCAN**
   Key observation(s): In the first plot for every dataset (Figure 3.8(a) through Figure 3.10(c)), we observed that against each point insertion, the CPU execution time for MBSCAN upon D' (updated dataset) is much higher as compared to that of the $iMass$ algorithm resulting in its better efficiency.

   Reason(s): The $iMass$ algorithm avoids re-building of $iForest$ post any new insertion. Only the nodal masses of the affected nodes are updated incrementally. Moreover while constructing the new mass-matrix, a pairwise dissimilarity score between any two existing points in D (base dataset) is evaluated incrementally (Section 3.7.4).

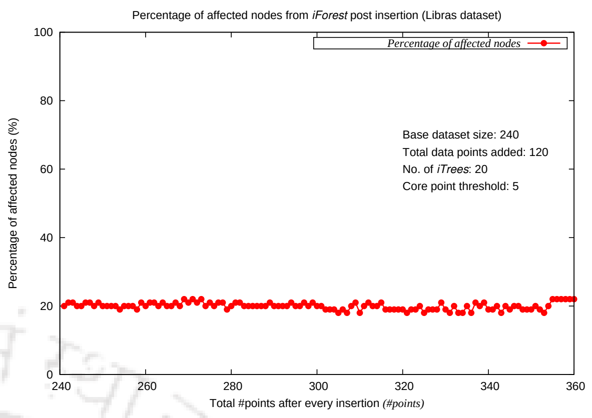
2. **Lower percentage of affected nodes**
   Key observation(s): In the second plot for each dataset (Figure 3.8(b) through Figure 3.10(d)), we observe that a lesser percentage of nodes ($< 28\%$ across datasets) were affected due to insertion of any new point.

   Reason(s): The node-splitting criterion as per the MBSCAN algorithm is used by a newly inserted point to find the appropriate nodes while positioning itself in any $iTree$. Therefore only a handful number of nodes across $iTrees$ belonging to a given $iForest$ would contain the newly entered point.

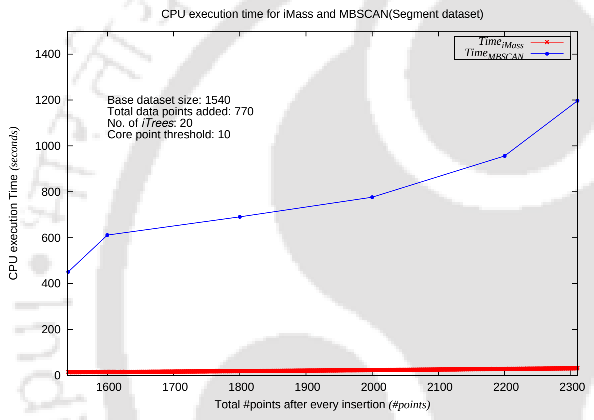
3. **Efficiency achieved due to $iMass$ while building the $iForest$ incrementally**



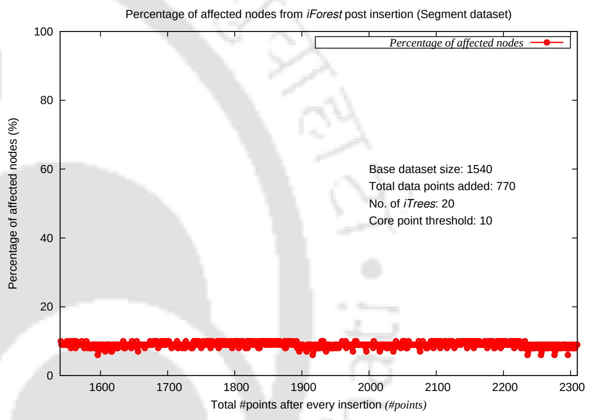
(a) Execution time for *iMass* and MBSCAN after every insertion for Libras dataset.



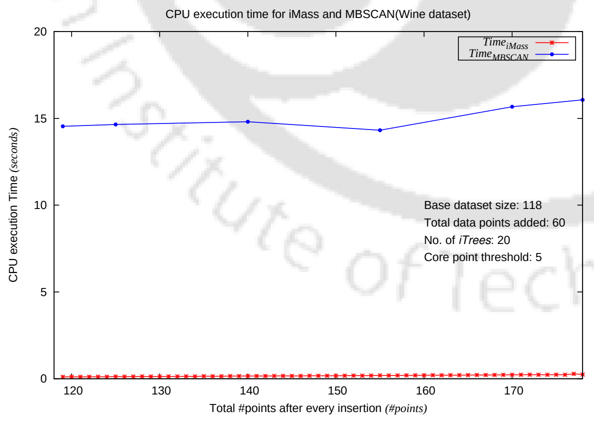
(b) Percentage of affected nodes in the *iForest* after every insertion for Libras dataset.



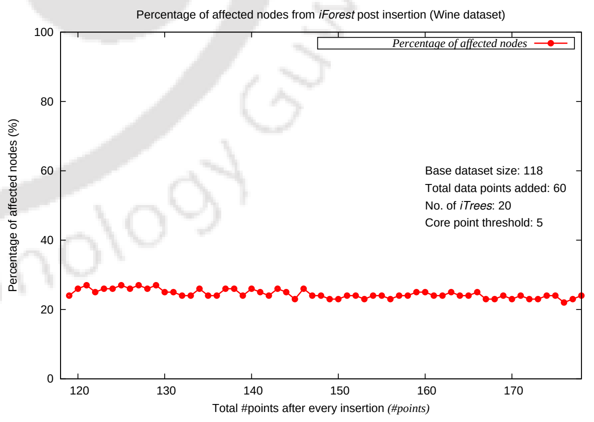
(c) Execution time for *iMass* and MBSCAN after every insertion for Segment dataset.



(d) Percentage of affected nodes in the *iForest* after every insertion for Segment dataset.

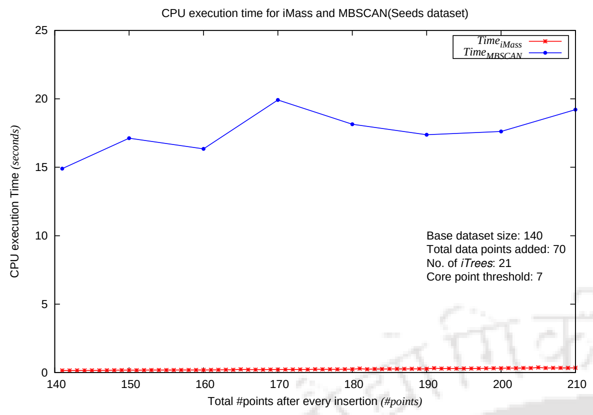


(e) Execution time for *iMass* and MBSCAN after every insertion for Wine dataset.

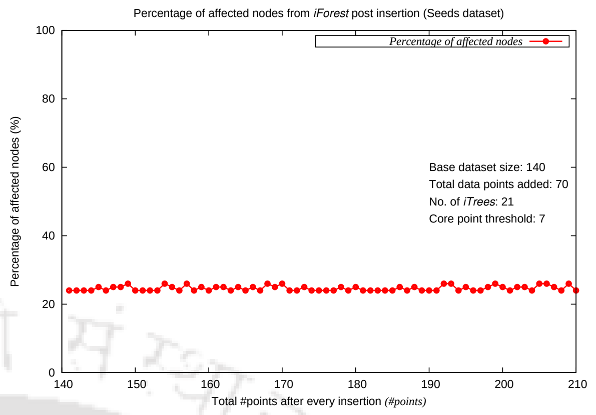


(f) Percentage of affected nodes in the *iForest* after every insertion for Wine dataset.

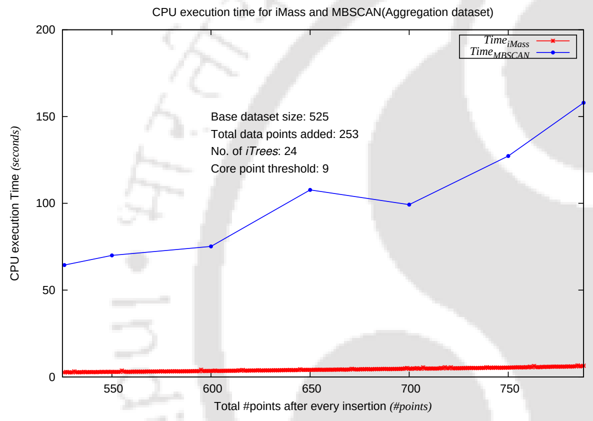
FIGURE 3.8: Efficiency comparison between *iMass* and MBSCAN along with the percentage of affected nodes due to *iMass* for datasets: Libras, Segment, Wine.



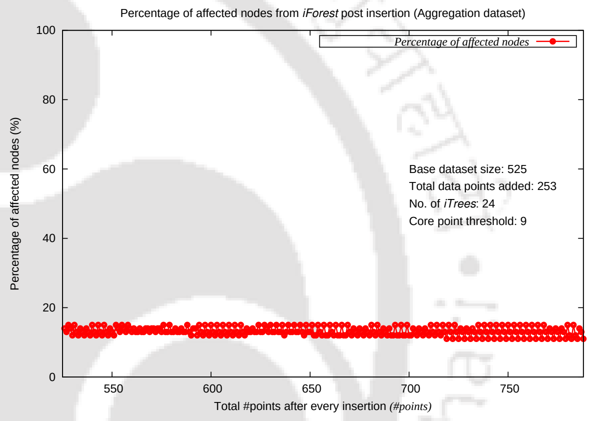
(a) Execution time for *iMass* and MBSCAN after every insertion for Seeds dataset.



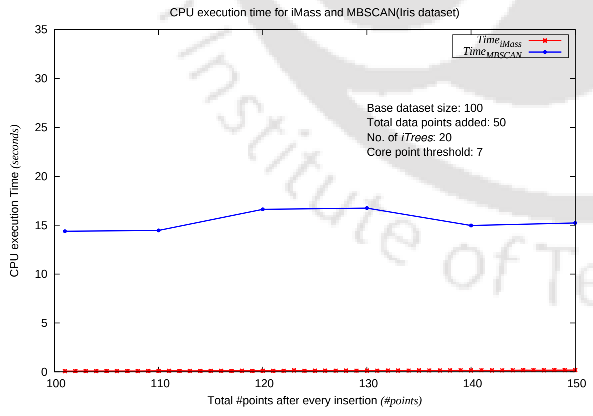
(b) Percentage of affected nodes in the *iForest* after every insertion for Seeds dataset.



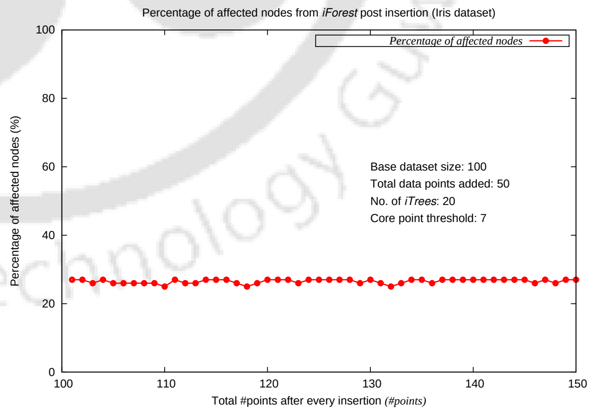
(c) Execution time for *iMass* and MBSCAN after every insertion for Aggregation dataset.



(d) Percentage of affected nodes in the *iForest* after every insertion for Aggregation dataset.

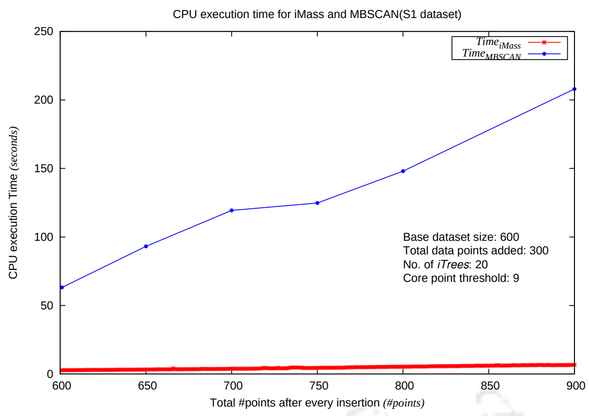


(e) Execution time for *iMass* and MBSCAN after every insertion for Iris dataset.

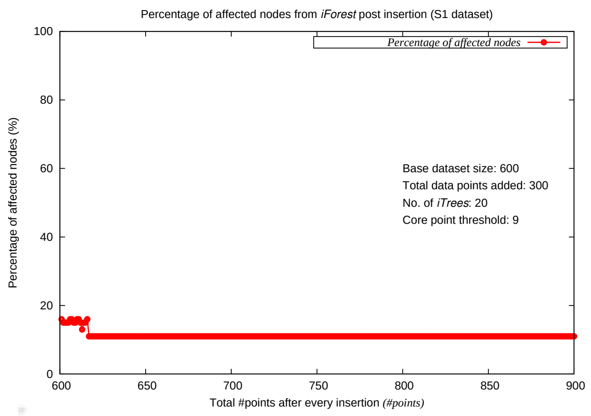


(f) Percentage of affected nodes in the *iForest* after every insertion for Iris dataset.

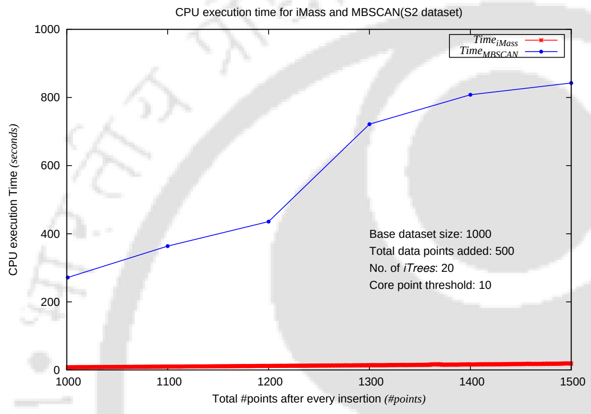
FIGURE 3.9: Efficiency comparison between *iMass* and MBSCAN along with the percentage of affected nodes due to *iMass* for datasets: Seeds, Aggregation, Iris.



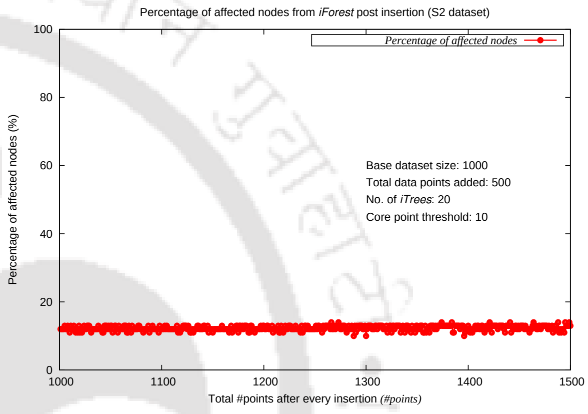
(a) Execution time for *iMass* and MBSCAN after every insertion for S1 dataset.



(b) Percentage of affected nodes in the *iForest* after every insertion for S1 dataset.



(c) Execution time for *iMass* and MBSCAN after every insertion for S2 dataset.



(d) Percentage of affected nodes in the *iForest* after every insertion for S2 dataset.

FIGURE 3.10: Efficiency comparison between *iMass* and MBSCAN along with percentage of affected nodes due to *iMass* for datasets: S1, S2.

Key Observation(s): A significant amount of gain in terms of efficiency (time required) is observed (Refer Table 3.10) while building the *iForest* for *iMass* algorithm as compared to MBSCAN.

Reason(s): The *iMass* algorithm avoids re-construction of *iForest* afresh upon insertion of any new data point. Instead once a point is inserted, the node-split criterion is used by any priorly positioned node of an *iTree* to direct the newly entered point to its further appropriate nodes. The points within unaffected nodes retain their prior position in an *iTree* without inflicting any changes. Moreover, no additional *iTrees* are added to the *iForest*.

4. **Efficiency achieved while building the mass-matrix incrementally**
Key Observation(s): A significant amount of gain in terms of efficiency

(time required) is observed (Refer Table 3.10) while building the mass-matrix for $iMass$ algorithm as compared to MBSCAN.

Reason(s): For any pair of points (x, y), the *lowest node_ids* are stored while determining the Algo-Comp values. Whenever a new point places itself into the appropriate nodes of an $iTree$, the $iMass$ algorithm accesses the *lowest node_id* of (x, y) within that $iTree$. This is done in order to check whether the new point has penetrated into that designated lowest leveled node. If the newly entered point is present within that lowest leveled node, then the mass of (x, y) w.r.t., that $iTree$ is increased by one (Refer Section 3.7.4). In this way $iMass$ avoids re-computation of mass values for any pair of existing points $(x, y) \in D$ post new insertion. Moreover from Figure 3.8(b) through Figure 3.10(d), we observe that the percentage of affected nodes including the lowest leveled nodes in the $iForest$ reaches a maximum of 27% across all datasets. These reasons collectively ensure a reduced time duration required for computing pair wise dissimilarity in the new mass-mass matrix.

3.10.3 Cluster analysis

In order to perform cluster comparison between $iMass$ and MBSCAN, we adopted the following strategy: Given a base dataset D of size n, we obtain the intermediate set of clusters dynamically after a point insertion had taken place. Suppose k points were inserted upon D one at a time, then after the entry of k^{th} point, we extracted the final set of clusters $C'(\mathcal{IM})(say)$ due to $iMass$. The updated dataset D' with $n + k$ points was fed at once into the MBSCAN algorithm to produce a set of clusters $C'(\mathcal{M})(say)$.

Experimentally, we did not observe any loss of clusters for datasets: Libras, Segment and Aggregation due to $iMass$ and MBSCAN. Based on our choice of parameters for Libras, a total of 11 clusters were produced ($\therefore |C'(\mathcal{IM})| = |C'(\mathcal{M})| = 11$) with a mean cluster size of around 7 points. For Segment dataset, a single cluster was produced upon execution of both the methods, however there was an increase in number of non-core points for the set $C'(\mathcal{M})$. Similar results were observed in case of Aggregation dataset producing a single cluster containing 788 points.

The random split-attribute (q) and split-point (p) values involved while constructing an $iTree$ implies that the $iForest$ obtained due MBSCAN and $iMass$ may not be identical. The positioning of nodes per $iTree$ impacts the mass of smallest

Table 3.10: Extent of reduction achieved for building $iForest$ and mass-matrix incrementally due to $iMass$ algorithm.

Dataset	$\lvert \boldsymbol{D'} \rvert$	***iForest***			**Mass-matrix**		
		MBSCAN (sec)	***iMass* (sec)**	**Reduction %**	**MBSCAN (sec)**	***iMass* (sec)**	**Reduction %**
Libras	241	13	0.002580	99.98	4.4741	0.4009	91.03
	270	14	0.002525	99.98	5.1014	0.4953	90.28
	300	14	0.002359	99.98	6.8980	0.6208	90.99
	320	14	0.003170	99.97	10.1717	0.7097	93.02
	360	16	0.001829	99.98	16.5059	0.9130	94.46
Segment	1541	7	0.001960	**99.97**	444.1200	13.6027	**96.93**
	1600	7	0.000650	**99.99**	604.4340	14.5633	**97.59**
	2000	7	0.000670	**99.99**	769.4430	22.516	**97.07**
	2200	7	0.000680	**99.99**	948.9450	27.3928	**97.11**
	2310	7	0.000680	**99.99**	1189.2400	30.1611	**97.46**
Wine	119	14	0.001500	99.98	0.5356	0.0985	81.60
	125	14	0.002070	99.98	0.6408	0.1079	83.15
	155	13	0.001620	99.98	1.3750	0.1701	86.96
	170	14	0.001500	99.98	1.6561	0.2021	87.79
	178	14	0.002000	99.98	2.0570	0.2134	89.62
Seeds	141	14	0.001340	**99.99**	0.8938	0.1379	**84.56**
	160	15	0.001810	**99.98**	1.3327	0.1758	**86.80**
	180	16	0.001700	**99.98**	2.1317	0.2246	**89.46**
	200	15	0.001800	**99.98**	2.6040	0.2893	**88.88**
	210	16	0.001600	**99.98**	3.1947	0.3515	**97.46**
Aggregation	526	18	0.002480	99.98	46.4284	2.4899	94.63
	550	16	0.002370	99.98	53.9519	2.7265	94.94
	650	17	0.002304	99.98	90.6784	3.8187	95.78
	750	17	0.002740	99.98	110.1890	5.1206	93.35
	788	17	0.002360	99.98	140.8830	6.1713	95.61
Iris	101	14	0.002720	**99.98**	0.3806	0.0646	**83.01**
	110	14	0.002020	**99.98**	0.4582	0.0795	**82.64**
	130	16	0.002100	**99.98**	0.7418	0.1105	**85.09**
	140	14	0.002200	**99.98**	0.0573	0.1297	**86.44**
	150	14	0.001500	**99.98**	3.1947	0.1634	**86.61**
S1	601	14	0.002180	99.98	49.1020	2.6358	94.63
	650	14	0.001820	99.98	79.1108	3.1398	96.03
	700	14	0.001810	99.98	105.3700	3.69813	96.49
	800	14	0.001990	99.98	134.9630	5.0534	96.25
	900	14	0.003370	99.97	193.9200	6.3625	96.71
S2	1001	14	0.001800	**99.98**	257.5270	7.7268	**96.99**
	1100	14	0.002930	**99.98**	349.6020	9.6862	**97.22**
	1300	15	0.001890	**99.98**	706.4120	13.6831	**98.06**
	1400	14	0.002200	**99.98**	793.8920	16.1137	**97.97**
	1500	14	0.001900	**99.98**	828.3700	17.9052	**97.83**

local region or the lowest leveled node for any pair of points. As a result, the set of core and non-core points may differ to produce different $C'(\mathcal{M})$ and $C'(\mathcal{IM})$.

On the basis of our chosen parameters, the synthetic datasets S1 and S2 retained a 50% cluster exactness for $iMass$ w.r.t., MBSCAN. S1 was reported to have a marginal decrease in the number of core points for $iMass$. There was no immediate occurrence of noise points till the entry of 300^{th} point upon a base dataset of size 600. In case of S2, the largest cluster within $C'(\mathcal{M})$ was reported to contain 933 points. The number of non-core points decreased from 11 to 2 while computing $C'(\mathcal{IM})$ for S2.

We additionally provided comparisons based on three quality measures: Normalized Mutual Information (NMI), Rand index (RI) and F1-score[7] (Refer Table 3.11) related to five class labeled data while for remaining three unlabeled data, a cluster accuracy percentage is depicted. NMI [72] provides the reduction in entropy of class labels given that the cluster labels are already known. Since NMI is normalized, it enables us to measure and compare the NMI values between different clusterings. The following formula provides the NMI measure for a given algorithm.

$$NMI = \frac{2 \cdot I(Y;C)}{H(Y) + H(C)} \tag{3.11}$$

where Y is the number of class labels, C represents the cluster labels and $H(.)$ is the entropy. $I(Y;C)$ is given by the following relation:

$$I(Y;C) = H(Y) - H(Y|C) \tag{3.12}$$

where $H(Y|C)$ represents the entropy of class labels within each cluster.

RI measures the percentage of correct decisions. Its calculation is based on the evaluation of TP (True Positive), FP (False Positive), True Negative (TN) and False Negative (FN). A TP decision allocates two similar items within same cluster. A TN puts two different items in different clusters. FP allocates two dissimilar objects to same cluster while FN assigns two similar items to dissimilar clusters. The RI is given by:

$$RI = \frac{TP + TN}{TP + FP + FN + TN} \tag{3.13}$$

The F-measure penalizes FN more than FP contrary to the RI measure. For our cluster evaluation purpose, we measure the F1-score as follows:

[7]nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-clustering-1.html

$$\begin{aligned} F1\text{-}score &= \frac{2pr}{p+r} \\ where\ \ p &= \frac{TP}{TP+FP} \\ r &= \frac{TP}{TP+FN} \end{aligned} \tag{3.14}$$

Here p and r (Equation 3.14) denote Precision and Recall respectively.

TABLE 3.11: Clusters quality evaluation metrics using Equations 3.11, 3.12, 3.13 and 3.14.

Dataset	***#Classes***	***iMass***			**MBSCAN**		
		NMI	**RI**	**F1-score**	**NMI**	**RI**	**F1-score**
Libras	15	**0.272**	**0.80438**	**0.10896**	0.272	0.80438	0.10896
Segment	7	**1.0**	**0.17848**	**0.30290**	1.0	0.17848	0.30290
Wine	3	**1.0**	0.38488	0.55583	0	**0.52105**	**0.68512**
Seeds	3	**1.0**	0.38684	0.55787	1.0	**0.38743**	**0.55848**
Iris	3	**1.0**	0.38743	**0.55848**	0.81497	**0.54598**	0.42210

For most of the class labeled datasets in Table 3.11, we observe that the *iMass* clustering algorithm either retains or has a better NMI value than that of MBSCAN. However for datasets: Wine and Seeds, MBSCAN has higher RI and F1-score as compared to *iMass*. For the other three unlabeled datasets: Aggregation, S1 and S2, a mean cluster accuracy of 60.375% was achieved.

3.11 Conclusion

In this chapter we aimed at developing an approximate incremental version of the MBSCAN clustering algorithm known as *iMass*. We intended to extract a similar set of clusters through *iMass* while achieving a higher degree of efficiency. Intelligent construction of the new mass-matrix and efficient design of *iForest* using the prior node-split criterion information enabled *iMass* to achieve the much desired efficiency. Experimentally we showed the effectiveness of *iMass* over the non-incremental MBSCAN, therefore $T_{point-ins} < T_{naive}$.

We also showed that the efficiency achieved by *iMass* was at the cost of an average memory overhead of around 66.77%. One of the prime reasons for this memory overhead was the introduction of an additional space to store the *lowest node_ids* for every pair of points per *iTree*. With the help of *lowest node_ids*, we were able to identify in constant time whether a new point has penetrated into a lowest leveled node. Moreover by retaining the exactness of clusters for certain datasets and maintaining an overall mean accuracy of about 60.375% for unlabeled data, we showed that $C'(\mathcal{IM}) \approx C'(\mathcal{M})$. For labeled data, we showed that the $iMass$ algorithm achieved similar or improved results over MBSCAN in terms of NMI, RI and F1-score thereby proving the objectives as stated in Section 3.4.

The MBSCAN [2] clustering algorithm was built on the strength of random entities: the split-attribute (q) and split-point (p) values. As a result it proved to be a major challenge to produce an exact incremental extension to MBSCAN. The construction of $iForest$ is heavily reliant on the subsample size $\mathcal{D} \subset D$ and the number of $iTrees$ (t). In our proposed algorithm $iMass$, we chose not to increase t because increasing the number of $iTrees$ against new insertions would tend to mitigate the advantages that we may derive by maintaining a consistent number of $iTrees$. The creation of internal nodes within an $iTree$ is also dependent on the random entities used by the MBSCAN algorithm. Since a prior execution of MBSCAN is performed before implementing $iMass$, at no point we can guarantee that two independent runs of the $iMass$ algorithm will produce an identical set of clusters. However, we have been able to achieve a highest efficiency upto an order of 2.28 or about 191 times across datasets due to $iMass$ which shows its worthy extension as an approximate incremental version of MBSCAN.

Chapter 4

$BISDB_{add}$: Towards Exact Incremental Clustering in Batch-Mode for Insertion using SNN-DBSCAN

In the previous chapter, we oversaw an intelligent tuning done to the expensive components of the baseline algorithm. The proposed scheme however was limited to single point insertions. Taking into consideration the repeated reconstruction of heavier algorithmic components, the point based updates may eventually not prove to be an efficient technique while dealing with a larger base dataset. Moreover, it is also desirable for an incremental algorithm to produce results identical to the naive or non-incremental approach. Therefore, the efforts laid in our first contribution motivated us to expand our research towards proposing an exact incremental solution. We chose to incrementally extend a robust density based clustering algorithm known as SNN-DBSCAN [24] (SNNDB), where updates (insertions) are made in batches to the base dataset.

We initially proposed two sub-variant algorithms viz. $Batch-Inc1$ and $Batch-Inc2$. While $Batch-Inc1$ solves only a single component of SNNDB incrementally, $Batch-Inc2$ deals with two components. Both these algorithms process insertions in batch mode leading towards the designing of most effective variant in form of $BISDB_{add}$ (Batch Incremental Shared Nearest Neighbor Density Based Clustering Algorithm for addition). The $BISDB_{add}$ algorithm targets all the components of SNNDB incrementally.

4.1 Motivation

Dynamic datasets undergo frequent changes in their size upon periodic insertion. A naive method to get an exact clustering over the changed dataset requires a redundant execution of the clustering algorithm. Moreover for minor changes in input, the variation in output is also expected to be minimal. These changes inflicted upon the dataset cannot be ignored as they might be significant for data points and their neighborhood. With increase in frequency of such updates, the problem of redundant computation may lead to efficiency and latency issues.

TABLE 4.1: Motivation behind developing the $BISDB_{add}$ clustering algorithm.

Motivation	**Description**
Redundant computation	Non-incremental algorithms fail to address the issue of redundant computation while handling dynamic datasets. They involve the entire set of data points against every new update made to the dataset.
Small frequent Updates	When minimal number of insertions are made upon a larger base dataset, the changes in clustering is also expected to be small. As a result, there is a need for designing intelligent algorithms to handle such frequent updates efficiently without redundant computation.
InSDB [1] handles pointwise addition	InSDB handles addition of points one at a time. The process may get slower as the size of base dataset increases with new insertions. This is because in order to find the affected points against every insertion, a single scan of the whole dataset is required. This scanning time is bound to rise with increase in the size of base dataset. As a result there is a need to process updates in batch mode for quickening the cluster detection process against new updates.

SNNDB [24] is a robust graph-based clustering algorithm that enables finding clusters of arbitrary shapes, sizes and densities. Existing incremental extension to SNNDB *i.e.*, IncSNN-DBSCAN [1] (InSDB) facilitates addition of data points one at a time. As a result, the process involved in rebuilding the expensive components of SNNDB against every point insertion incurs a high computational cost. To address this issue, we propose an exact incremental solution to SNNDB processing updates in batch mode. Entry of data points in batches enables faster processing of updates in one attempt. This procedure was otherwise not possible with point based insertion scheme. Table 4.1 provides a brief description about the motivation behind our work.

4.1.1 Chapter contributions

The key contribution(s) made in this chapter may be summarized as follows:

1. We propose three incremental variants of SNNDB each of which processes updates made due to addition of data points in batch mode. These three algorithms are $Batch - Inc1$, $Batch - Inc2$ and $BISDB_{add}$ (See Table 4.2). Experimentally, we observed that the third variant $BISDB_{add}$ is the most efficient as compared to the other two variants.

2. We showed the effectiveness of our fastest incremental variant $BISDB_{add}$ over SNNDB [24] while handling minimal changes made to the dataset.

3. We demonstrated the fact that when size of base dataset increases, point wise insertion of data no longer remains an effective option to detect clusters dynamically. The updates made to a larger base dataset in batch mode proves to be more efficient than both the naive (SNNDB [24]) and point-based incremental method (InSDB [1]).

4. A thorough cluster analysis is provided.

Table 4.2: Brief overview of our proposed batch incremental clustering algorithms for addition (Refer Section 4.3 for definitions of related concepts).

Algorithm	**Brief working mechanism**	**Advantage**	**Improvement**
$Batch - Inc1$	Computes the KNN lists incrementally, detects same clusters as SNNDB, performs batch wise insertion.	Reduces the time taken to compute the KNN lists post new insertions.	$Batch - Inc2$
$Batch - Inc2$	Computes the KNN lists and similarity matrix incrementally, detects same clusters as SNNDB, performs batch wise insertion.	Reduces the time taken to compute the KNN lists and construct similarity matrix post new insertions.	$BISDB_{add}$.
$BISDB_{add}$	Computes the KNN lists, similarity matrix along with core and non-core points incrementally, detects same clusters as SNNDB, performs batch wise insertion.	Reduces the time taken to compute the KNN lists, construct similarity matrix and identify core and non-core points post new insertions.	

4.2 Related work and background

Li *et al.* [73] proposed a robust structured non-negative matrix factorization (RSNMF) framework leveraging the use of $l_{2,p}$-norm loss function [74] to deal with noises in clustering. The work focused on learning a robust discriminative representation of feature sets while handling dimensionality reduction. In a $\mathcal{R}^d$ space, the phenomenon of 'curse of dimensionality' [24, 64] is a major concern due to which the proximity between data points obtained through any geometric model eg: distance becomes unreliable.

High dimensional data such as images may involve superfluous features along with the presence of noisy elements. In order to identify a subset of useful and redundancy constrained features, a non-negative spectral clustering scheme coupled with analysis of redundant features [75] was proposed. The non-negative spectral analysis technique helped learning of cluster labels related to input data more accurately. The simultaneous learning of cluster labels and attribute matrix enabled selection of the most discriminating features appropriately. Another robust clustering technique called MBSCAN [2] overcomes the limitations of distance based clustering in $\mathcal{R}^d$ space by adopting a data dependent dissimilarity measure. MBSCAN utilizes the measure of probability mass [2] instead of any geometric model for computing pairwise dissimilarity of points.

One of the building block algorithms for our proposed batch-incremental methods in this chapter is SNNDB [24]. SNNDB [24] is an amalgamation of shared nearest neighbors [52] (SNN) clustering scheme and DBSCAN [23]. SNNDB uses the concept of shared nearest neighbors to determine the proximity score between pairs of points. The similarity value between two points p and q is the number of elements the points have in common between their corresponding KNN [76] lists (Refer Section 4.3 for details). The use of SNN enables SNNDB in addressing the issues related to distance based clustering in a multi-dimensional space.

There also exists a prior incremental extension to SNNDB in form of InSDB [1]. InSDB facilitates entry of data points one at a time. This leads to repetitive construction of expensive SNNDB components against every insert. As the size of base dataset increases, the scheme of clustering adopted by InSDB may tend to become inefficient compared to batch mode processing of updates.

4.3 Preliminaries and Definitions

In this section, we define the key terms and concepts used in this contribution (Refer Table 4.3 for the meaning of notations used henceforth).

TABLE 4.3: Major notations used in this chapter (second contribution).

Notation	Description
C	Set of Clusters prior to any changes in dataset
C'	Set of Clusters after dataset is updated
D	Original (Base) dataset
D'	Changed dataset after insertions
B	Number of batches
k	No. of points per batch (inserted)
k'	Total no. of points to be inserted
K	size of the K-Nearest Neighbour list
δ_{sim}	Strong link formation threshold
δ_{core}	Core point formation threshold
$\mathcal{P}(.)$	Power set
KNN(.)	KNN list of any data point.
Sim_Mat (.)	Similarity matrix of dataset.
$Core$ (.)	Set of core points.
$Non\text{-}Core$ (.)	Set of non-core points.
$\|.\|$	Size of a set.
$adj(.)$	Set of adjacent points to a given point.

4.3.1 K-nearest neighbor (KNN) list

We define the KNN list of a data point by identifying its top-K (K $\in \mathbf{Z}^+$) closest [1] points.

4.3.2 Shared nearest neighbors (SNN)

The concept of shared nearest neighbors was first introduced by Jarvis and Patrick [52]. The SNN between two data points p and q is defined as number of points they have in common between their respective KNN lists. The SNN value is also referred to as the similarity value of p with q or vice-versa (Equation 4.1). It is defined by the following equation:

[1] For our purpose, we adopt the Euclidean distance measure.

$$similarity(p, q) = KNN(p) \cap KNN(q) \tag{4.1}$$

where $KNN\ (x)$ is the number of elements present in the KNN list of any data point x.

4.3.3 Similarity matrix or SNN graph

Similarity matrix represents the shared nearest neighbor (SNN) graph. The data points are modeled as nodes and the similarity value between them is considered to be the edge weight. An individual cell of the similarity matrix contains the edge weight or similarity value or the degree of closeness between concerned pair of points.

4.3.4 K-SNN graph

Here K represents the size of KNN list $\forall\, x \in D$ (base dataset). From the original SNN graph, an edge is retained between a pair of points p and q, only if p and q are present in each other's KNN list and the edge weight between them is greater than or equal to a certain threshold δ_{sim} (say). However, if the edge weight falls below δ_{sim}, then the link is not formed. The remaining edges which are present in the SNN graph are identified as strong links. This method of obtaining a residual graph from the original SNN graph is known as K-Nearest Neighbor Sparsification of the SNN graph [24]. We refer the SNN graph containing the nodes connected by strong links as the K-sparsified SNN graph or K-SNN graph.

4.3.5 Core and non-core points

In the K-SNN graph, if the number of strong links adjacent to a particular point exceeds a certain threshold value δ_{core} (say), then it is considered to be a core point. The remaining points are classified as non-core.

4.3.6 Noise points

The non-core points which do not share a link with any of the core points and fail obtain a cluster membership are classified as noise points.

4.3.7 Clustering

Given a dataset D, a similarity function *sim(x,y)*, and a point density function *dense(x)*, we define clustering by a mapping $f : D \to C$, where $C = \mathcal{P}\ (D)$. If x,y $\in D$, $x \neq y$ and there exists two threshold values δ_{sim}, δ_{core}, then may have the following interpretations:

1. *If sim(x,y) $\geq \delta_{sim}$, dense(x) $> \delta_{core}$ and dense(y) $> \delta_{core}$, then f(x) = f(y).*
2. *If sim(x,y) $\geq \delta_{sim}$, dense(x) $> \delta_{core}$ and dense(y) $\not> \delta_{core}$, such that $\exists z \in D$ where $x \neq y \neq z$, dense(z) $> \delta_{core}$ and sim(y,z) $\geq \delta_{sim}$. Then if sim(y,z) $>$ sim(x,y), then f(y) = f(z), otherwise if sim(x,y) $>$ sim(y,z), then f(y) = f(x).*
3. $\forall\ x \in D$, *if dense(x) $\not> \delta_{core}$ and $\nexists y \in D$ such that sim(x,y) $\geq \delta_{sim}$ and dense(y) $> \delta_{core}$, then* $\{x\} \notin C$.

As per the first point, if the degree of closeness or similarity between points x and y is greater than or equal to a threshold value δ_{sim} and x, y are dense or core points, then both x and y are a part of the same cluster.

As per the second point, the similarity between points x and y is greater than or equal to a threshold value δ_{sim} and x is core but y is non-core. There exists another core point z and the similarity between y and z is greater than or equal to δ_{sim}. In that case, if y is more similar to z than x, then points y and z belong to the same cluster. However, if y is more similar to point x than z, then y and x belong to the same cluster.

The third point states that if x is a non-core point and there exists no core point y with which x has a similarity value greater than or equal to δ_{sim}, then x is categorized as a noise point.

4.3.8 Exact Batch Incremental Clustering (Addition)

Given a dataset D alongwith its initial clustering $f : D \rightarrow C$ where $C \subseteq \mathcal{P}\ (D)$, an insertion sequence of B batches with 'k' points per batch takes place. After k' $\leq kB$ number of insertions where $kB(\text{mod } k') \equiv 0$, let D' be the updated data set. Then an incremental clustering given by a mapping $h : D' \rightarrow C'$, with $C' \subseteq \mathcal{P}\ (D')$ is isomorphic to the one time clustering $f(D')$ produced by the non-incremental algorithm.

4.4 Problem formulation

For B number of batch insertions ($B \in \mathbf{N}$) with k points/batch, let T_{naive} be the total time taken by the non-incremental method, $T_{point-ins}$ be the total time taken by the point insertion based incremental method with $k' \leq kB$ inserts, and $T_{batch-ins}$ be the total time taken by the batch incremental method. Let C_{naive}, $C_{point-ins}$ and $C_{batch-ins}$ be the respective set of clusters obtained after k' updates. If $|Mem_{batch-ins} - Mem_{naive}|$ be the difference in percentage of average memory consumed, then we aim to establish the following objectives:

1. (a) $T_{batch-ins} < T_{naive}$
   (b) $C_{batch-ins} = C_{naive}$
2. (a) $T_{batch-ins} < T_{point-ins}$
   (b) $C_{batch-ins} = C_{point-ins}$
3. $|Mem_{batch-ins} - Mem_{naive}| \leq \delta$, where δ is a small real number.

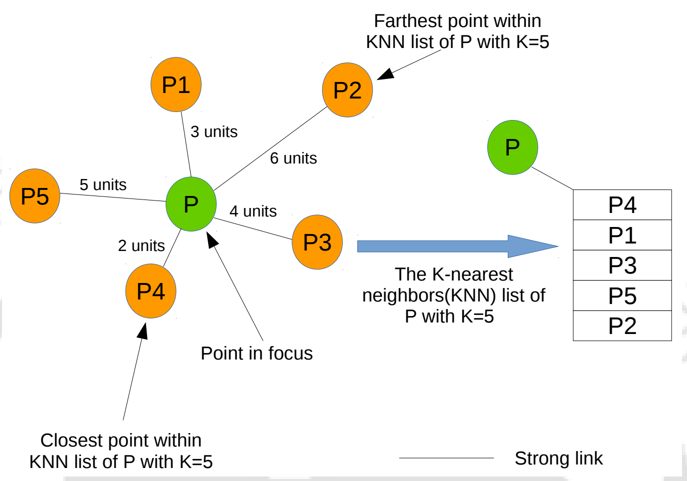
4.5 The SNNDB and InSDB clustering algorithm

The components of SNNDB [24] algorithm are:

- K-nearest neighbors (KNN).
- Shared nearest neighbors (SNN) graph.
- Set of core and non-core points.

KNN list consists of the top-K nearest neighbors for any data point. Please refer to Figure 4.1 for the representation of KNN list for data point P (say). In this figure, let the points P1, P2, P3, P4 ,P5 be at a distance of 3, 6, 4, 2 and 5 units respectively from P. Then for K = 5, the KNN list for P is the set {P4, P1, P3, P5 ,P2}.

FIGURE 4.1: KNN list for point P where K = 5


Farthest point within KNN list of P with K=5
P1
P2
3 units
6 units
P
P5
5 units
P
4 units
P3
2 units
P4
Point in focus
The K-nearest neighbors(KNN) list of P with K=5
P4
P1
P3
P5
P2
Closest point within KNN list of P with K=5
Strong link

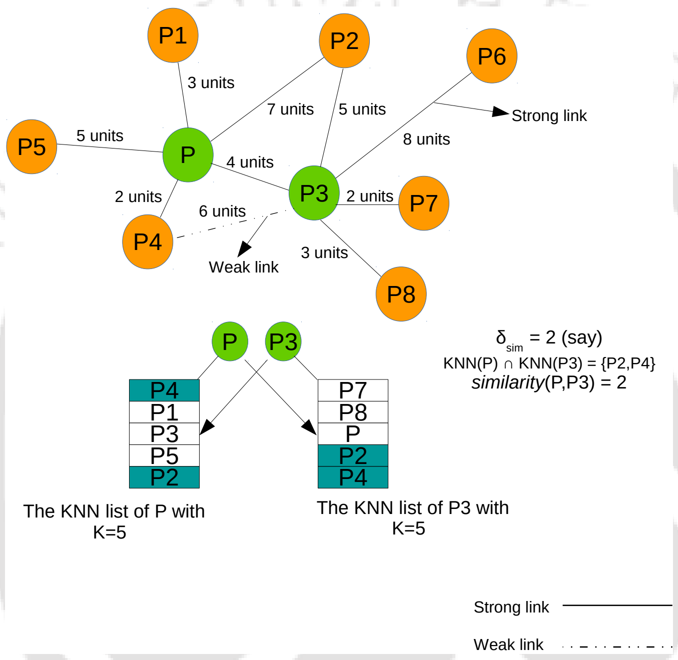
The concept of shared nearest neighbors or SNN is inherited from the clustering scheme proposed by Jarvis and Patrick [52]. The SNN clustering technique does not use any distance metric for deciding the measure of closeness between any two data points. Instead it relies on the number of shared data points between the KNN lists of any pair of points (p, q) to evaluate their proximity. The proximity score obtained is treated as the similarity value between p and q. While constructing the SNN graph, the data points are treated as nodes while the edge weight is equivalent to the similarity value between the pairs of points. This step is followed by the "K-Nearest Neighbor Sparsification" [24, 52] of SNN graph. While building a K-SNN graph, an edge is formed between any two nodes p and q iff the following two conditions are satisfied:

1. Points p and q are present in each others' KNN list.
2. The similarity value between p and q is greater than or equal to a certain threshold δ_{sim} (say).

Each of the edges constructed between any pair of points (p,q) satisfying the above two conditions are considered as strong links. Figure 4.2 demonstrates the similarity value calculation and strong link formation between two points P and

P3. The KNN list of P contains {P4, P1, P3, P5 P2} while the KNN list of P3 consists of {P7, P8, P, P2, P4}. We observe that both P and P3 are included in each others' KNN list. The proximity score or the degree of closeness between P and P3 is therefore given as 2. This is because points P and P3 share two elements: {P2,P4} between their KNN lists. If the value of δ_{sim} is set to be 2, then an edge between points P and P3 is considered to be a strong link since $|\{P2,P4\}| \geq \delta_{sim}$. [2]

FIGURE 4.2: Similarity value between points P and P3 in the K-SNN graph given that P $\in$ KNN(P3) and P3 $\in$ KNN(P) and K = 5.



The graph obtained by this mechanism is known as the K-sparsified SNN (K-SNN) graph [24, 52]. In the K-SNN graph, all the existing edges between any pair of nodes are strong links. While constructing an edge between p and q, if any one of the above two conditions is violated, an edge is not formed. All the connected components contained in the K-SNN graph are now treated as the final set of clusters by the SNN [52] algorithm.

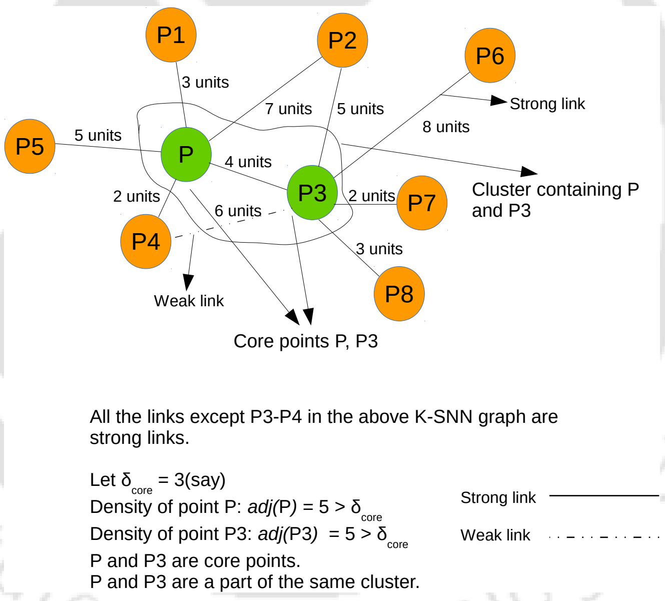
However, the SNNDB [24] algorithm produces K-SNN graph without considering its connected components as clusters. Instead, SNNDB adopts a clustering scheme similar to the DBSCAN [23] algorithm. SNNDB identifies the dense(core) and border (non-core) points to find its final set of clusters. In the K-SNN graph, for any given point p (say), SNNDB detects the number of strong links adjacent to p

[2]Having a point in the KNN list does not guarantee the formation of a shared strong link between the concerened point and its neighbor. For a shared strong link to exist, each of the two conditions for strong link formation must be satisfied.

(denoted as *adj (p)*). If *adj (p)* $> \delta_{core}$ (a certain threshold) then p is designated as a core point, otherwise p is a non-core point. The number of strong links associated with point p provides a measure of its density.

Similar to DBSCAN [23], if p and q are two core points connected by a strong link, then both these points obtain the same cluster membership (First point under **Clustering** definition from Section 4.3). However, if one of them is a non-core point, then that point is allocated to a cluster containing its nearest core point (Second point under **Clustering** definition from Section 4.3). The nearest core point is the one that shares a strong link with the concerned non-core point, and has highest edge weight as compared to other adjacent core points. The set of points which fail to obtain any cluster membership are classified as noise points (Third point under **Clustering** definition from Section 4.3).

FIGURE 4.3: Cluster containing core points P and P3 in the K-SNN graph. If δ_{core} is set as 4, then $adj(\text{P}) > \delta_{core}$ and $adj(\text{P3}) > \delta_{core}$



All the links except P3-P4 in the above K-SNN graph are strong links.

Let δ_{core} = 3(say)
Density of point P: *adj(P)* = 5 > δ_{core}
Density of point P3: *adj(P3)* = 5 > δ_{core}
P and P3 are core points.
P and P3 are a part of the same cluster.

In Figure 4.3, let us assume that the core point formation threshold (δ_{core}) is set to be 3. Now for point P, the number of adjacent strong links is five. Therefore *adj*(P) equals 5. Similarly for point P3, *adj*(P3) is also determined as 5. Points P3 and P4 share a weak link[3] which is not considered as a link. It is just given for representational purpose. Since, the density of points P and P3 exceeds the threshold value of δ_{core}, P and P3 are designated as core points. As per the DBSCAN [23] clustering scheme, points P and P3 become a part of the same cluster.

[3]A weak link is only a virtual link (dotted line) represented to show its difference with a strong link (continuous line). A K-SNN graph only consists of strong links.

IncSNN-DBSCAN [1] or InSDB is an incremental extension to the SNNDB [24] clustering algorithm. InSDB facilitates detection of clusters dynamically while points are added to the base dataset D one at a time. InSDB tags each data point $p \in D$ with the following properties: KNN list, strengths of shared strong links, number of adjacent strong links, core or non-core status. When a new data point arrives, InSDB identifies only those among old points which undergo changes in their properties. Only the affected points are targeted by the algorithm, while the unaffected points are allowed to exist in their previous state.

Let Npt be a new data point entering D. Upon entry of Npt, D changes to D'. Now, for any point $p \in D$, if p exhibits changes in its properties (as stated above), then InSDB targets p. The changes that p incurs in its properties may lead to creation of new SNN connections or removal of existing ones. New SNN connections could merge the existing clusters and their removal could split them. The selective handling of affected data points ensures that the reconstruction time of the updated KNN lists, K-SNN graph is drastically reduced. InSDB shows that a very small percentage of existing points ultimately gets affected due to which it becomes more efficient than SNNDB. However, InSDB is a point-based insertion technique, which might slow down as the size of D increases. This is because when insertions are made upon a larger base dataset, the time required to find the affected points will increase. Moreover, repetitive construction of necessary algorithmic components such as KNN lists and K-SNN graph against every insertion may slow down the overall cluster detection process.

4.6 Structure of the proposed batch incremental SNNDB clustering algorithms for addition

The generic structure of our proposed batch incremental clustering algorithms for addition is as follows:

1. We characterize each data point by its following properties:
   (a) KNN list.
   (b) SNN value or similarity value with each of the adjacent data points connected by a strong link.
   (c) Core or non-core status.

2. New data points are added[4] [71] to the base dataset in batches.
3. The updated values of properties for the batch of newly added points are computed.
4. The affected points belonging to the base dataset may undergo changes in values of at least one of their properties upon entry of new data points.
5. The data points which remain unaffected due to entry of new points do not change the values of their properties.
6. Some of the old points may change their status from core to non-core while others may change from non-core to core. The strength of shared links between data points may alter. If the link strength falls below δ_{sim}, the link gets broken.
7. The overall changed dataset consists of data points with updated property values.
8. Two connected core points are grouped into the same cluster. The cluster expansion takes place by grouping the core points accordingly. The non-core points are put into a cluster of their nearest core point. Points which fail to obtain any cluster membership are categorized as noise points.
9. The updated dataset becomes the new base dataset having an increase in its size. The updated property values for each of the data points are retained. These values are utilized for processing the next batch of arriving points. Repeat Steps 2 to 9 until the requisite number of insertions have been made.

4.7 Batch-Incremental SNNDB Clustering Algorithms for Addition

In this section, we present our proposed batch-incremental SNNDB clustering algorithms for addition viz. $Batch-Inc1$, $Batch-Inc2$ and $BISDB_{add}$. The goal of these algorithms is to extract clusters dynamically while points are added in batches to a base dataset D. Prior to executing each of these batch incremental algorithms, the naive SNNDB [24] clustering algorithm is executed upon D to obtain the following information:

[4]The order of inserting data points doesn't influence the outcome of the algorithm.

1. KNN list $\forall\, p \in D$.
2. Similarity matrix (K-SNN graph) w.r.t., D given as $Sim\_Mat(D)$.
3. Set of core and non-core points w.r.t., D given as $Core(D)$ and $Non\text{-}Core(D)$.
4. Set of clusters C w.r.t., D.
5. Set of noise points w.r.t., D.

After the data points are added to D, the changed dataset D' increases in size due to insertion of points ($|D'| > |D|$). Therefore the set $D \cap D'$ ($D \cap D' = D$) effectively represents the old points in the base dataset.

In the following subsections, we present each of our proposed batch incremental addition algorithms in a step-wise manner supported by graphical illustrations wherever necessary.

4.7.1 The $Batch - Inc1$ clustering algorithm

The $Batch - Inc1$ clustering algorithm builds the updated KNN lists of individual points in the base dataset incrementally. When new points arrive in a batch, some of the old points may get affected as they undergo change in their property values. By targeting only the affected points, the KNN list for each data point present in the base dataset is constructed. The points which remain unaffected due to new insertions retain their existing KNN lists. The new similarity matrix (updated K-SNN graph or K-SNN$_{updated}$ graph), new core and non-core points are determined non-incrementally. The steps of $Batch - Inc1$ algorithm are as follows:

1. **Step 1 - Set the parameters:** The algorithm takes three parameters: K, δ_{sim} and δ_{core}. The parameters have the following meanings:
   (a) K denotes the size of KNN list for each data point.
   (b) Given that two data points p and q are present in each others' KNN list, δ_{sim} is the minimum value of SNN required for p, q to form a strong link between them.
   (c) δ_{core} is the minimum number of strong links adjacent to a point p exceeding which p becomes a core point.
2. **Step 2 - Obtain the required data from prior SNNDB execution:**

(a) Get the base dataset D where $|D| = n$ (say).

(b) Get the KNN list $\forall\, p_i \in D, i = 1, 2, 3, \ldots, n$.

(c) Get the similarity matrix $Sim\_Mat(D)$.

3. **Step 3 - Insert a batch of new points:** Add a batch containing k new data points upon D. D changes to D' where $|D'| = n+k$.

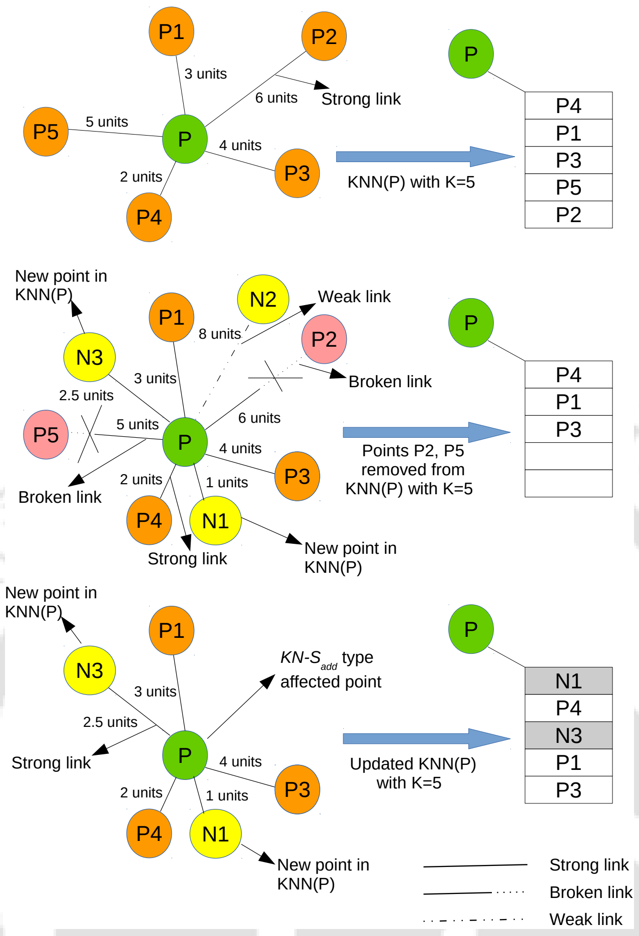
4. **Step 4 - Compute the KNN list of newly entered points:** In this step, the KNN list of all the newly added data points is computed non-incrementally. If k data points are added in a single batch, then $\forall\, p_j \in D', j = n+1, n+2, n+3, \ldots, n+k$, we find KNN($p_j$).

5. **Step 5 - Compute the updated KNN list for old data points in $D \cap D'$ incrementally:** The number of existing data points in D (base dataset) prior to any insertion is n. When k new points are added to D, D changes to D' ($|D'| = n + k$). From the set $D \cap D'$ (set of old points), the algorithm identifies those points that can accommodate any newly added point in their KNN list by replacing an old one. If the size of nearest neighbor list is K, then a maximum of K old points can be replaced by the new ones from the KNN list. The set of old points in $D \cap D'$ which contain at least one newly added point in their KNN list are categorized as $KN - S_{add}$ type affected points.

   The term $KN - S_{add}$ means that both the KNN list as well as the similarity measures of the affected data points may be altered. KN stands for change(s) in the KNN list while S signifies a possible change in the similarity values (shared link strength) of the affected data point with points in its updated KNN list (KNN$_{updated}$(.)). If the new link strength falls below δ_{sim}, the link ceases to exist further. The new points and the unaffected old points are not categorized as $KN - S_{add}$ type. The unaffected old points retain their previous KNN list. $Batch - Inc1$ therefore focuses only on re-building the KNN$_{updated}$(.) list for $KN - S_{add}$ type points. The KNN$_{updated}$(.) lists for unaffected old points in $D \cap D'$ are not constructed separately.

   Running example: Let us visit Figure 4.4 for an illustrative example of this step (Step 5). Consider the point P, where KNN(P)= {P4, P1, P3, P5, P2} (Assuming K=5) (top most image in Figure 4.4) prior to entry of any new points in the dataset. Let three new points N1, N2 and N3 (yellow color) enter the dataset. For our purpose, we consider that N1 and N3 are at a distance of 1 and 2.5 units respectively from P while N2 is at a distance

Figure 4.4: The formation of $KN - S_{add}$ type affected points upon entry of new points.



of 8 units (say). On comparing distances with other nearest neighbors of P, it is clear that the points N1 and N3 can potentially enter into the KNN list of P displacing points P2 and P5. This results in creation of two vacant slots in KNN(P) (second image in Figure 4.4). As a result, the link between pairs of points: (P,P2) and (P,P5) gets broken[5]. Consequently points N1 and N3 occupy the two vacant slots created in KNN(P).

Between N1, N2 and N3, we consider only N1 and N3 to share a strong link with P. On sorting the current set of points in increasing order of distance to point P, the updated KNN list of P (KNN$_{updated}$(P)) obtained incrementally consists of {N1, P4, N3, P1, P3} (bottom image in Figure 4.4). Point P (green color) is therefore a $KN - S_{add}$ type affected point since it accommodates new points N1, N3 in its updated KNN list. New point N2 does not have any influence over the KNN list of P and is therefore not a member

[5]The link gets broken as the points are no longer present in each others' KNN list, a necessary condition to construct a shared strong link (Refer Section 4.5).

of KNN$_{updated}$(P). The newly entered points and the old unaffected points are not classified as $KN - S_{add}$ type. The non-$KN - S_{add}$ type old points (say P1, P3 or P4) retain their previous KNN list, therefore we have the following:
KNN$_{updated}$(P1) = KNN(P1),
KNN$_{updated}$(P3) = KNN(P3),
KNN$_{updated}$(P4) = KNN(P4).

6. **Step 6 - Construct the updated K-SNN graph:** The algorithm constructs the updated K-SNN (K-SNN$_{updated}$) graph or the new similarity matrix $Sim\_Mat(D')$ non-incrementally. The updated dataset D' now consists of $n + k$ points. Therefore, $\forall\, p_j \in D', j = 1, 2, 3, \ldots, n + k$, $Batch - Inc1$ determines if a shared strong link can be constructed $\forall\, q \in$ KNN$_{updated}(p_j)$.

7. **Step 7 - Identify new core and non-core points:** For each point in K-SNN$_{updated}$ graph ($Sim\_Mat(D')$), if the number of adjacent strong links is greater than δ_{core}, the point obtains a core status otherwise a non-core. The new set of core and non-core points are stored in $Core(D')$ and $Non - Core(D')$ respectively.

8. **Step 8 - Form Clusters:** Two connected core points are placed into the same cluster. A non-core point is assigned to a cluster of its nearest core point[6].

9. **Step 9 - Discard noise points:** The non-core points which are not connected to any core point become noise points. Such points do not obtain any cluster membership.

10. **Step 10 - Retain the updated values:**
    (a) $D = D'$
    (b) $n = n + k$
    (c) $\forall\, p_i \in D, i = 1, 2, 3, \ldots, n + k$
    KNN(p_i) = KNN$_{updated}(p_i)$.
    (d) $Sim\_Mat(D) = Sim\_Mat(D')$
    (e) $Core(D) = Core(D')$
    (f) $Non\text{-}Core(D) = Non\text{-}Core(D')$

11. **Repeat Steps 3 to 10 for the next batch of entering points.**

[6]The core point with which the shared link strength is highest becomes the "nearest" core point.

4.7.2 The $Batch - Inc2$ clustering algorithm

The $Batch - Inc2$ clustering algorithm constructs the updated KNN list and the new similarity matrix (K-SNN$_{updated}$ graph) incrementally in order to further improve the efficiency over $Batch - Inc1$. The steps of $Batch - Inc2$ algorithm are as follows:

1. **Step 1 - Set the parameters:** The algorithm takes three parameters: K, δ_{sim} and δ_{core}.
2. **Step 2 - Obtain the required data from prior SNNDB execution:**
   (a) Get the base dataset D where $|D| = n$ (say).
   (b) Get the KNN list $\forall\, p_i \in D, i = 1, 2, ..., n$.
   (c) Get the similarity matrix $Sim\_Mat(D)$.
3. **Step 3 - Insert a batch of new points:** Add a batch containing k new data points upon D. D changes to D' where $|D'| = n+k$.
4. **Step 4 - Compute the KNN list of newly entered points:** Step 4 is similar to that of $Batch - Inc1$.
5. **Step 5 - Compute the updated KNN list for old data points in $D \cap D'$ incrementally:** Step 5 is similar to that of $Batch - Inc1$.
6. **Step 6 - Construct the updated K-SNN graph incrementally:** $Batch-Inc2$ introduces a new type of affected point known as the S_{add} type affected point. S_{add} type points are those which do not change their old KNN list upon entry of new data points. However they may be a part of the updated KNN list of any $KN - S_{add}$ type point. S_{add} type points may contain at least one $KN - S_{add}$ type point in their unchanged KNN list. The non-$KN - S_{add}$ type points displaced by newly entered points from the updated KNN list of a $KN - S_{add}$ type point also belong to S_{add} type[7]. The term S_{add} means that for any point $p \in D \cap D'$ (set of old points), only the value of shared strong link (edge weight) with p's adjacent points may change but KNN(p) remains the same. Therefore, for any S_{add} type affected point p, we have KNN$_{updated}(p)$ = KNN(p).

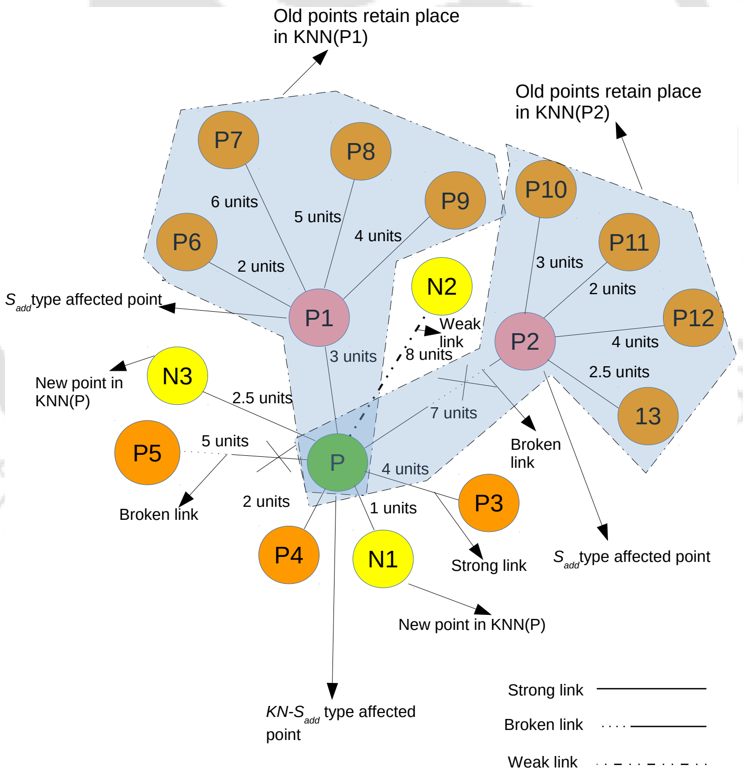
   S_{add} type points can be determined from the updated KNN list of any $KN - S_{add}$ type point. In the KNN$_{updated}$(.) list of any $KN - S_{add}$ type point, the

[7]An affected point is either $KN - S_{add}$ type or S_{add} type but not both.

old points which are non-$KN - S_{add}$ type are classified as S_{add} type. The old points which are non-$KN - S_{add}$ type and are replaced by the newly entered points are also classified as S_{add} type. New data points are neither $KN - S_{add}$ nor S_{add} type. The old unaffected points retain their original KNN list as well as the similarity values and are neither $KN - S_{add}$ nor S_{add} type.

In order to find the new state of shared strong links in the K-SNN$_{updated}$ graph, $Batch - Inc2$ scans the updated KNN lists of only the $KN - S_{add}$ and S_{add} type affected points. The existing links between the unaffected points in $D \cap D'$ are allowed to retain their prior similarity value. By targeting only the $KN - S_{add}$ and S_{add} type affected points, $Batch - Inc2$ rebuilds the entire K-SNN$_{updated}$ graph incrementally without involving the changed dataset in its totality.

FIGURE 4.5: The formation of S_{add} type affected points upon entry of new points



Running example: Figure 4.5 illustrates the formation of S_{add} type affected points upon entry of new data points. Similar to $Batch - Inc1$, let N1, N2 and N3 be the three new points entering the dataset. N1 and N3 find a place in KNN$_{updated}$(P) due to which P is classified as a $KN - S_{add}$ type point (green color). KNN$_{updated}$(P) consists of {N1, P4, N3, P1, P3} (from $Batch - Inc1$). However, if we focus on the two shaded polygons in Figure 4.5, we observe the following:

(a) Within the left polygon:
KNN$_{updated}$(P1) = KNN(P1) = {P6, P, P9, P8, P7}

(b) Within the right polygon:
KNN$_{updated}$(P2) = KNN(P2) = {P11, P13, P10, P12, P}

From the updated KNN list of P1 and P2, we observe that both the points retain their original KNN list even after the entry of new points N1, N2 and N3. Another notable observation is that P1 $\in$ KNN$_{updated}$(P) and P1 is non-$KN - S_{add}$ type, therefore P1 can be categorized as a S_{add} type point (pink color). However, P2 $\notin$ KNN$_{updated}$(P) but P2 $\in$ KNN(P). This means that point P2 has been displaced from the updated KNN list of P. Therefore P2 (a non-$KN - S_{add}$ type) also qualifies to be a S_{add} type point.

As per $Batch - Inc2$, the S_{add} type points can be determined from scanning the updated KNN list of a $KN - S_{add}$ type point. Let us revisit Figure 4.4 where we observe that for the $KN - S_{add}$ type point P, KNN$_{updated}$(P) = {N1, P4, N3, P1, P3}. Prior to insertion of new points KNN(P) contained {P4, P1, P3, P5, P2}. This means that in KNN$_{updated}$(P), the retained points are P1, P3 and P4 while the replaced points are P2 and P5. The new points are N1 and N3. Among the retained points (Refer Figure 4.5), point P1 sustains its original KNN list post entry of new points. In addition, the presence of P1 in KNN$_{updated}$(P) means that P1 may continue to share a strong link with P. Since P1 (a non-$KN - S_{add}$ type point) $\in$ KNN$_{updated}$(P), P1 qualifies to be a S_{add} type point having a shared strong link with P. For the replaced point P2 (Figure 4.5), KNN$_{updated}$(P2) = KNN(P2). However, P2 being a non-$KN - S_{add}$ type point is also categorized as a S_{add} type point without a shared strong link since P2 $\notin$ KNN$_{updated}$(P).

7. **Step 7 - Identify new core and non-core points:** Step 7 is similar to that of $Batch - Inc1$.

8. **Step 8 - Form Clusters:** Step 8 is similar to that of $Batch - Inc1$.

9. **Step 9 - Discard noise points:** Step 9 is similar to that of $Batch - Inc1$.

10. **Step 10 - Retain the updated values:**

(a) $D = D'$

   (b) $n = n + k$

   (c) $\forall\, p_i \in D, i = 1, 2, 3, \ldots, n + k$
   $\mathrm{KNN}(p_i) = \mathrm{KNN}_{updated}(p_i)$.

   (d) $Sim\_Mat(D) = Sim\_Mat(D')$

   (e) $Core(D) = Core(D')$

   (f) $Non - Core(D) = Non - Core(D')$

11. **Repeat Steps 3 to 10 for the next batch of entering points.**

4.7.3 The $BISDB_{add}$ clustering algorithm

The $BISDB_{add}$ clustering algorithm builds the updated KNN list, the updated K-SNN graph and the new set of core and non-core points incrementally. With each component of SNNDB being handled incrementally, $BISDB_{add}$ attempts to further reduce the computational cost involved as compared to $Batch - Inc1$ and $Batch - Inc2$. The steps of the $BISDB_{add}$ algorithm are as follows:

1. **Step 1 - Set the parameters:** The algorithm takes three parameters: K, δ_{sim} and δ_{core}.

2. **Step 2 - Obtain the required data from prior SNNDB execution:**

   (a) Get the base dataset D where $|D| = n$ (say).

   (b) Get the KNN list $\forall\, p_i \in D, i = 1, 2, 3, \ldots, n$.

   (c) Get the similarity matrix $Sim\_Mat(D)$.

3. **Step 3 - Insert a batch of new points:** Add a batch containing k new data points upon D. D changes to D' where $|D'| = n{+}k$.

4. **Step 4 - Compute the KNN list of newly entered points:** Step 4 is similar to that of $Batch - Inc1$ and $Batch - Inc2$.

5. **Step 5 - Compute the updated KNN list for old data points in $D \cap D'$ incrementally:** Step 5 is similar to that of $Batch - Inc1$ and $Batch - Inc2$.

6. **Step 6 - Construct the updated K-SNN graph incrementally:** Step 6 is similar to that of $Batch - Inc2$.

7. **Step 7 - Identify new core and non-core points incrementally:** In the K-SNN$_{updated}$ graph, for each of the total number of $KN - S_{add}$ and S_{add} type points in $D \cap D'$, $BISDB_{add}$ checks whether the number of strong links adjacent to the affected point exceeds δ_{core}. If this happens, the concerned point is treated as a core point or else it is a non-core point. The remaining points retain their existing core or non-core status from the previous iteration[8].

8. **Step 8 - Form Clusters:** Step 8 is similar to that of $Batch - Inc1$ and $Batch - Inc2$.

9. **Step 9 - Discard noise points:** Step 9 is similar to that of $Batch - Inc1$ and $Batch - Inc2$.

10. **Step 10 - Retain the updated values:**

    (a) $D = D'$

    (b) $n = n + k$

    (c) $\forall\, p_i \in D, i = 1, 2, 3, \ldots, n + k$
    KNN(p_i) = KNN$_{updated}$(p_i).

    (d) $Sim\_Mat(D) = Sim\_Mat(D')$

    (e) $Core(D) = Core(D')$

    (f) $Non - Core(D) = Non - Core(D')$

11. **Step 11 - Repeat Steps 3 to 10 for the next batch of entering points.**

4.7.4 Shared link properties between affected points post insertion

Any change in the KNN list of an affected point may lead to a deviation in the values of associated shared strong links. We present all possible scenarios of the state of shared strong links between $KN - S_{add}$ and S_{add} type affected points.

1. $KN - S_{add} \leftrightarrow KN - S_{add}$ **link:** With entry of new points in the updated KNN list of $KN - S_{add}$ type points, some of the old points may get replaced.

[8]For the first batch of arriving points, the core or non-core status of a point in $D \cap D'$ is derived from initial SNNDB execution upon D.

ALGORITHM 2: $BISDB_{add}(D, \text{K}, \delta_{sim}, \delta_{core})$

```
1  Input: $D$, K , $\delta_{sim}, \delta_{core}$;
2  Output: $Clusters$;
   // Set nrow as the total no. of data points after increment of
      nrow2 points upon nrow1
3  $nrow \leftarrow nrow1 + nrow2$;
   // Update dataset after increment
4  for $i1 \leftarrow 1$ to $nrow2$ do
5      Append new data point i1 to base dataset data_matrix[];
6      $i \leftarrow$ i+1;
   // Find KNN list of new points
7  for $i \leftarrow 1$ to $nrow1$ do
8      for $i1 \leftarrow nrow1$ to $nrow$ do
9          Compute the distance between data points i1 and i ;
10         if $i <= K$ then
11             Insert data point i to the KNN list of data point i1 ;
12         else
13             Insert data point i next to the KNN list of data point i1 ;
14             sort $(KNN\_matrix[i1])$;
15             $KNN\_matrix[i1].pop()$;
   // Find points that can be potentially affected
16 for $i1 \leftarrow nrow1$ to $nrow$ do
17     for $j1 \leftarrow nrow1$ to $nrow$ do
18         if $distance(i1, j1|i1 \neq j1) <= distance(i1, KNN\_matrix[i1][K])$ then
19             Insert data point k to the KNN list of data point i ;
20             sort $(KNN\_matrix[i1])$;
21             $KNN\_matrix[i1].pop()$;
22         else
23 for $i \leftarrow 1$ to $nrow1$ do
24     for $k \leftarrow nrow1$ to $nrow$ do
25         Compute distance between i and k;
26         if $distance(i, k|i \neq k) <= distance(i, KNN\_matrix[i][K])$ then
27             Insert data point k to the KNN list of data point i ;
28         else
29             Do nothing;
   // Identify $KN - S_{add}$ and $S_{add}$ type affected points
30 for $i \leftarrow 1$ to $nrow$ do
31     if $KNN\_list[i].size() > K$ then
32         $i \in KN - S_{add}$ type points;
33     else
```

```
34 for each $i \in KN - S_{add}$ do
35     for each $j \in KNN\_matrix[i] \cup$ points displaced from $KNN\_matrix[i]$
        do
36         if $j \notin KN - S_{add} \wedge j$ is not a new point then
37             $j \in S_{add}$ type points;
38         else
               –
   // Construct the updated K-SNN graph and detect core, non-core
      points incrementally
39 for each $i \in KN - S_{add} \cup S_{add}$ do
40     for each $j \in KNN\ \_matrix[i]$ do
41         if $similarity(i, j) > \delta_{sim}$ then
42             An edge is formed between pints i and j;
43         else
               –
44     if $similarity\_matrix[i].size() > \delta_{core}$ then
45         $i \in CORE$ points set;
46     else
47         $i \in Non\text{-}CORE$ points set;
48 Cluster formation is similar to the SNNDB algorithm;
49 Repeat entire process for the next batch of entering points;
```

If the removed points contributed to the shared link strength, then the similarity value is bound to decrease. However, if the replaced points were not a part of the contributory set to shared link strength, then the similarity value remains same. Moreover, if the newly added points lie in the common neighborhood of two linked data objects ensuring replacement of the non-contributory points to their shared link strength, then in that case the inserted points add to the similarity value of the concerned pair. Therefore, for a $KN - S_{add} \leftrightarrow KN - S_{add}$ type link, the strength of shared link either decreases, remains same or increases.

2. $KN - S_{add} \leftrightarrow S_{add}$ **link:** The S_{add} type points do not change their KNN list. Therefore if the removed points from the KNN list of any $KN - S_{add}$ type point previously contributed to the shared link strength with a S_{add} type point, then the strength of shared link is bound to decrease. However, if the replaced points were not a part of the contributory set to shared link strength, then the similarity value remains identical for a $KN - S_{add} \leftrightarrow S_{add}$ type link.

3. $S_{add} \leftrightarrow S_{add}$ **link:** With no change in the KNN list for S_{add} type points post new insertions, the points which originally contributed to the shared link

strength remain unaffected. As a result no change in shared link strength is observed for a $S_{add} \leftrightarrow S_{add}$ type link.

4.7.5 Summary of the batch-incremental SNNDB clustering algorithms for addition

TABLE 4.4: Summary of the batch-incremental SNNDB clustering algorithms for addition

Components-Algorithm	$Batch - Inc1$	$Batch - Inc2$	$BISDB_{add}$
Updated KNN list	**Incrementally**	**Incrementally**	**Incrementally**
Updated K-SNN graph	**Non-Incrementally**	**Incrementally**	**Incrementally**
Updated core and non-core points	**Non-Incrementally**	**Non-Incrementally**	**Incrementally**

In an attempt to improve the efficiency over SNNDB while handling dynamic insertion, we initially propose the $Batch-Inc1$ algorithm. $Batch-Inc1$ computes the updated KNN list of all the data points incrementally while rest of the components are computed similar to SNNDB. In order improve upon $Batch-Inc1$, we propose $Batch - Inc2$ which rebuilds both the updated KNN lists and the K-SNN$_{updated}$ graph upon entry of new data points incrementally. The third algorithm in form of $BISDB_{add}$ computes all the three components of SNNDB incrementally. This involves detection of core and non-core points apart from constructing KNN list and updated K-SNN graph (Refer Table 4.4).

The SNNDB method takes $\mathcal{O}(N^2)$ time towards completion where N is the total number of data points. This is mainly due the construction of similarity matrix and KNN lists. $Batch - Inc1$ provides marginal improvement by building the updated KNN lists incrementally in $\mathcal{O}(N)$ time. However, building the K-SNN$_{updated}$ graph non-incrementally involves quadratic time complexity. $Batch - Inc2$ aims to address this issue by reconstructing the K-SNN$_{updated}$ graph incrementally upon entry of new data points. While building the K-SNN$_{updated}$ graph, $Batch - Inc2$ only updates the shared strong link strengths of $KN - S_{add}$ and S_{add} type points leaving rest of the unaffected points. For identifying the new core and non-core points, $Batch - Inc2$ involves all the data points in D' (updated dataset). This results in $Batch - Inc2$ having a linear time complexity. $BISDB_{add}$ identifies the new set of core and non-core points incrementally and therefore improves upon the previous two sub-variant algorithms for addition. $BISDB_{add}$ also runs in linear time (Refer Algorithm 2 for pseudo-code of $BISDB_{add}$). Next we present the time complexity analysis of the $BISDB_{add}$ algorithm.

4.8 Time complexity analysis of the $BISDB_{add}$ clustering algorithm

Let D be the base dataset where $|D| = n$ ($n \in \mathbf{Z}^+$) be its initial size. Let D' be the updated dataset after new insertions. Let us assume that a total of B batches are inserted with k points per batch ($k \ll n$). For the B^{th} batch arrival at any point in time, we have $|D'| = n + kB$, where the current size of base dataset is $n + (B-1)k = N$ (say). The size of each entering batch is significantly smaller than the current size of base dataset, $\therefore k \ll N$.

We provide the time complexity analysis by assuming that the first batch of insertions had been made ($\therefore B = 1$). [Line 7-15]: While updating the KNN list dynamically, the inner loop varies through total number of newly added points in a single batch of size k. The inner loop finds the KNN list for each of the k newly added points. The time taken to compute the distance between a new point and an old point is $\mathcal{O}(1)$. Initially for each iteration a running time of $\mathcal{O}(k^2)$ is required. Since $k \ll N$, k^2 may be treated as a constant. However, we cannot guarantee that the current set of points which occupy the KNN window for each of the new points are its actual top K points. This is because the top K slots in the KNN list for new points are presently filled by the k old data points. In order to detect the actual KNN list for a newly entered point, $BISDB_{add}$ scans through the remaining N-K old data points along with $(k-1)$ newly entered points. With K being the size of KNN list and K$\ll N$, the KNN lists of the newly added points are constructed in linear time.

[Line 16-29]: For computing the updated KNN list of the older points, $BISDB_{add}$ identifies each point $p \in D \cap D'$ (set of old points) that are affected by the entry of new data points belonging to $(D' - D)$. The affected points update their new KNN list while rest of the old points retain their previous KNN list. This particular operation incurs a running time of $\mathcal{O}(N \cdot k)$.

[Line 30-38:] The old points which alter their KNN list are the $KN - S_{add}$ type affected points. $BISDB_{add}$ identifies the S_{add} affected type points from the KNN list of the $KN - S_{add}$ type points. This operation takes 2K$\cdot\mathcal{O}(|KN - S_{add}|) \simeq$

$\mathcal{O}(1)$ time, given that both K and $KN - S_{add}$ represent constant entities and are considerably smaller than $|D'|$ (See details of proof in Section 4.8.1).

[Line 39-48]: The shared nearest neighbor graph is constructed by targeting the $KN - S_{add}$ and S_{add} type points. This segment of $BISDB_{add}$ has a running time of $\mathcal{O}((|KN - S_{add}|+|S_{add}|)\cdot 2\text{K}(\text{K}+c'\log_2\text{K})) \simeq \mathcal{O}(1)$ since $|KN - S_{add}|, |S_{add}|, c'$ and K are small constant entities (See details of proof in Section 4.8.1). The detection of core and non-core points happens in constant time while clusters are extracted in linear time.

4.8.1 Comparing time complexity proofs of SNNDB and $BISDB_{add}$

- **SNNDB** [24]: Let T_{KNN} = time taken to compute the KNN lists of N data points in base dataset D, then

$$T_{KNN} = N(N-1) \tag{4.2}$$

Let T_{Edge} = time taken to construct an edge between any pair of points in D, then

$$T_{Edge} = 2c'\Big(\log_2 K + \frac{K}{2}\Big) \qquad [where\ c' \in \mathcal{R}^+, c' \ll N, K \ll N] \tag{4.3}$$

where K is the size of the KNN list. The term $2 \cdot c' \log_2 K$ refers to the time taken to search the presence of both data points in each others' KNN list while $c'K$ time is needed to find the similarity measure between any pair of data points[9].

Let T_{K-SNN} = time taken to construct the updated K-SNN graph.

$$\begin{aligned}\therefore T_{K-SNN} &= \frac{N^2}{2}\cdot T_{Edge}\\ \implies T_{K-SNN} &= \frac{N^2}{2}\cdot 2\cdot c'\Big(\log_2 K + \frac{K}{2}\Big)\\ \implies T_{K-SNN} &= N^2\cdot c'\Big(\log_2 K + \frac{K}{2}\Big)\ [Using\ Equation\ 4.3]\end{aligned} \tag{4.4}$$

[9]The KNN list of any data point is sorted in increasing order of distances with its top-K neighboring points.

Let $T_{Core+Non-Core}$ be the time taken to identify core and non-core points,

$$\therefore T_{Core+Non-Core} = N \tag{4.5}$$

A linear scan of the dataset is sufficient to identify points with more than δ_{core} number of strong links adjacent to it. Let $T_{Cluster}$ be the time taken to extract clusters,

$$\therefore T_{Cluster} = N \cdot K \tag{4.6}$$

Now if T_{SNNDB} be the total time taken by the non-incremental SNNDB [24] algorithm, then (using Equations 4.2, 4.4, 4.5, 4.6) we obtain the following:

$$\begin{aligned} T_{SNNDB} &= T_{KNN} + T_{K-SNN} + T_{Core+Non-Core} \\ &\quad + T_{Cluster} \\ &= cN^2 + KN \quad [K \ll N] \\ &\quad [where\ c = 1 + c'\frac{K}{2} + c' \log_2 K] \\ &\quad [c', K\ are\ constants, \therefore c\ is\ constant] \\ \implies T_{SNNDB} &\simeq \mathcal{O}(N^2) \end{aligned} \tag{4.7}$$

The term $c' \log_2 K$ is used for searching the presence of a point in another point's KNN list. This operation is mandatory in order to validate the strong link formation criterion.

- $BISDB_{add}$: Let n be the size of base dataset and k be the number of points added per batch. We deduce the running time of $BISDB_{add}$ when a single batch insertion has been made. Let N be the total number of points after a batch insertion ($\therefore N = n + k$).

$$\begin{aligned} T_{KNN} &= k(n + k - 1) + kn \\ &= k(N - 1) + k(N - k) \quad [where\ k \ll N] \\ &= k(2N - k - 1) \\ &= 2kN - k(k + 1) \end{aligned} \tag{4.8}$$

The term $k(n + k - 1)$ depicts the time required in filling the KNN list for k newly entered points. While the term kn signifies the time required to detect if any of the new points penetrate into the KNN list of old points. Effectively kn amount of time is required to find the $KN - S_{add}$ type points, $\therefore T_{KN-S_{add}} = kn$.

Let $T_{S_{add}}$ = time taken to find the S_{add} type points. Since S_{add} type points are determined from the updated KNN list of the $KN - S_{add}$ type points, we have the following equation:

$$\begin{aligned} T_{S_{add}} &= |KN - S_{add}| \cdot 2K \\ &\qquad [where\ |KN - S_{add}| \ll N] \end{aligned} \tag{4.9}$$

The term $2K$ is used in Equation 4.9 because the displaced points from the updated KNN list of any $KN - S_{add}$ type point are also checked for their S_{add} status. At most K number of points can be displaced from the updated KNN list of a $KN - S_{add}$ type point.

$$\begin{aligned} T_{K-SNN} &= (|KN - S_{add}| + |S_{add}|) \cdot 2c'\left(\frac{K}{2} + \log_2 K\right) \\ &\qquad [where\ |S_{add}| \ll N] \end{aligned} \tag{4.10}$$

For calculating T_{K-SNN}, only $KN - S_{add}$ and S_{add} type points are taken into consideration, while rest of the points remain untouched. The term $2c'(\frac{K}{2} + \log_2 K)$ depicts the time required to construct or break a possible strong link for a $KN - S_{add}$ or S_{add} type point with each data item in their updated KNN list.

$$T_{Core+Non-Core} = |KN - S_{add}| + |S_{add}| \tag{4.11}$$

For finding the core and non-core points incrementally, the algorithm checks only the $KN - S_{add}$ and S_{add} type points while the unaffected points retain their previous core or non-core status.

$$T_{Cluster} = K \cdot N \tag{4.12}$$

Now if $T_{BISDB_{add}}$ be the total time taken by the $BISDB_{add}$ algorithm, then (using Equations 4.8, 4.10, 4.11, 4.12) we obtain the following:

$$\begin{aligned}
\therefore T_{BISDB_{add}} &= T_{KNN} + T_{K-SNN} + T_{Core+Non-Core} \\
&+ T_{Cluster} \\
&= N(K+2k) - c_1 + c_2 \\
&[Here\ k, K, |KN - S_{add}|, |S_{add}|\ are\ constants] \\
&[\therefore c_1, c_2\ are\ constants] \\
\implies T_{BISDB_{add}} &\simeq \mathcal{O}(N)
\end{aligned} \tag{4.13}$$

$$Here\ c_1 = k(k+1)\ and\ c_2 = (|KN - S_{add}| + |S_{add}|) * (1 + 2c'(\tfrac{K}{2} + \log_2 K))$$

On comparing Equation 4.13 with Equation 4.7, we observe that $BISDB_{add}$ is more efficient than SNNDB.

4.9 Experimental evaluation

We performed experiments on three real world and two synthetic datasets (Table 4.5) to prove the efficiency of $BISDB_{add}$ over SNNDB [24] and InSDB [1].

TABLE 4.5: Datasets description.

Dataset	Size	#Attributes	Description
Mopsi2012	13000	2	Search locations in Finland
5D	100000	5	Synthetic dataset
Birch3	100000	2	Gaussian clusters
KDDCup'04	60000	70	Identifying homologous proteins to native sequence
KDDCup'99	54000	41	Network intrusion detection data

The real world datasets were Mopsi2012 (or Mopsi12)[10], KDDCup'99[11] and KDD-Cup'04[12] while the synthetic datasets were: 5D points set (synthetically generated) and Birch3[13]. We simulated our algorithms in C++ on a Linux platform (Intel (R) Xeon (R) CPU E5530 @ 2.40GHz) with 32GB RAM.

The experiments were conducted in following phases.

1. Phase-1: Finding the most effective batch incremental variant (addition).

[10] https://cs.joensuu.fi/sipu/datasets/
[11] http://archive.ics.uci.edu/ml/index.php
[12] http://archive.ics.uci.edu/ml/index.php
[13] https://cs.joensuu.fi/sipu/datasets/

2. Phase-2: Prove efficiency of the most effective batch incremental variant (addition) over InSDB [1].
3. Phase-3: Show that InSDB [1] becomes ineffective when larger updates (addition) are made to the base dataset.
4. Phase-4: Prove efficiency of the most effective batch incremental variant (addition) over SNNDB [24].

4.9.1 Phase-1: Finding the most effective batch incremental variant (addition)

We adopted three datasets: Mopsi2012, 5D points set and Birch 3 to conduct experiments in this phase. For our experimental purpose, we defined a new term called Algorithm-Components (AlgoComp). AlgoComp consisted of following components:

- Base dataset
- KNN lists
- Similarity matrix
- Core and non-core points
- Clusters

A base dataset is one taking which SNNDB [24] is executed in order to set the values of other components in AlgoComp. Initially, the same base dataset is fed as input to the batch incremental algorithms (addition). After processing a certain batch of updates, the base dataset alters its size. The new set of points becomes the updated base dataset over which the next batch of incoming points is processed. The new KNN lists, similarity matrix (SNN graph), core and non-core points along with clusters become a part of the updated AlgoComp.

Next we describe the experiments carried out for comparing the proposed batch incremental clustering algorithms: $Batch - Inc1$, $Batch - Inc2$ and $BISDB_{add}$ dataset wise.

1. **Mopsi2012 (Addition):** The values of parameters were set as: K = 10, δ_{sim} = 3 and δ_{core} = 4. Initially, the size of base dataset was taken to be 10000. We executed the SNNDB algorithm upon base dataset to initialize the AlgoComp values. A total of 3000 points were inserted in multiple batches with the batch size varying from 2 to 50 processing between 1500 and 60 batches.

FIGURE 4.6: Mopsi2012 dataset: Efficiency comparison between batch incremental algorithms (addition).

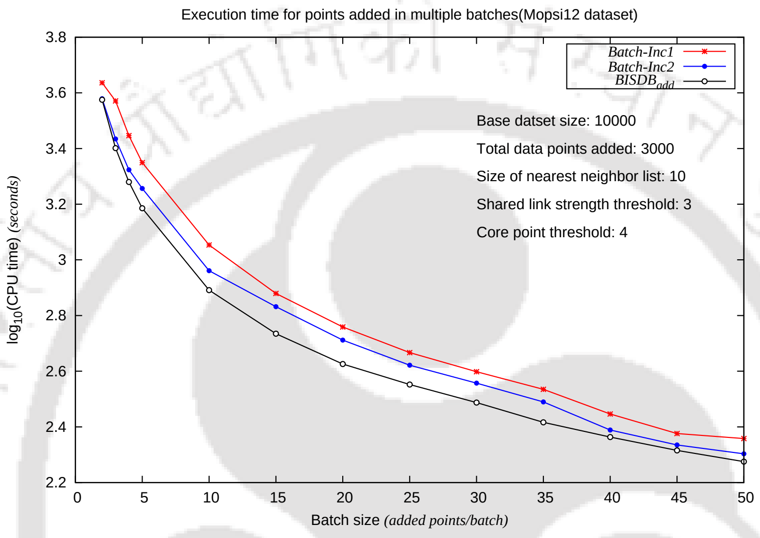
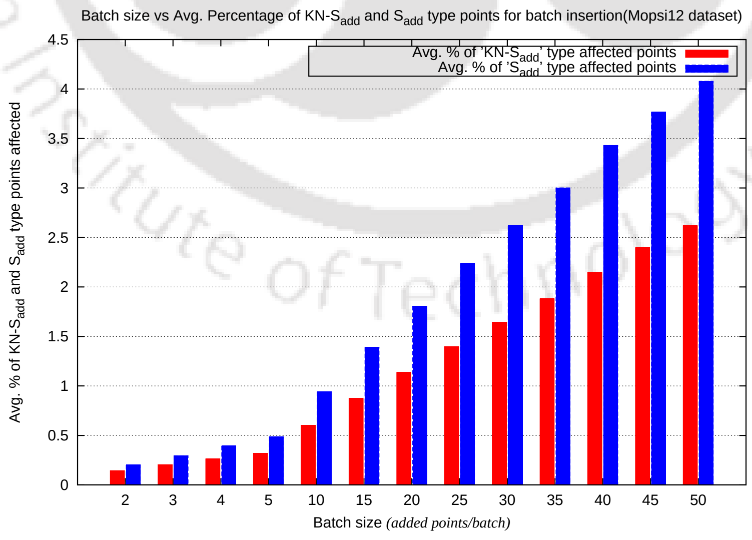
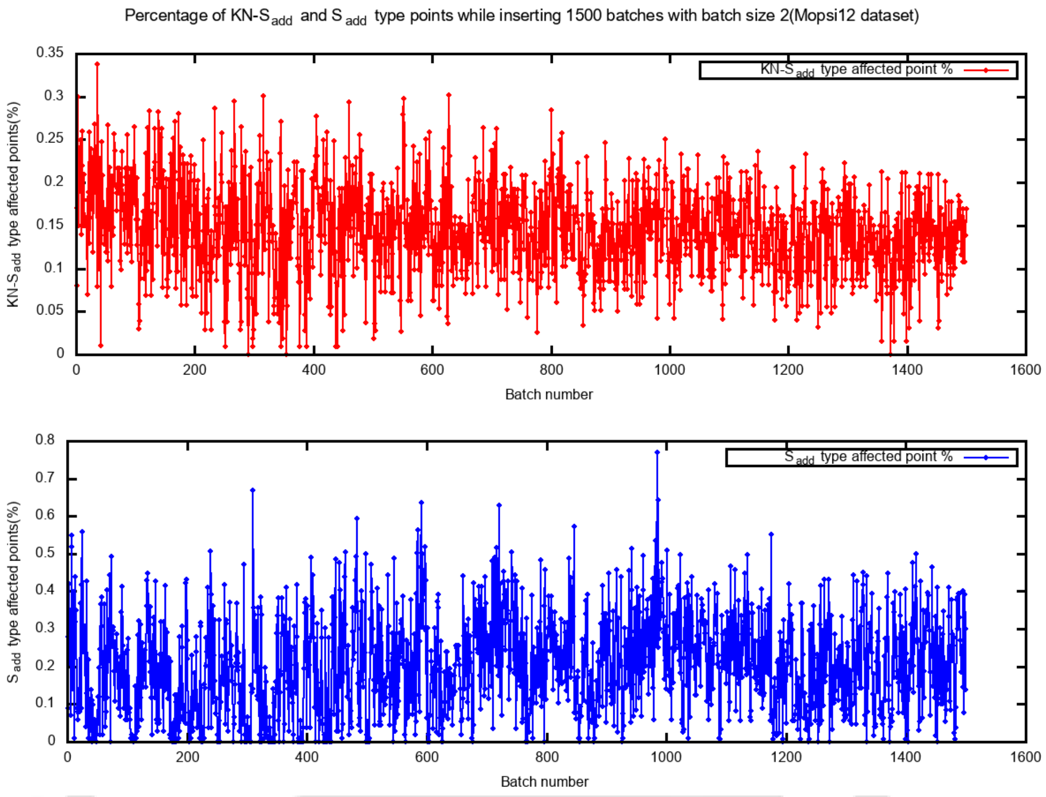


FIGURE 4.7: Mopsi2012 dataset: Average percentage of $KN - S_{add}$ and S_{add} type points for multiple batch insertions of varying batch size ($BISDB_{add}$).



For processing the first batch of entering points, the AlgoComp values obtained from executing SNNDB is utilized. For a batch of size 2, we executed each of the three incremental algorithms independently 1500 times to insert

FIGURE 4.8: Mopsi2012 dataset: Percentage of $KN-S_{add}$ and S_{add} type points created while adding 1500 batches ($BISDB_{add}$).



3000 points. After every two points were inserted, the algorithms computed the new clusters incrementally using the pre-computed AlgoComp values from previous batch insertion. The final set of clusters that were produced after inserting 1500 batches, were identical to those of the SNNDB method which processed 13000 points at once. The same set of clusters were also produced by InSDB after inserting 3000 points one at a time (Refer Section 4.10 for details). Similarly, for a batch consisting 50 points, each of the three incremental variants (addition) processed 600 batches sequentially. Out of the three incremental algorithms, $BISDB_{add}$ performed most efficiently. For various batch sizes (varying from 2 to 50), $BISDB_{add}$ maintained a better efficiency than the other two incremental variants (Refer Figure 4.6).

The incremental algorithms targeted only the $KN-S_{add}$ and S_{add} type points while retaining the properties of remaining data points. This selective handling of data is the main reason behind improving the efficiency of $BISDB_{add}$ over SNNDB (as shown in Phase-4). We experimentally observed the percentage of $KN - S_{add}$ and S_{add} type affected points for the step where 1500 batches were inserted with a batch of size 2. A maximum of around 0.34% $KN - S_{add}$ type and about 0.76% of S_{add} type points were produced while processing batch number 36 and 985 respectively (Refer Figure 4.8). An insignificant percentage of $KN - S_{add}$ and S_{add} type affected points against every batch batch insertion meant that $BISDB_{add}$ would outperform the SNNDB [24] algorithm. We additionally observed that the mean percentage of $KN - S_{add}$ and S_{add} type points per batch reached a maximum of around 2.6% and 4% respectively for a batch of size 50 while adding 600 batches sequentially (Figure 4.7).

List of key observation(s) and reason(s) for Mopsi12 dataset (Addition):

- **Key Observation:** CPU execution time for batch-incremental algorithms reduces with increase in batch size.
  Analysis/Reason(s): With increase in batch size, the total number of batches to be processed decreases. The overall reconstruction time for K-SNN graph, KNN lists, detecting core and non-core points decreases.

- **Key Observation:** $BISDB_{add}$ achieves the maximum efficiency.
  Analysis/Reason(s): Constructing updated KNN lists, K-SNN graph, finding core and non-core points incrementally.

- **Key Observation:** Average percentage of $KN - S_{add}$ and S_{add} type points increases with increasing batch size. However the average percentage of $KN - S_{add}$ type points is less than that of S_{add} type points.
  Analysis/Reason(s): With increase in batch size, the number of old points affected due to any batch insertion increases. Given any batch update, the number of points in $D \cap D'$ changing their previous KNN list is less than those affected points which do not change their KNN list. Therefore the average percentage of $KN - S_{add}$ type points is less than that of S_{add} type points.

- **Key Observation:** Percentage of $KN - S_{add}$ or S_{add} type points while processing 1500 batches (addition) remains less than 1%.
  Analysis/Reason(s): The number of points altering their KNN list upon new insertions or the points forming new links and breaking existing ones are less than 1% of the total number of points in updated base dataset across all batch updates.

2. **5D (Addition):** The parameters for execution were set as: K = 4, $\delta_{sim} = 2$ and $\delta_{core} = 2$. The base dataset consisted of 80000 points over which 20000

points were added. The AlgoComp values were computed by performing SNNDB on the base dataset. Initially, we added 2 points per batch while processing 10000 batches sequentially. Similarly, a maximum of 50 points per batch were added to process 400 batches. Similar to Mopsi12, we observed that with increase in batch size the time required to process the updates got reduced. Our observation illustrates the efficiency of $BISDB_{add}$ over other two incremental methods: $Batch-Inc1$ and $Batch-Inc2$ for various batch insertions (Figure 4.9).

FIGURE 4.9: 5D dataset: Efficiency comparison between batch incremental algorithms (addition).

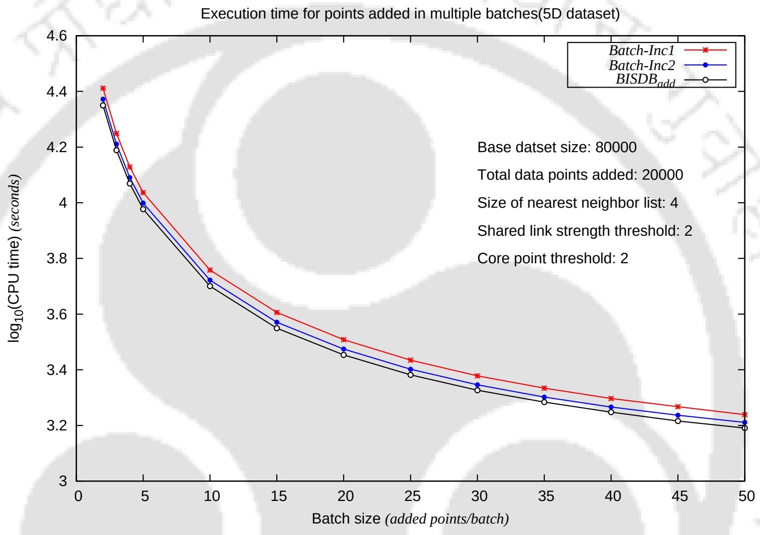
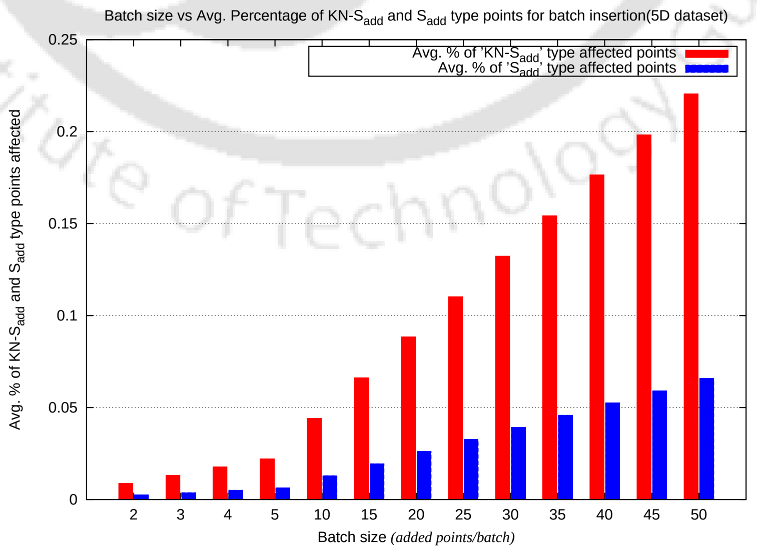
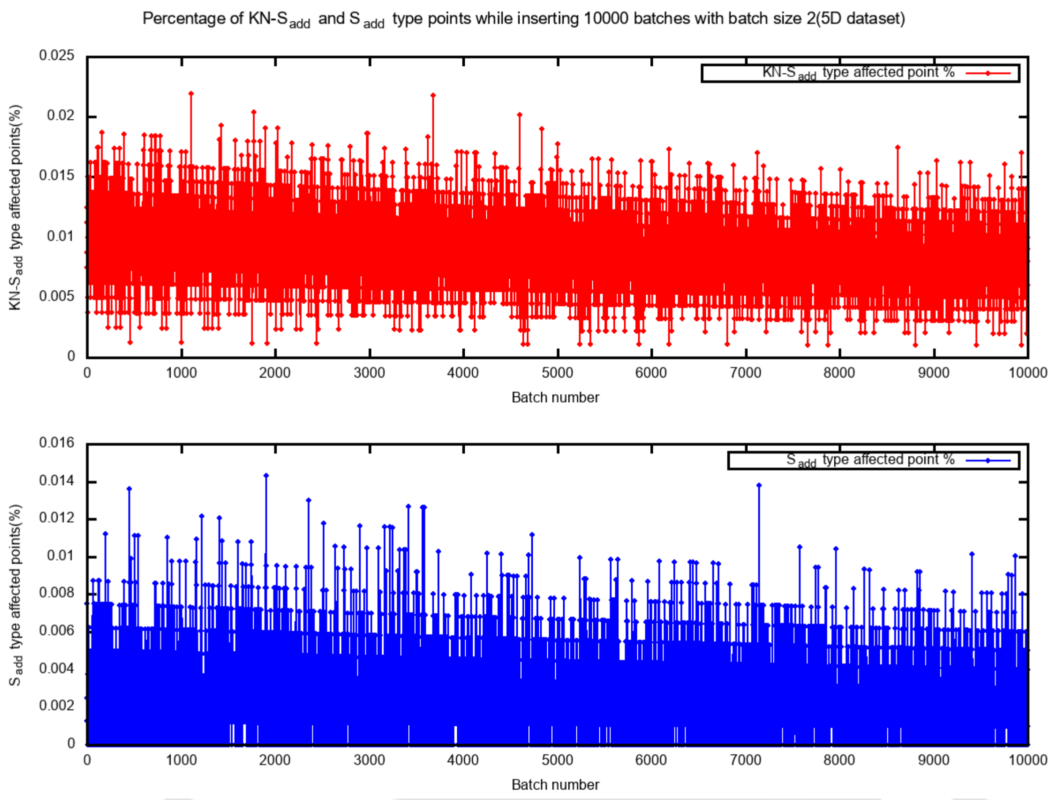


FIGURE 4.10: 5D dataset: Average percentage of $KN - S_{add}$ and S_{add} type points for multiple batch insertions of varying batch size ($BISDB_{add}$).



While adding 10000 batches, a maximum of around 0.0218% of $KN - S_{add}$ type and about 0.0143% of S_{add} type affected points were identified while

FIGURE 4.11: 5D dataset: Percentage of $KN - S_{add}$ and S_{add} type points created while adding 1500 batches ($BISDB_{add}$).



processing batch number 1104 and 1902 respectively (Figure 4.11). The mean percentage of $KN - S_{add}$ and S_{add} type points per batch reached only around 0.22% and 0.06% for batches of size 50 (Figure 4.10).

List of key observations and reasons (Addition) for 5D dataset:

- **Key Observation:** Average percentage of $KN - S_{add}$ and S_{add} type points increases with increasing batch size. However the average percentage of $KN - S_{add}$ points is more than that of S_{add} points.

  Analysis/Reason(s): With increase in batch size, the number of old points affected due to any batch insertion increases.

  Given a batch update, the number of points in $D \cap D'$ changing their previous KNN list is greater than those affected points which do not change their KNN list. Therefore the average percentage of $KN - S_{add}$ points is more than that of S_{add} type points.

- **Key Observation:** Percentage of $KN - S_{add}$ or S_{add} type points while processing 10000 batches (addition) remains less than 0.05%.

  Analysis/Reason(s): The number of points altering their KNN list or those forming new links and breaking existing ones due to any batch update are less than 0.05% of the total number of points in the updated base dataset.

3. **Birch3 (Addition):** The parameters were set as: K = 5, δ_{sim} = 2 and δ_{core} = 2. The base dataset size was taken to be 80000. A total of 20000 points were added in multiple batches to extract the clusters dynamically. The batch size varied from 2 to 50 as the CPU time for batch-incremental algorithms reduced with increase in batch size. $BISDB_{add}$ proved to be the most efficient algorithm out of the three addition variants (Figure 4.12).

FIGURE 4.12: Birch3 dataset: Efficiency comparison between batch incremental addition algorithms.

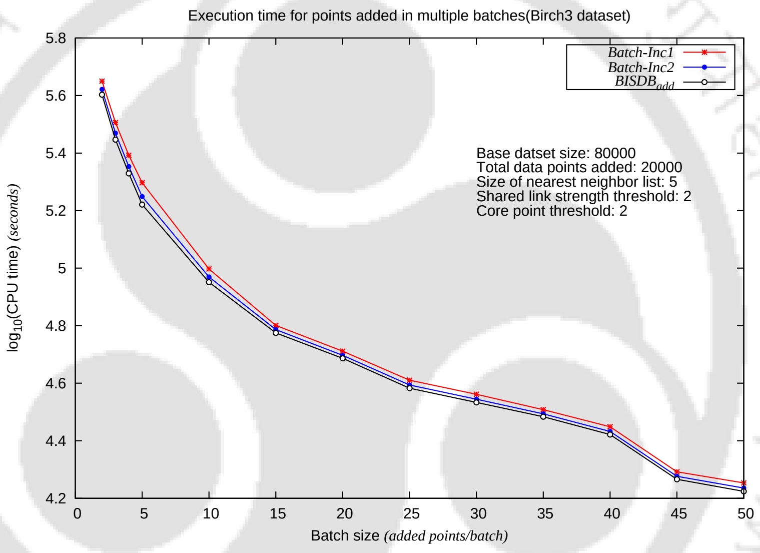


FIGURE 4.13: Birch3 dataset: Average percentage of $KN - S_{add}$ and S_{add} type points for multiple batch insertions of varying batch size ($BISDB_{add}$)

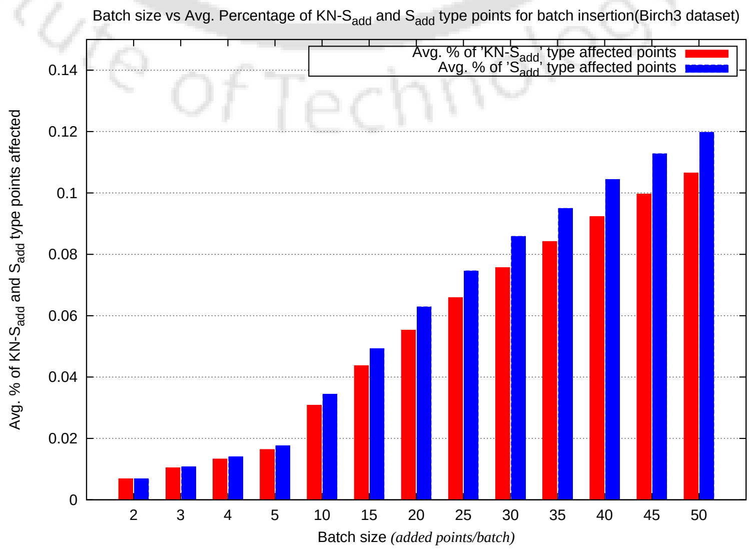
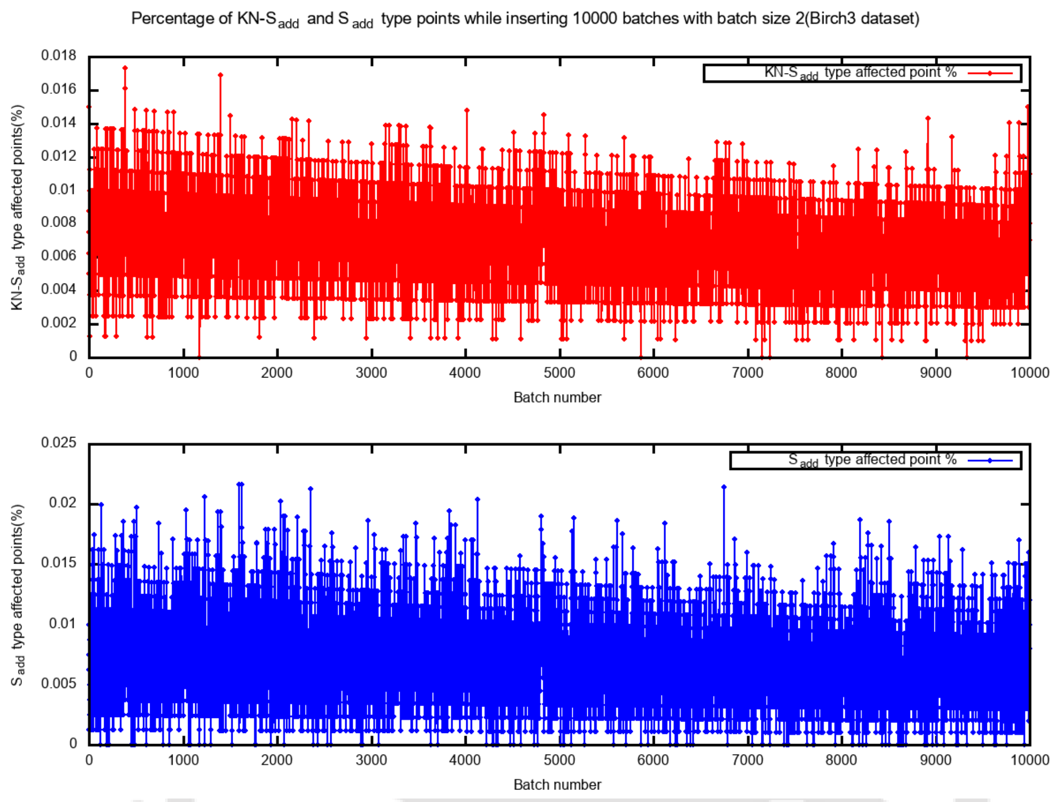


FIGURE 4.14: Birch3 dataset: Percentage of $KN - S_{add}$ and S_{add} type points created while adding 1500 batches ($BISDB_{add}$).



We computed the percentage of affected points while inserting 10000 batches with a batch of size 2. A maximum of about 0.0173% $KN - S_{add}$ type and about 0.0216% of S_{add} type affected points were identified while processing batch number 386 and 1599 respectively (Figure 4.14). The mean percentage of $KN - S_{add}$ and S_{add} type points per batch was found to be around 0.11% and 0.12% respectively for a batch of size 50 involving 400 batches (Figure 4.13).

List of key observations and reasons (Addition) for Birch3 dataset:

- **Key Observation:** Average percentage of $KN - S_{add}$ and S_{add} type points increases with increasing batch size. However the average percentage of $KN - S_{add}$ points is less than that of S_{add} points.
  Analysis/Reason(s): With greater batch size, the number of affected points in $D \cap D'$ due to any batch insertion increases.

  Given any batch update, the number of old points changing their previous KNN list is less than those affected points which do not change their KNN list. Therefore the average percentage of $KN - S_{add}$ type points is less than that of S_{add} type points.

- **Key Observation:** Percentage of $KN - S_{add}$ or S_{add} type affected points while processing 10000 batches (addition) remains less than 0.05%.
  Analysis/Reason(s): The number of points altering their KNN list or forming new links and breaking existing ones are less than 0.05% of the size of updated base dataset.

$BISD_{add}$ therefore proves to be the most efficient batch incremental algorithm (addition) out of the three proposed variants (Figure 4.15). The selective handling of affected data points in $D \cap D'$ while reconstructing the algorithmic components incrementally leads to a grater efficiency of $BISDB_{add}$. The clusters are detected dynamically with minimal interference on the base dataset.

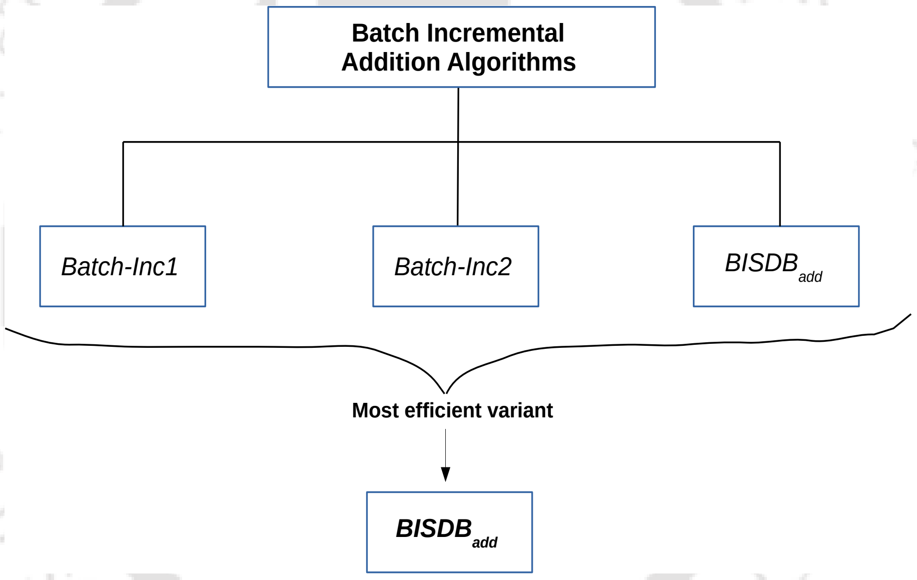


FIGURE 4.15: Summary of Phase-1 experiments for addition.

4.9.2 Phase-2: Prove efficiency of the most effective batch incremental variant (addition) over InSDB [1]

In this phase, we aim to establish the fact that $BISDB_{add}$ is more efficient than InSDB [1] for constant and variable updates made to the base dataset.

Constant updates: In case of constant updates, a fixed number of points were inserted to the base dataset in multiple batches of identical batch size. We compared the efficiency of $BISDB_{add}$ with InSDB [1] which executes the same number of insertions in a point wise manner (Refer Table 4.6).

TABLE 4.6: Performance comparison of $BISDB_{add}$ and InSDB [1] for constant updates.

Dataset	Size	Base dataset size	Added	InSDB (sec)	$BISDB_{add}$ 50p/b (sec)	Speedup ratio
Mopsi2012	13000	10000	3000	9289.24	188.50	49.27
5D points set	100000	80000	20000	44773.57	1551.82	28.85
Birch3	100000	80000	20000	681310.62	16751.17	40.67

Key observation(s): $BISDB_{add}$ inserted 50 points per batch for three datasets: Mopsi12, 5D and Birch3 to make up for the total number of points to be added. For each of the three datasets, the batch incremental method proved to be more efficient than the point based InSDB algorithm.

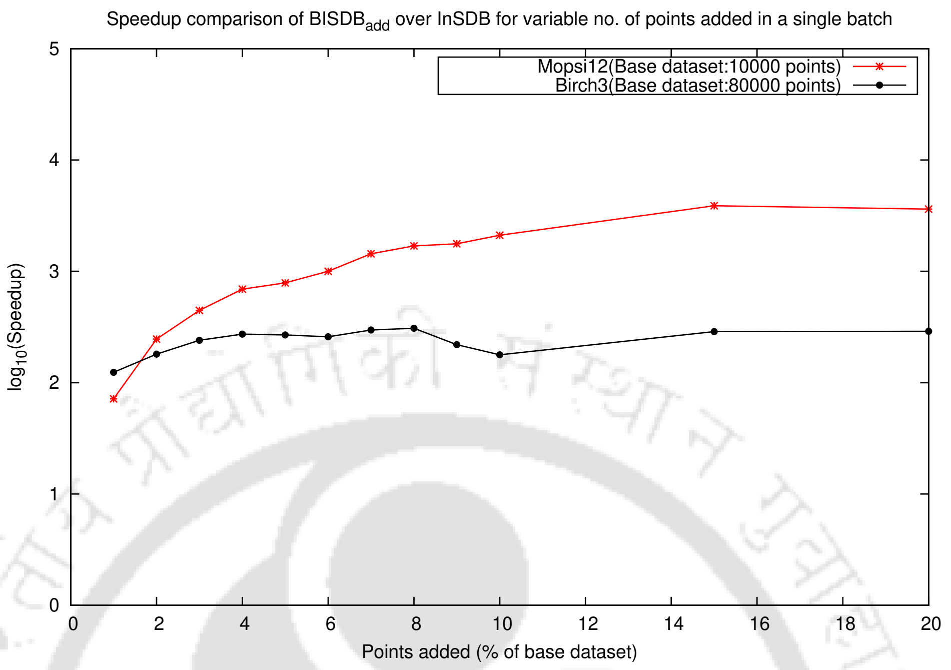
Analysis/Reason(s): In InSDB, the data points are added one at a time. Although the clusters are detected incrementally, the construction of KNN lists and the K-SNN graph takes place after every point insertion. The detection of core, non-core points and the clusters also become repetitive of the number of points added. This makes the overall process slow as compared to $BISDB_{add}$, where the reconstruction of algorithmic components is only repetitive of the number of batches inserted.

Each time a new point is added, the affected points are determined by scanning the base dataset. The size of base dataset increases after every insertion. As a result, the task of finding affected points from the base dataset becomes computationally more intensive with every new insertion. If $T_{BISDB_{add}}$ and T_{InSDB} represent the final CPU execution times after all the points have been added, then experimentally we showed that $T_{BISDB_{add}} < T_{InSDB}$ (Refer Table 4.6):

Variable updates: A variable number of insertions were made to the base dataset in a single batch. We executed $BISB_{add}$ by making insertions ranging from 1% to 20% of the base dataset in one batch. Identical number of insertions were carried out in a point wise manner while executing the InSDB algorithm. Each time a batch update was inflicted, the efficiency of $BISB_{add}$ was compared with InSDB [1]. We observed a general tendency of slow increase (or near constancy) in speedup with increasing percentage of updates (addition) being made to the base dataset. Figure 4.16 demonstrates the speedup comparison of $BISB_{add}$ with InSDB for variable updates.

$BISD_{add}$ therefore outperforms InSDB when updates are made to a larger base dataset. Efficiency is also observed in case of updates being inflicted on the base dataset in variable proportions, $\therefore T_{batch-ins} < T_{point-ins}$.

Figure 4.16: Speedup comparison of $BISDB_{add}$ with InSDB for variable number of points added in a single batch

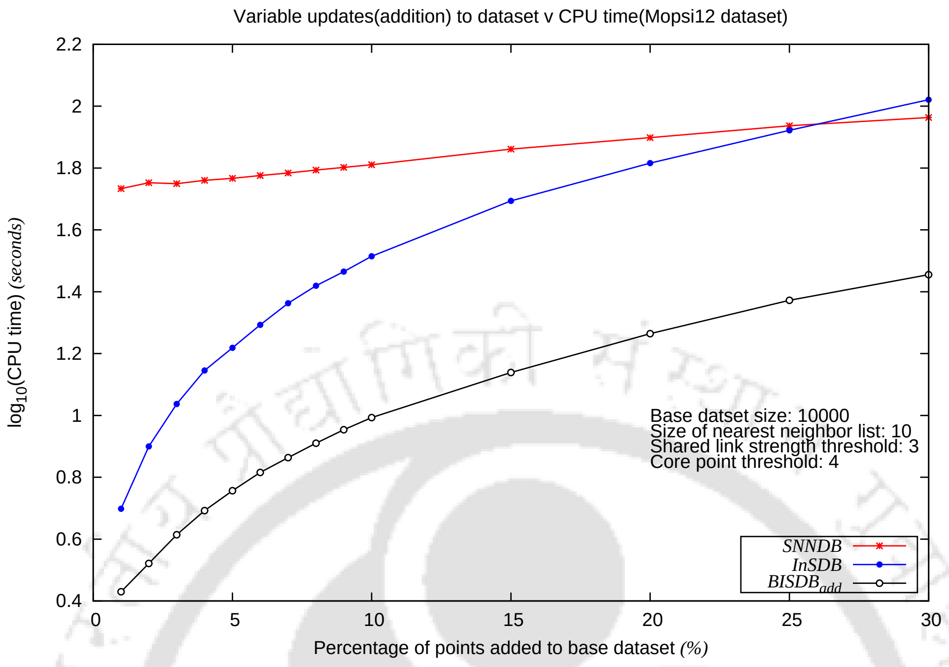


4.9.3 Phase-3: $BISDB_{add}$ and SNNDB are more effective than InSDB when large changes are made to the base dataset

Through experimental observations, we establish that when larger changes are made to the base dataset, the naive method (SNNDB) and the batch incremental algorithm ($BISDB_{add}$) outperform the point-based incremental technique (InSDB). For illustration, we used the Mopsi12 dataset to demonstrate this property (Figure 4.17).

We implemented the $BISDB_{add}$ and InSDB [1] algorithm by taking a base dataset of size 10000. While executing $BISDB_{add}$, we inserted data points in a single batch, with batch size varying from 1% to 30% of the base dataset. Identical number of insertions were made in a point wise manner while executing the InSDB algorithm. Corresponding to every batch insertion, the CPU execution time of $BISDB_{add}$ was compared with the point based method InSDB. We also executed the SNNDB [24] clustering algorithm and compared its efficiency with $BISDB_{add}$ and InSDB. The total input size for SNNDB is a combination of base dataset and the added points in a batch.

FIGURE 4.17: Execution time of SNNDB, InSDB and $BISDB_{add}$ for variable updates (addition) in a single batch.



From Figure 4.17, we identify that both $BISDB_{add}$ and InSDB [1] maintained a better efficiency than SNNDB [24] till a certain stage. However when the extent of insertion exceeded 25% of the base dataset, the non-incremental SNNDB started achieving a better efficiency than the point-based InSDB method. The $BISDB_{add}$ algorithm consistently outperformed both the InSDB and SNNDB algorithm for additions of all inducted batch sizes. Point-based incremental algorithm therefore fails to achieve a better efficiency when larger updates are inflicted upon a base dataset.

4.9.4 Phase-4: Prove the efficiency of $BISDB_{add}$ over SNNDB

We executed the $BISD_{add}$ algorithm and compared its efficiency with SNNDB [24]. In this phase, both constant and variable updates were made to the base dataset. For constant updates, a fixed number of points were inserted to the base dataset in multiple batches. The variable updates were made in a single batch with the batch size varying from 1% to 20% of the base dataset. We then measured the speedup of $BISDB_{add}$ w.r.t.,. the SNNDB algorithm.

Key observation(s): For both constant and variable updates (Figures 4.18, 4.19), the speedup of $BISDB_{add}$ over SNNDB gradually decreases with increasing size of updates being made to the base dataset.

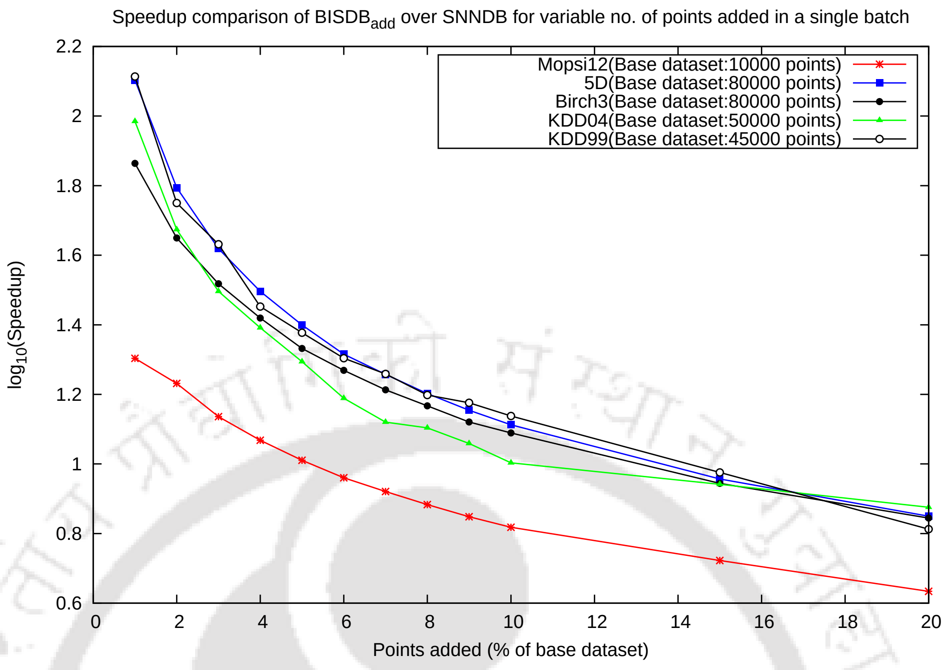


FIGURE 4.18: Speedup of $BISDB_{add}$ over SNNDB for variable addition.

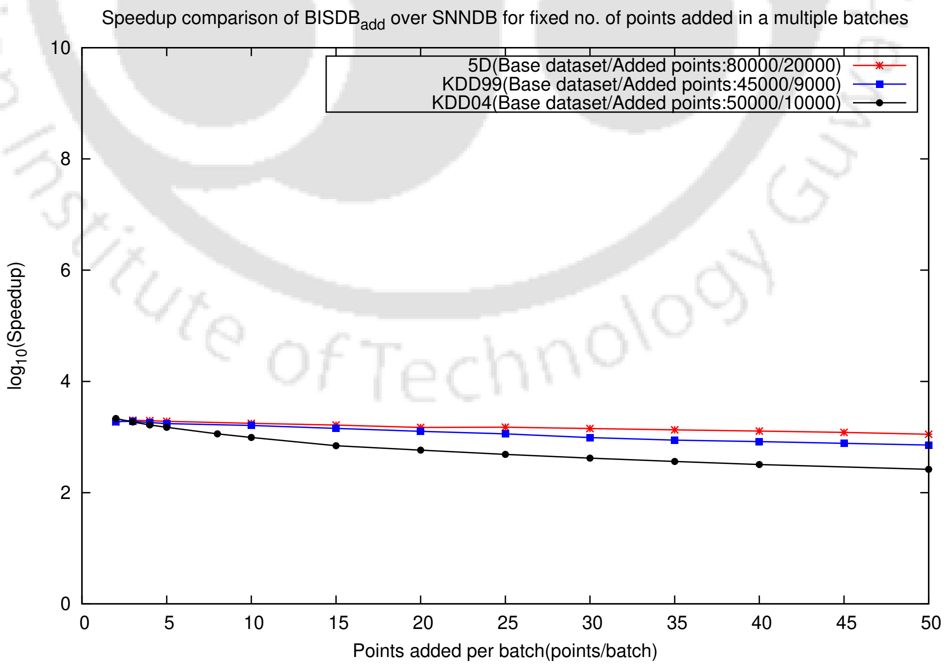


FIGURE 4.19: Speedup of $BISDB_{add}$ over SNNDB for constant addition.

Analysis/Reason(s): While processing smaller updates, $BISDB_{add}$ deals with insignificant percentage of $KN - S_{add}$ and S_{add} type affected points. As the rest of the points retain their AlgoComp values, post new insertions the time required to reconstruct the KNN lists, K-SNN graph and extract clusters incrementally is very less.

As observed, $BISDB_{add}$ achieved the desired efficiency in terms of CPU execution time over SNNDB, $\therefore T_{batch-ins} < T_{naive}$. However, the efficiency was obtained at the cost of certain memory overhead. $BISDB_{add}$ consumed a maximum of about 60% more memory than SNNDB in case of 5D dataset. The mean memory overhead due to $BISDB_{add}$ over SNNDB was around 38.87%, $\therefore |Mem_{batch-ins} - Mem_{naive}| \approx 0.39$.

4.10 Cluster analysis

In this section we present the details of clusters that were obtained after executing the naive method (SNNDB [24]), the point-based method (InSDB [1]) and the proposed batch incremental clustering algorithm ($BISDB_{add}$). We compared the number of clusters, core and non-core points along with the outliers that were obtained from executing the respective algorithms (Refer Tables 4.7 and 4.8). Based on the tabular results, it is evident that the set of clusters obtained from running the naive method and the incremental methods for addition are identical.

TABLE 4.7: Cluster details of SNNDB for all the datasets to compare with the incremental addition algorithms.

Dataset	Algorithm	Input size	#Clusters	#Core points	#Non-core points	#Outliers
Mopsi12	SNNDB	13000	408	10533	1372	1095
5D		100000	2451	6332	4316	89352
Birch3		100000	11293	68653	19585	11762
KDD'99		54000	979	7086	3172	43742
KDD'04		60000	729	1486	3810	54704

TABLE 4.8: Cluster details of InSDB and $BISDB_{add}$ for all the datasets.

Dataset	Algorithm	Base dataset/ Added points	#Clusters	#Core points	#Non-core points	#Outliers
Mopsi12	$BISDB_{add}$ / InSDB	10000 / 3000	408	10533	1372	1095
5D		80000 / 20000	2451	6332	4316	89352
Birch3		80000 / 20000	11293	68653	19585	11762
KDD'99		45000 / 9000	979	7086	3172	43742
KDD'04		50000 / 10000	729	1486	3810	54704

Next we analyze the correctness of clusters obtained from the $BISDB_{add}$ algorithm w.r.t., SNNDB [24] and InSDB [1] based on the following Lemmas.

Lemma 4.1. *Post k number of updates (insertion), the base dataset D ($|D| = n$) changes to D'. KNN(D') obtained from $BISDB_{add}$, SNNDB [24] and InSDB [1] are identical.*

Proof: After k^{th} update, only the $KN - S_{add}$ type affected points change their KNN lists and find the appropriate top K closest points ($BISDB_{add}$). Similarly, if all the $n+k$ points are taken into consideration, only the top K closest points would occupy the appropriate positions of KNN(x) $\forall\, x \in D'$ (SNNDB). InSDB adopts a similar incremental policy of KNN list computation post any point insertion.

Lemma 4.2. $\forall\,(x, y) \in D'$ *(updated dataset),* $sim(x, y)$ *obtained from* $BISDB_{add}$*, SNNDB [24], InSDB [1] are identical.*

Proof: $\because$ KNN(D') post k number of updates are identical for each class of aforementioned algorithms (Lemma 4.1), therefore the number of shared data points ($sim(x, y)$) between the KNN lists $\forall\,(x, y) \in D'$ for $BISDB_{add}$, SNNDB and InSDB are also identical.

Lemma 4.3. $\forall\, x \in D'$, $Core(x), Non{-}core(x)$ *obtained from* $BISDB_{add}$*, SNNDB [24], InSDB [1] are identical.*

Proof:$\because$ $\forall\,(x, y) \in D'$, $sim(x, y)$ is same for each class of aforementioned algorithms (Lemma 4.2). As a result, the number of strong links adjacent to a point is also identical for these algorithms, $\therefore$ $\forall\, x \in D'$, x will retain its same core or non-core status for $BISDB_{add}$, SNNDB and InSDB.

On the basis of Lemmas 4.1, 4.2, 4.3, it can be concluded that the set of clusters obtained from $BISDB_{add}$, SNNDB [24] and InSDB [1] are identical. Therefore we may have the following observation(s):

- $C_{batch-ins}(C_{BISDB_{add}}) = C_{naive}(C_{SNNDB})$
- $C_{batch-ins}(C_{BISDB_{add}}) = C_{point-ins}(C_{InSDB})$

where C_A represents the set of clusters for a given algorithm A [71].

4.10.1 Clustering results in brief

The Mopsi12 dataset with 13000 points consisted of 408 clusters. The largest cluster contained 1373 points while the average cluster size was 29.171. Around 81% of data were core points, as a result 91.4% of data points obtained a cluster membership while rest were treated as noise points. For 5D synthetic dataset, nearly 89.3% of points were outliers. This indicates the sparse distribution of the 5D dataset resulting in only a small fraction of clusters being generated. In Birch3 dataset, about 88% of points obtained a cluster membership. The maximum cluster size was 35 while on an average a cluster contained about 7.81 data points.

In case of KDD'99 and KDD'04 dataset, a large share of data points were outliers. For KDD'99, the maximum cluster size was 104 while for KDD'04 it was eighteen. The clusters obtained from the algorithms are parameter sensitive. The parameter values were specified while conducting experimental evaluation for individual datasets (Refer Section 4.9.1). The results provided are based on parameter values set prior to any execution. Any change in these values may alter the clustering output along with the set of core, non-core and noise points.

4.11 Conclusion

In this chapter, we proposed an incremental alternative to the SNNDB [24] clustering algorithm. InSDB [1], an existing incremental extension of SNNDB relies on point-based insertion. The process becomes extremely slow when updates are made on a larger base dataset. Also when the size of updates increases, InSDB fails to detect clusters efficiently as compared to SNNDB. This is a major flaw on the part of InSDB algorithm which acts as a motivation behind designing of batch incremental clustering algorithms handling addition of data points.

The SNNDB algorithm computes the KNN list, similarity matrix, core and non-core points while detecting clusters of arbitrary shapes, sizes and densities. In order to produce an incremental extension to SNNDB in batch mode, each of these components need to be computed incrementally. In our first line of proposed batch incremental clustering algorithms for addition, $Batch-Inc1$ finds the KNN list of the data points incrementally. Next, we proposed $Batch-Inc2$ finding both the KNN lists and K-SNN graph (similarity matrix) incrementally. However,

$BISDB_{add}$ proved to be the most efficient of all, as it computes the KNN lists, similarity matrix, core and non-core points incrementally.

The incremental algorithms are designed to work for smaller updates made to the base dataset. In this chapter, we showed that for small changes the batch incremental approach maintains a higher efficiency. The efficiency gradually diminishes with increase in the size of data updates. We also observed that for variable updates made to the base dataset, $BISDB_{add}$ maintained a better efficiency over SNNDB and InSDB, $\therefore T_{batch-ins} < T_{naive}$ and $T_{batch-ins} < T_{point-ins}$. However, the efficiency of $BISDB_{add}$ over naive SNNDB algorithm was achieved at a cost of about 38.87% mean memory usage.

The set of clusters obtained by SNNDB, InSDB and $BISDB_{add}$ are identical thereby proving the objectives as stated in Section 4.4. For each of the five datasets, we computed the clusters, core and non-core points along with the outliers. While Mopsi12 dataset involved a significant percentage of data points in formation of clusters, KDD'99 and KDD'04 mostly consisted of outliers.

Chapter 5

$BISDB_{del}$: Towards Exact Incremental Clustering in Batch-Mode for Deletion using SNN-DBSCAN

In the previous chapter, we targeted the individual components of SNN-DBSCAN [24] (SNNDB) to come up with batch incremental algorithms for addition viz. $Batch-Inc1$, $Batch-Inc2$ and $BISDB_{add}$. In this chapter, we propose incremental alternatives for facilitating deletion of data points [71] in batch mode. For this purpose, initially two sub-variant algorithms viz. $Batch-Dec1$ and $Batch-Dec2$ were proposed. While $Batch-Dec1$ targets only a single component of SNNDB incrementally, $Batch-Dec2$ deals with two components. Both these algorithms lead towards designing of the most effective batch incremental variant in form of $BISDB_{del}$ (Batch Incremental Shared Nearest Neighbor Density Based Clustering Algorithm for deletion). The $BISDB_{del}$ algorithm targets all the components of SNNDB incrementally.

5.1 Motivation

Existing incremental extension to SNNDB [24] viz. IncSNN-DBSCAN (InSDB) [1] facilitates addition of data points one at a time. However, InSDB fails to detect clusters dynamically when points are deleted from the base dataset[1]. In order to address this issue, we incrementally extend SNNDB supporting deletion of data points in batches. Table 5.1 provides a brief description about the motivation behind our work.

TABLE 5.1: Motivation behind developing the $BISDB_{del}$ clustering algorithm.

Motivation	**Description**
Small frequent Updates	When minimal number of deletions are made from a larger base dataset, the changes in clustering are also expected to be small. Designing of incremental algorithms will ensure an efficient handling of redundant computation while processing these minor changes.
InSDB [1] does not handle deletion	InSDB only facilitates insertion of data points. While deletion is also an integral part of dynamic datasets, the clustering results may be affected when points are removed from the base dataset. Old links may break and clusters may split. Therefore it becomes important to develop an efficient incremental algorithm capable of handling deletion in batch mode.

5.1.1 Chapter contributions

The key contribution(s) made in this chapter may be summarized as follows:

1. We propose three incremental variants of SNNDB, each of which processes updates made due to deletion of data in batch mode. These three algorithms are $Batch-Dec1$, $Batch-Dec2$ and $BISDB_{del}$ (Table 5.2). Experimentally, we observed that the third variant $BISDB_{del}$ is the most efficient as compared to the other two variants.

2. We showed the effectiveness of our fastest incremental variant $BISDB_{del}$ over SNNDB [24] while handling minimal changes made to the dataset.

3. We demonstrated the fact that when size of base dataset decreases, point wise deletion ($BISDB_{del}$ with a batch size of one) no longer remains an effective option to detect clusters dynamically.

4. A thorough cluster analysis is also provided.

[1]Base dataset is one from which points are deleted.

Table 5.2: Brief overview of our proposed batch incremental clustering algorithms for deletion (Refer Section 5.3 for definitions of related concepts).

Algorithm	**Brief working mechanism**	**Advantage**	**Improvement**
Batch-Dec1	Computes the KNN lists incrementally, detects same clusters as SNNDB, supports batch wise deletion.	Reduces the time taken to compute the KNN lists post new point removals.	*Batch-Dec2*
Batch-Dec2	Computes the KNN lists and similarity matrix incrementally, detects same clusters as SNNDB, supports batch wise deletion.	Reduces the time taken to compute the KNN lists and construct K-SNN graph post new removals.	$BISDB_{del}$
$BISDB_{del}$	Computes the KNN list, similarity matrix and the set of core and non-core points incrementally, same clusters as SNNDB are detected, supports batch wise deletion.	Reduces the time taken to compute the KNN lists, construct K-SNN graph, identify core and non-core points post new removals.	

5.2 Related work and background

SNNDB [24] is a combination of shared nearest neighbors [52] (SNN) clustering scheme and DBSCAN [23]. The concept of SNN is used to determine the proximity score between any pair of data points. Two data objects are said to share a strong link if the number of overlapping points between their KNN lists exceeds a certain threshold provided both the points are present in each others' KNN list. There also exists a prior incremental extension to SNNDB in form of InSDB [1]. InSDB limits itself to only addition of data points without focusing on deletion (See Chapter 4, Section 4.2).

5.3 Preliminaries and Definitions

In this section, we define the key terms and concepts used in this contribution (Refer Table 5.3 for the meaning of notations used henceforth).

Table 5.3: Major notations used in this chapter (third contribution).

Notation	Description
C	Set of Clusters prior to any changes in dataset
C'	Set of Clusters after dataset is updated
D	Original (Base) dataset
D'	Changed dataset after deletions
B	Number of batches
k	No. of points per batch (deleted)
k'	Total no. of points to be deleted
K	Size of the K-Nearest Neighbour list
δ_{sim}	Strong link formation threshold
δ_{core}	Core point formation threshold
$\mathcal{P}(.)$	Power set
KNN (.)	KNN list of any data point.
e-KNN (.)	Extended KNN list of any data point.
Sim_Mat (.)	Similarity matrix of dataset.
Core (.)	Set of core points of dataset.
Non-Core (.)	Set of non-core points of dataset.
$\lvert.\rvert$	Size of a set.

5.3.1 K-nearest neighbor (KNN) list

We define the KNN list of a data point by identifying its top-K (K $\in \mathbf{Z}^+$) closest [2] points.

5.3.2 Shared nearest neighbors (SNN)

The SNN [52] or similarity between two data points p, q is defined as number of points p and q have in common between their respective KNN lists. The SNN value is given by the following equation:

$$similarity(p, q) = KNN(p) \cap KNN(q) \tag{5.1}$$

where *KNN* (x) represents the number of elements present in the KNN list of data point x.

[2]For our purpose, we adopt the Euclidean distance measure.

5.3.3 Similarity matrix or SNN graph

Similarity matrix represents the shared nearest neighbor (SNN) graph. The data points are treated as nodes while the similarity value between any pair of points is the considered as the edge weight.

5.3.4 K-SNN graph

K-Sparsified SNN (K-SNN) graph is the residual graph formed after "K-Nearest Neighbor sparsification" of the original SNN graph [24, 52]. Here K represents the size of KNN list for each data point in the dataset. The surviving links (edge weight $\geq \delta_{sim}$) in the K-SNN graph are termed as the shared strong links or strong links (See details in Chapter 4, Section 4.3).

5.3.5 Core, non-core and noise points

In the K-SNN graph, if the number of strong links associated with a particular point is greater than a certain threshold value δ_{core} (say), then the point obtains a core status. The remaining points are classified as non-core.

The non-core points which do not share a link with any of the core points and fail obtain a cluster membership are classified as noise points.

5.3.6 Clustering

Given a dataset D, a similarity function *sim(x,y)*, and a point density function *dense(x)*, we define clustering by a mapping $f : D \rightarrow C$, where $C = \mathcal{P}(D)$. If x,y $\in D$, $x \neq y$ and there exists two threshold values δ_{sim}, δ_{core}, then may have the following interpretations:

1. *If sim(x,y)* $\geq \delta_{sim}$, *dense(x)* $> \delta_{core}$ *and dense(y)* $> \delta_{core}$, *then f(x) = f(y).*

2. *If sim(x,y)* $\geq \delta_{sim}$, *dense(x)* $> \delta_{core}$ *and dense(y)* $\not> \delta_{core}$, *such that* $\exists z \in D$ *where* $x \neq y \neq z$, *dense(z)* $> \delta_{core}$ *and sim(y,z)* $\geq \delta_{sim}$. *Then if sim(y,z) > sim(x,y), then f(y) = f(z), otherwise if sim(x,y) > sim(y,z), then f(y) = f(x).*

3. *$\forall x \in D$, if dense(x) $\ngtr \delta_{core}$ and $\nexists y \in D$ such that sim(x,y) $\geq \delta_{sim}$ and dense(y) $> \delta_{core}$, then $\{x\} \notin C$.*

As per the first point, if the degree of closeness or similarity between points x and y is greater than or equal to a threshold value δ_{sim} and x, y are dense or core points, then both x and y are a part of the same cluster.

As per the second point, the similarity between points x and y is greater than or equal to a threshold value δ_{sim} and x is core but y is non-core. There exists another core point z and the similarity between y and z is greater than or equal to δ_{sim}. In that case, if y is more similar to z than x, then points y and z belong to the same cluster. However, if y is more similar to point x than z, then y and x belong to the same cluster.

The third point states that if x is a non-core point and there exists no core point y with which x has a similarity value greater than or equal to δ_{sim}, then x is categorized as a noise point.

5.3.7 Batch Incremental Clustering (Deletion)

Given a data set D alongwith its initial clustering $f : D \rightarrow C$ where $C \subseteq \mathcal{P}\ (D)$, a deletion sequence of B batches with 'k' points per batch takes place. After k' $\leq kB$ number of deletions where $kB \pmod{k'} \equiv 0$, let D' be the updated data set. Then an incremental clustering given by a mapping $h : D' \rightarrow C'$, with $C' \subseteq \mathcal{P}\ (D')$ is isomorphic to the one time clustering $f(D')$ produced by the non-incremental algorithm.

5.4 Problem formulation

For B number of batch removals ($B \in \mathbf{N}$) with k points/batch, let T_{naive} be the total time taken by the non-incremental method, $T_{point-del}$ be the total time taken by the point based deletion incremental method with $k' \leq kB$ removals, and $T_{batch-del}$ be the total time taken by the batch incremental method. Let C_{naive}, $C_{point-del}$ and $C_{batch-del}$ be the respective set of clusters obtained after k' updates. If $|Mem_{batch-del} - Mem_{naive}|$ be the difference in percentage of average memory consumed, then we aim to establish the following objectives:

1. (a) $T_{batch-del} < T_{naive}$
   (b) $C_{batch-del} = C_{naive}$
2. (a) $T_{batch-del} < T_{point-del}$
   (b) $C_{batch-del} = C_{point-del}$
3. $|Mem_{batch-del} - Mem_{naive}| \leq \delta$, where δ is a small real number.

5.5 Structure of the proposed batch incremental SNNDB clustering algorithms for deletion

The generic structure of our proposed batch incremental clustering algorithms for deletion is as follows:

1. We characterize each data point by its following properties:
   (a) KNN list.
   (b) SNN value or similarity value with each of the adjacent data points connected by a strong link.
   (c) Core or non-core status.
2. New data points are deleted from the base dataset in batches.
3. The values of properties for the batch of deleted points are erased.
4. The affected points belonging to the base dataset may undergo changes in values of at least one of their properties upon removal of existing data points.
5. The surviving data points which remain unaffected due to deletion of points do not change the values of their properties.
6. Some of the existing points may change their status from core to non-core while others may change from non-core to core. The strength of shared links between data points may alter. If the link strength reduces below δ_{sim}, the link gets broken resulting in possible splitting of clusters.
7. The overall changed dataset consists of data points with updated property values.

8. Two connected core points are contained in the same cluster. The cluster expansion takes place by grouping the core points accordingly. The non-core points are put into a cluster of their nearest core point. The nearest core point is the one that shares a strong link with the concerned non-core point, and has a higher edge weight as compared to the other adjacent core points (See Chapter 4). Points which fail to obtain any cluster membership are classified as noise points.

9. The reduced dataset becomes the new base dataset. The updated property values for each of the existing data points are retained. These values are utilized for processing the next batch of deleted points. Repeat Steps 2 to 9 until the requisite number of deletions have been made.

5.6 Batch-Incremental SNNDB Clustering Algorithms for Deletion

In this section we present our proposed batch-incremental SNNDB clustering algorithms for deletion viz. $Batch-Dec1$, $Batch-Dec2$ and $BISDB_{del}$. The goal of these algorithms is to find clusters dynamically while points are deleted in batches from a base dataset D. Prior to executing each of these batch incremental algorithms, the naive SNNDB [24] clustering algorithm is executed upon D. However, while executing SNNDB, we increase the window size of KNN list for each data point. Each of the algorithms for deletion now maintains an additional space for (w - 1)·K extra points in their expanded KNN (e-KNN) list, where $w \in \mathbf{N}$, $w \geq 2$ and K is the size of the original KNN list. The total size of e-KNN list for each data point turns out to be K+ (w-1)·K or w·K. For a given point $p \in D$, each of the points in e-KNN(p) including the additional $(w - 1)$·K points are placed in increasing order of their distance from p.

Contrary to the proposed three variants for addition, no new point enters the base dataset D. Instead, the data points are removed from D. As a result, the original KNN list for some of the existing points may shrink in size. In order to maintain a valid size of the original KNN list, e-KNN list becomes functional. In case of shrinkage, points from the additional (w - 1)·K space migrate to the top K-slots of the e-KNN list to fill the vacant positions.

The following set of information is obtained by executing the SNNDB method w.r.t., base dataset D:

1. e-KNN list $\forall\, p \in D$.

2. Similarity matrix (K-SNN graph) w.r.t., D given as $Sim\_Mat(D)$.

3. Set of core and non-core points w.r.t., D given as $Core(D)$ and $Non-Core(D)$.

4. Set of clusters C w.r.t., D.

5. Set of noise points w.r.t., D.

After the data points are removed from D, the changed dataset D' gets reduced in size due to loss of points ($|D'| < |D|$). Therefore $D \cap D'$ ($D \cap D' = D'$) represents the surviving or existing set of points in the data space.

In the following subsections, we present each of our proposed batch incremental deletion algorithms in a step-wise manner supported by graphical illustrations wherever necessary.

5.6.1 The $Batch-Dec1$ clustering algorithm

The $Batch-Dec1$ algorithm builds the updated e-KNN list for each of the existing data points incrementally. The new similarity matrix (K-SNN$_{updated}$ graph) and the set of new core and non-core points are determined non-incrementally. The steps of the $Batch-Dec1$ algorithm are as follows:

1. **Step 1 - Set the parameters:** The algorithm takes four parameters: K, w, δ_{sim} and δ_{core}. The parameters have the following meanings:
   (a) K denotes the size of original KNN list for each data point.
   (b) w is the multiplier that decides the number of times the original KNN list be expanded in order to build the e-KNN list.
   (c) Given that two data points p and q are present in each others' KNN list, δ_{sim} is the minimum value of SNN required for p, q to form a strong link between them.
   (d) δ_{core} is the minimum number of strong links adjacent to a point p exceeding which p becomes a core point.

2. **Step 2 - Obtain the required data from prior SNNDB execution:**

(a) Get the base dataset D where $|D| = n$ (say).

(b) Get the e-KNN list $\forall\, p_i \in D, i = 1, 2, ..., n$.

(c) Get the similarity matrix $Sim\_Mat(D)$.

3. **Step 3 - Delete a batch of existing points:** Delete a batch containing k data points from D. D changes to D' where $|D'| = n - k$.

4. **Step 4 - Remove the components associated with the deleted points:**

   (a) Remove the e-KNN list of the deleted points.

   (b) Remove all the associated shared strong links with the deleted points from the K-SNN graph.

   (c) The deleted points lose their cluster membership (if any).

5. **Step 5 - Compute the updated e-KNN list for existing data points in $D \cap D'$ incrementally:** The number of existing points in D (base dataset) prior to any deletion is n. When k data points are removed from D, D changes to D' ($|D'| = n - k$). Among the remaining points in D', the algorithm identifies those points that have removed an already deleted point from their e-KNN list. While investigating, $Batch - Dec1$ only checks the top-K slots of the concerned point excluding the additional (w-1)·K space. If the concerned point loses any data point from its top-K window due to prior deletion, it is categorized as a $KN - S_{del}$ type affected point. Consequently, the first additional point from the (w-1)·K space migrates to fill the emptied slot in order to maintain the exact size (length K) of original KNN list.

   The movement of a single point from the extra (w-1)·K space in the e-KNN list prevents the shrinkage of top-K space in the original KNN list. In a worst case scenario, for any given point $p \in D'$ (changed dataset), a maximum of K points might be removed from the top-K slots of e-KNN(p) due to prior deletion. In that case, the next set of K points placed in succession in the additional (w-1)·K space ($w \geq 2$), fills the emptied top-K slots. However, the migration creates vacant slot(s) within the additional (w-1)·K space. These vacant slots in e-KNN$_{updated}$ (p) are filled by other surviving points in D' which are closer to p [3]. $Batch - Dec1$ therefore focuses only on re-building the e-KNN$_{updated}$(.) list for $KN - S_{del}$ type points. The e-KNN$_{updated}$(.) lists for remaining unaffected points in $D \cap D'$ are not computed separately.

[3]When no more points remain in additional (w-1)·K space to prevent the shrinkage of top-K window, $Batch - Dec1$ involves entire D' to rebuild e-KNN$_{updated}$(p).

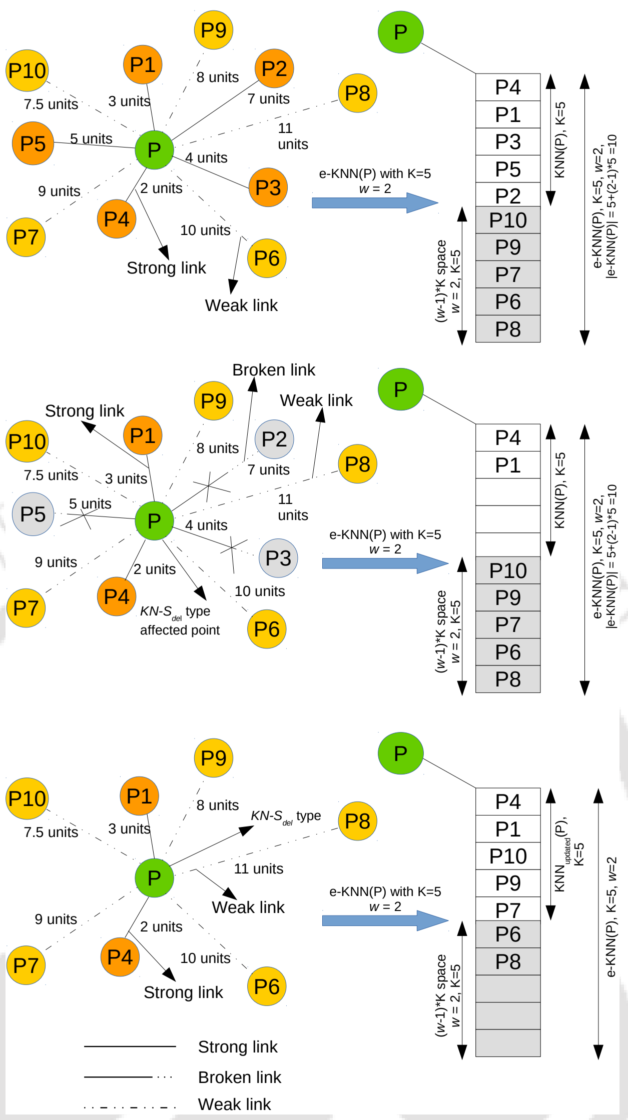


FIGURE 5.1: The formation of $KN - S_{del}$ type affected points upon deletion of existing points

Running example: Through Figure 5.1, we illustrate this step (Step 5) from *Batch* – *Dec*1. We present this example by assuming K = 5 and w = 2. Consider the point P (top most image from Figure 5.1), for which we have the following prior to any deletion:

(a) KNN(P) = {P4, P1, P3, P5, P2}

(b) e-KNN(P) = {P4, P1, P3, P5, P2, P10, P9, P7, P6, P8}

With parameters w and K assuming values 2 and 5 respectively, the total size of e-KNN(P) is 10 (since |e-KNN(.)| = K+ (w-1)·K). The first five points of e-KNN(P):{P4,P1,P3,P5,P2} are a part of KNN(P), while the remaining five points are stored in order to prevent shrinkage of KNN(P) in case of deletion.

Let P2,P5 and P3 be the deleted points (middle image from Figure 5.1). As a result, these points are removed from KNN(P) and consequently from e-KNN(P). KNN(P) now shrinks in size and is reduced to points P4, P1. The strong links that P shared with P2, P5 and P3 are broken. P is therefore categorized as a $KN - S_{del}$ type point.

As per $Batch-Dec1$, the size of additional (w-1)·K space is 5 and it consists of points:{P10,P9,P7,P6,P8} placed in increasing order of distance from P. The removal of P2,P5 and P3 creates three vacant slots within KNN(P) (middle image from Figure 5.1). As a result points P10, P9 and P7 from the additional (w-1)·K space migrate towards KNN(P) to overcome the shrinkage [4]. The updated KNN list and the current e-KNN list of P are given as:

(a) KNN$_{updated}$ (P) = {P4, P1, P10, P9, P7}

(b) e-KNN(P) = {P4, P1, P10, P9, P7, P6, P8}

We observe that the additional (w-1)·K space in e-KNN(P) currently consists of points P6 and P8 (bottom image from Figure 5.1) and three empty slots. Prior to next batch of deletion, *Batch-Dec*1 fills these empty slots in e-KNN(P) to produce e-KNN$_{updated}$(P). This is done to prevent the shrinkage of KNN(P) in case of deletions from subsequent iterations.

6. **Step 6 - Construct the updated K-SNN graph:** The algorithm constructs the K-SNN$_{updated}$ graph or the new similarity matrix ($Sim\_Mat(D')$) non-incrementally based on the remaining points in D'. D' now consists of $n-k$ points. Therefore, $\forall\, p_i \in D', i = 1,2,3,\dots,n-k$, $Batch-Dec1$ determines if a shared strong link can be constructed $\forall\, q \in$ KNN$_{updated}(p_i)$ where KNN$_{updated}(p_i) \subset$ e-KNN$_{updated}(p_i)$.

7. **Step 7 - Identify new core and non-core points:** For each point in K-SNN$_{updated}$ graph ($Sim\_Mat(D')$), if the number of adjacent strong links exceeds δ_{core}, the point obtains a core status otherwise a non-core. The new set of core and non-core points are stored in $Core(D')$ and $Non-Core(D')$ respectively.

[4]We assume that P does not share a strong link with P10, P9 and P7 yet they can be present in KNN$_{updated}$ (P).

8. **Step 8 - Form Clusters:** Two connected core points are grouped into the same cluster. A non-core point is pulled towards the cluster of its nearest core point[5].

9. **Step 9 - Discard noise points:** The non-core points which are not connected to any core point are classified as noise points. Such points do not obtain any cluster membership.

10. **Step 10 - Retain the updated values:**

    (a) $D = D'$

    (b) $n = n - k$

    (c) $\forall\, p_i, i = 1, 2, ..., n - k$
    e-KNN(p_i) = e-KNN$_{updated}(p_i)$.

    (d) $Sim\_Mat(D) = Sim\_Mat(D')$

    (e) $Core(D) = Core(D')$

    (f) $Non - Core(D) = Non - Core(D')$

11. **Step 11 - Repeat Steps 3 to 10 for the next batch of deleted points.**

5.6.2 The $Batch - Dec2$ clustering algorithm

The $Batch - Dec2$ algorithm constructs the updated e-KNN list and the new similarity matrix (K-SNN$_{updated}$ graph) incrementally. The new core and non-core points are determined non-incrementally. The steps of $Batch - Dec2$ algorithm are as follows:

1. **Step 1 - Set the parameters:** The algorithm takes four parameters: K, w, δ_{sim} and δ_{core}.

2. **Step 2 - Obtain the required data from prior SNNDB execution:**

    (a) Get the base dataset D where $|D| = n$ (say).

    (b) Get the e-KNN list $\forall\, p_i \in D, i = 1, 2, 3, \ldots, n$.

    (c) Get the similarity matrix $Sim\_Mat(D)$.

[5]The core point with which the shared link strength is highest becomes the "nearest" core point.

3. **Step 3 - Delete a batch of existing points:** Delete a batch containing k data points from D. D changes to D' where $|D'| = n$-k.

4. **Step 4 - Remove the components associated with the deleted points:**

   (a) Remove the e-KNN list of the deleted points.

   (b) Remove all the associated shared strong links with the deleted points from the K-SNN graph.

   (c) The deleted points lose their cluster membership (if any).

5. **Step 5 - Compute the updated e-KNN list for existing data points in $D \cap D'$ incrementally:** Step 5 is similar to that of $Batch - Dec1$.

6. **Step 6 - Construct the updated K-SNN graph incrementally:** $Batch-Dec2$ introduces a new type of affected point known as the S_{del} type affected point. The KNN lists of S_{del} type points remain unaffected due to removal of existing data points. The S_{del} type points may contain at least one $KN-S_{del}$ type point in their unchanged KNN list. Any surviving non-$KN-S_{del}$ point in the updated KNN list of a $KN-S_{del}$ type point is designated as S_{del} type. The non-$KN-S_{del}$ type points which migrate from the additional (w-1)·K space into the top-K window of a $KN-S_{del}$ type point are also categorized as S_{del} type[6].

   For any S_{del} type point $p \in D \cap D'$ (set of existing points), only the value of shared strong link (edge weight) with p's adjacent points may change but KNN (p) remains the same. As a result we have KNN$_{updated}(p)$ = KNN (p). Since KNN(P) $\subset$ e-KNN(P), $\therefore$ e-KNN$_{updated}$ (p) = e-KNN(p). The remaining set of unaffected points in $D \cap D'$ retain their original KNN lists as well as similarity values and are neither $KN - S_{del}$ nor S_{del} type.

   In order to find the new state of shared strong links in the K-SNN$_{updated}$ graph, $Batch - Dec2$ scans the updated KNN lists of only the $KN - S_{del}$ and S_{del} type points. The existing links between the unaffected points in $D \cap D'$ are allowed to retain their prior similarity value. By targeting only the $KN-S_{del}$ and S_{del} type affected points, $Batch-Dec2$ rebuilds the entire K-SNN$_{updated}$ graph incrementally without involving the changed dataset in its totality.

   Running example: Figure 5.2 illustrates the formation of S_{del} type affected points. We present this example by assuming K = 5 and w = 2. Let us

[6]An affected point is either $KN - S_{del}$ type or S_{del} type but not both.

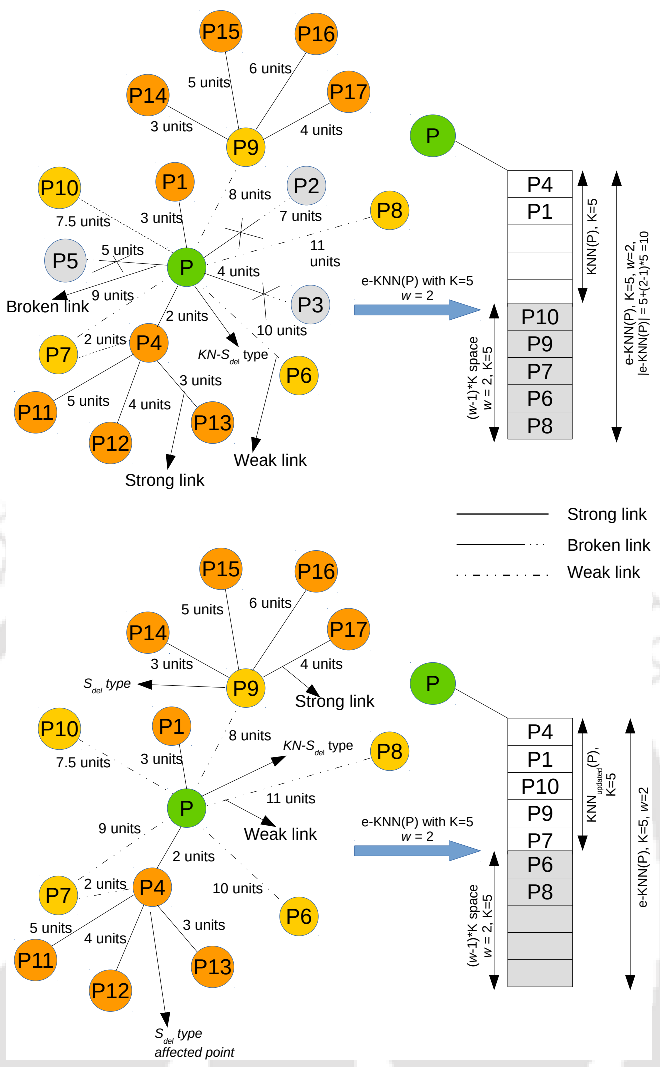


FIGURE 5.2: The formation of S_{del} type affected points upon deletion.

consider the point P (top image of Figure 5.2) having {P4, P1, P3, P5, P2} in its original KNN list (KNN(P)) prior to any deletion. Let P2, P3 and P5 be the deleted points from KNN(P). As a result, three empty slots are created in KNN(P) and the strong link that P shared with each of the deleted points gets broken. P is therefore categorized as a $KN - S_{del}$ type point. The three vacant slots in KNN(P) are filled by points P10, P9 and P7 respectively. These three points were a part of the additional $(w\text{-}1)\cdot$K space in e-KNN(P). The updated KNN list of P (KNN$_{updated}$ (P)) now comprises of {P4,P1,P10,P9,P7} (bottom image of Figure 5.2).

For points P4 and P9 in Figure 5.2, we make the following observations:

(a) KNN$_{updated}$ (P9) = KNN (P9) = {P14, P17, P15, P16,P}

(b) KNN$_{updated}$ (P4) = KNN (P4) = {P, P7, P13, P12, P11}

We observe that P4 and P9 have retained their original KNN list post deletion. P4 was originally present in KNN(P) while P9 has moved into KNN(P) from the additional (w-1)·K space. Since P is a $KN - S_{del}$ type point, therefore P4 and P9 qualify as S_{del} type points.

7. **Step 7 - Identify new core and non-core points:** Step 7 is similar to that of $Batch - Dec1$.

8. **Step 8 - Form Clusters:** Step 8 is similar to that of $Batch - Dec1$.

9. **Step 9 - Discard noise points:** Step 9 is similar to that of $Batch - Dec1$.

10. **Step 10 - Retain the updated values:**

    (a) $D = D'$

    (b) $n = n - k$

    (c) $\forall\, p_i, i = 1, 2, 3, \ldots, n - k$
    e-KNN(p_i) = e-KNN$_{updated}$(p_i).

    (d) $Sim\_Mat(D) = Sim\_Mat(D')$

    (e) $Core(D) = Core(D')$

    (f) $Non - Core(D) = Non - Core(D')$

11. **Step 11 - Repeat Steps 3 to 10 for the next batch of deleted points.**

5.6.3 The $BISDB_{del}$ clustering algorithm

The $BISDB_{del}$ clustering algorithm computes the updated KNN list, the updated K-SNN graph and the new set of core and non-core points incrementally. $BISDB_{del}$ attempts to improve the efficiency of dynamic cluster detection over $Batch - Dec1$ and $Batch - Dec2$. The steps of $BISDB_{del}$ algorithm are as follows:

1. **Step 1 - Set the parameters:** The algorithm takes four parameters: K, w, δ_{sim} and δ_{core}.

2. **Step 2 - Obtain the required data from prior SNNDB execution:**

    (a) Get the base dataset D where $|D| = n$ (say).

    (b) Get the e-KNN list $\forall\, p_i \in D, i = 1, 2, 3, \ldots, n$.

    (c) Get the similarity matrix $Sim\_Mat(D)$.

3. **Step 3 - Delete a batch of existing points:** Delete a batch containing k data points from D. D changes to D' where $|D'| = n$-k.

4. **Step 4 - Remove the components associated with the deleted points:**

   (a) Remove the e-KNN list of the deleted points.

   (b) Remove all the associated shared strong links with the deleted points from the K-SNN graph.

   (c) The deleted points lose their cluster membership (if any).

5. **Step 5 - Compute the updated e-KNN list for existing data points in $D \cap D'$ incrementally:** Step 5 is similar to that of $Batch - Dec1$ and $Batch - Dec2$.

6. **Step 6 - Construct the updated K-SNN graph incrementally:** Step 6 is similar to that of $Batch - Dec2$.

7. **Step 7 - Identify new core and non-core points incrementally:** In the K-SNN$_{updated}$ graph, for each of the total number of $KN - S_{del}$ and S_{del} type points in $D \cap D'$, $BISDB_{del}$ checks whether the number of strong links adjacent to the affected point exceeds δ_{core}. If this happens, the concerned point is treated as a core point or else it is a non-core point. The remaining points retain their existing core or non-core status from the previous iteration[7].

8. **Step 8 - Form Clusters:** Step 8 is similar to that of $Batch - Dec1$ and $Batch - Dec2$.

9. **Step 9 - Discard noise points:** Step 9 is similar to that of $Batch - Dec1$ and $Batch - Dec2$.

10. **Step 10 - Preserve the updated values:**

    (a) $D = D'$

    (b) $n = n - k$

    (c) $\forall\, p_i, i = 1, 2, 3, \ldots, n - k$
    e-KNN(p_i) = e-KNN$_{updated}(p_i)$.

    (d) $Sim\_Mat(D) = Sim\_Mat(D')$

    (e) $Core(D) = Core(D')$

[7]For the first batch of deleted points, the core or non-core status of a point in $D \cap D'$ is derived from initial SNNDB execution upon D.

(f) $Non - Core(D) = Non - Core(D')$

11. **Repeat Steps 3 to 10 for the next batch of deleted points**.

5.6.4 Shared link properties between affected points post deletion

Altering the state of KNN list may lead to a change in similarity values between the affected points. We present all possible scenarios of the state of shared strong links between $KN - S_{del}$ and S_{del} type affected points.

1. $KN - S_{del} \leftrightarrow KN - S_{del}$ **link:** Removal of existing data points from the KNN list of a $KN - S_{del}$ type point creates empty slots within the list. If the deleted points contributed to the shared link strength, then the link strength is bound to decrease. However, if the removed points did not contribute to the link strength, then the similarity value remains same provided the newly migrated points from $(w-1)$*K space do not lie in the common neighborhood of the involved pair of points. On contrary if the migrated points belong to the common neighborhood, then the link strength may increase. Therefore, for a $KN - S_{del} \leftrightarrow KN - S_{del}$ type link, the strength of shared link either decreases, remains same or increases.

2. $KN - S_{del} \leftrightarrow S_{del}$ **link:** The S_{del} type points do not change their KNN list after deletion. Therefore the removed points from the KNN list of a $KN - S_{del}$ type point cannot be a contributory source to the shared strong link. As a result, the link strength continues to remain same post deletion of data points. However, if the newly migrated points from $(w - 1)$*K space lies in the common neighborhood of a $(KN - S_{del}, S_{del})$ pair, then the link strength may go on to increase .

3. $S_{del} \leftrightarrow S_{del}$ **link:** With no change in KNN list for S_{del} type points post deletion, the points which originally contributed to the shared link strength remain unaffected. As a result an identical link strength is observed for a $S_{del} \leftrightarrow S_{del}$ type link.

```
ALGORITHM 3: $BISDB_{del}(D, K, \delta_{sim}, \delta_{core}, w)$
Input: $D$, K , $\delta_{sim}, \delta_{core}, w$;
Output: Clusters;
Set 'nrow' as original no. of data points;
// Update dataset after decrement
for $i \leftarrow 1$ to $n$ do
    Remove existing data point i from the base dataset data_matrix[];
    $i \leftarrow$ i+1;
// Update KNN list of the affected points from existing dataset
for $i \leftarrow 1$ to $nrow$ do
    for each $j \in KNN\_matrix[i]$ do
        if $j$ is a deleted point then
            Remove j from KNN_matrix[i] and shrink KNN (i);
        else
// Identify $KN - S_{del}$ type points
for $i \leftarrow 1$ to $nrow$ do
    if $KNN\_matrix[i].size() < K \wedge i \notin Deleted\_Set$ then
        $i \in KN - S_{del}$ points;
    else
// Fill the empty slots in e-KNN list of $KN - S_{del}$ type points
for each $i \in KN - S_{del}$ do
    for each $j \in KNN\_matrix[i] \wedge j$ is empty do
        Select a point closest to 'i' from (w-1)*K space and fill the empty slot
         in KNN_matrix[i];
    Fill the empty slots of (w-1)*K space with other points in data_matrix[];
    sort (e-KNN (i));
// Identify $S_{del}$ type points
for each $i \in KN - S_{del}$ do
    for each $j \in KNN\_matrix[i]$ do
        if $j \notin KN - S_{del}$ then
            $j \in S_{del}$ points;
        else
// Build K-SNN graph; find core, non-core points incrementally
for each $i \in KN - S_{del} \cup S_{del}$ do
    for each $j \in KNN\_matrix[i]$ do
        if $similarity(i, j) \geq \delta_{sim}$ then
            An edge is formed between points i and j;
        else
    if $similarity\_matrix[i].size() > \delta_{core}$ then
        $i \in CORE$ points set;
    else
        $i \in Non\text{-}CORE$ points set;
```

```
35 Cluster formation is similar to the SNNDB algorithm;
36 Repeat entire process for the next batch of removed points;
```

TABLE 5.4: Summary of the batch-incremental SNNDB clustering algorithms for deletion

Components-Algorithm	$Batch - Dec1$	$Batch - Dec2$	$BISDB_{del}$
Updated KNN list	**Incrementally**	**Incrementally**	**Incrementally**
Updated K-SNN graph	**Non-Incrementally**	**Incrementally**	**Incrementally**
Updated core and non-core points	**Non-Incrementally**	**Non-Incrementally**	**Incrementally**

5.6.5 Summary of the Batch-Incremental SNNDB Clustering Algorithms for deletion

The SNNDB method takes $\mathcal{O}(N^2)$ time towards completion where N is the total number of data points. In our pursuit to improve the efficiency over SNNDB, we initially propose the $Batch - Dec1$ algorithm. $Batch - Dec1$ provides marginal improvement by building the updated e-KNN lists incrementally in $\mathcal{O}(N)$ time. However, building the K-SNN$_{updated}$ graph involves quadratic time complexity. $Batch - Dec2$ aims to address this issue by reconstructing the K-SNN$_{updated}$ graph incrementally upon removal of data points. While building the K-SNN$_{updated}$ graph, $Batch - Dec2$ only updates the shared link strengths of $KN - S_{del}$ and S_{del} type points. For identifying the new core and non-core points, $Batch - Dec2$ involves all the data points in D' (updated dataset). This results in $Batch - Dec2$ having linear time complexity. $BISDB_{del}$ finds the new set of core and non-core points incrementally and therefore improves upon the previous two sub-variant algorithms for deletion. $BISDB_{del}$ also runs in linear time (Refer Algorithm 3 for pseudo-code of $BISDB_{del}$).

5.7 Time complexity analysis of the $BISDB_{del}$ clustering algorithm

Let D be the base dataset where $|D| = n\,(n \in \mathbf{Z}^+)$ be its initial size. Let D' be the updated dataset after deletions. Let us assume that a total of B batches are removed with k points per batch ($k \ll n$). For the B^{th} batch deletion at any point in time, we have $|D'| = n - kB$, where the current size of base dataset is $n - (B-1)k = N(say)$. The size of each deleted batch is significantly smaller than the current size of base dataset, $\therefore k \ll N$.

We provide the time complexity analysis by assuming that the first batch of deletions had been made ($\therefore B = 1$). [Line 7-11]: This segment of $BISDB_{del}$ updates the KNN list of data points dynamically. We clearly observe that for each existing point $i \in D'$, the algorithm checks for the presence of an already deleted point in i's KNN list. For a single point, the algorithm performs at most K comparisons. Deleting a data point j from i's KNN list and replacing by its immediate next point or a point present in the additional $(w-1)$·K space of e-KNN (i) list takes $\mathcal{O}(1)$ time. As a result, the steps required to compute the updated KNN lists of existing points has a worst case time complexity of K·$\mathcal{O}(n-k) \simeq \mathcal{O}(N)$. [Line 12-15]: For determining the number of $KN - S_{del}$ type points, a complete scan of the updated dataset (D') results in K·$\mathcal{O}(n-k) \simeq \mathcal{O}(N)$ time. [Line 26-34]: $BISDB_{del}$ identifies the set of core and non-core points incrementally along with the dynamic reconstruction of the K-SNN$_{updated}$ graph.

5.7.1 Time complexity proof $BISDB_{del}$

Let T_{e-KNN} be the time taken to compute the updated e-KNN list of existing data points in D', then

$$T_{e-KNN} = K \cdot N \tag{5.2}$$

In the new e-KNN list, the top K slots are checked for the trace of any deleted point. A removed point is replaced by its immediate next point positioned in the KNN list. The vacant slot(s) are filled by the points present in additional $(w-1) \cdot K$ space of the e-KNN list. Effectively KN amount of time is required to find the $KN - S_{del}$ type points, $\therefore T_{KN-S_{del}} = KN$.

Let $T_{S_{del}}$ be the time taken to find the S_{del} type points. Since S_{del} type points are determined from the updated e-KNN list of the $KN - S_{del}$ type points, we have the following equation:

$$\begin{aligned} T_{S_{del}} &= |KN - S_{del}| \cdot K \\ &\quad [where\ |KN - S_{del}| \ll N] \end{aligned} \tag{5.3}$$

The term K is used instead of $2K$ (See Equation 5.3) because the removed points from the updated e-KNN list of any $KN - S_{del}$ type point are not taken into consideration. Let T_{K-SNN} = time taken to construct the updated K-SNN graph,

$$\begin{aligned} \therefore T_{K-SNN} &= (|KN - S_{del}| + |S_{del}|) \cdot 2c'\Big(\frac{K}{2} + \log_2 K\Big) \\ &[where\ \ |S_{del}| \ll N] \end{aligned} \tag{5.4}$$

Let $T_{Core+Non-Core}, T_{Cluster}$ be the time taken to find the core, non-core points and the clusters,

$$\therefore T_{Core+Non-Core} = |KN - S_{del}| + |S_{del}| \tag{5.5}$$

$$T_{Cluster} = K \cdot N \tag{5.6}$$

If $T_{BISDB_{del}}$ be the total time taken by the $BISDB_{del}$ algorithm, then (using Equations 5.2, 5.4, 5.5, 5.6) we obtain the following:

$$\begin{aligned} T_{BISDB_{del}} &= T_{e-KNN} + T_{K-SNN} + T_{Core+Non-Core} \\ &\quad + T_{Cluster} \\ &= cKN + c_1 \qquad [c\ is\ constant] \\ &[\because K, KN - S_{del}, S_{del}\ are\ constants] \\ &[\therefore c_1\ is\ constant] \\ \implies T_{BISDB_{del}} &\simeq \mathcal{O}(N) \end{aligned} \tag{5.7}$$

$Here \quad c_1 = (|KN - S_{del}| + |S_{del}|)(1 + 2c'(\frac{K}{2} + \log_2 K))$

5.8 Experimental evaluation

The experiments were carried out on three real world and two synthetic datasets (Table 5.5) to prove the efficiency of $BISDB_{del}$ over SNNDB [24] and point-based deletion scheme [Point-based deletion signifies the execution of $BISDB_{del}$ (most effective batch incremental deletion algorithm) with batches of size one].

Table 5.5: Datasets description.

Dataset	Size	#Attributes	Description
Mopsi12	13000	2	Search locations in Finland
5D	100000	5	Synthetic dataset
Birch3	100000	2	Gaussian clusters
KDDCup'04	60000	70	Identifying homologous proteins to native sequence
KDDCup'99	54000	41	Network intrusion detection data

We simulated our algorithms in C++ on a Linux platform (Intel (R) Xeon (R) CPU E5530 @ 2.40GHz) with 32GB RAM. The experiments were conducted in following phases.

1. Phase-1: Finding the most effective batch incremental variant (deletion).
2. Phase-2: Prove efficiency of the most effective batch incremental variant (deletion) over point-based deletion scheme.
3. Phase-3: Show that point-based deletion becomes ineffective when larger updates (deletion) are made to the base dataset.
4. Phase-4: Prove efficiency of the most effective batch incremental variant (deletion) over SNNDB [24].

5.8.1 Phase-1: Finding the most effective batch incremental variant (deletion)

We adopted three datasets: Mopsi2012, 5D points set and Birch 3 to conduct the experiments in this phase. For experimental purpose, we defined a new term called Algorithm-Components (AlgoComp). AlgoComp consisted of following components:

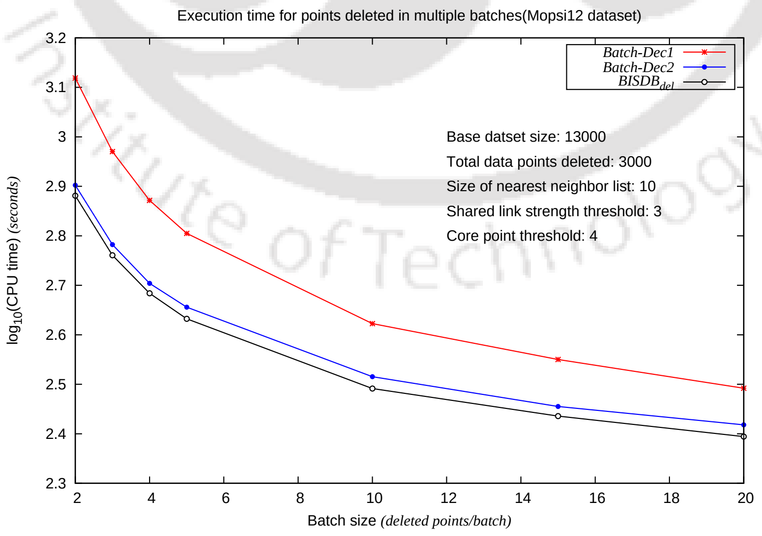
- Base dataset
- KNN lists
- Similarity matrix
- Core and non-core points
- Clusters

A base dataset is one taking which SNNDB [24] is executed in order to set the values of other components in AlgoComp. Initially, the same base dataset is fed as input to the batch incremental algorithms (deletion). After processing a certain batch of updates, the base dataset reduces its size. Points are deleted from the previous base dataset while producing clusters dynamically. The new set of points becomes the updated base dataset over which the next batch outgoing points is processed. The new KNN lists, similarity matrix (K-SNN graph), core and non-core points along with clusters become a part of the updated AlgoComp.

Next we describe the experiments carried out for comparing the proposed batch incremental clustering algorithms: $Batch - Dec1$, $Batch - Dec2$ and $BISDB_{del}$ dataset wise.

1. **Mopsi2012 (Deletion):** The values of parameters were set as: K = 10, δ_{sim} = 3 and δ_{core} = 4, while the value of w was chosen to be 2. We initialized the AlgoComp values by running SNNDB over 13000 points, and then deleted 3000 points in multiple batches incrementally. We started by removing a minimum of 2 points per batch, and continued the experiments upto a batch of size 20. $BISDB_{del}$ turned out to be the most effective deletion algorithm (Figure 5.3) compared to $Batch - Dec1$ and $Batch - Dec2$.

FIGURE 5.3: Mopsi12 dataset: Efficiency comparison between batch incremental deletion algorithms.



While identifying the affected points, we observed that a maximum of around 0.30% $KN - S_{del}$ type and about 0.69% of S_{del} type points were produced while processing batch number 1229 and 898 respectively out of a total of

FIGURE 5.4: Mopsi12 dataset: Average percentage of $KN - S_{del}$ and S_{del} type points for multiple batch insertion of varying batch size ($BISDB_{del}$).

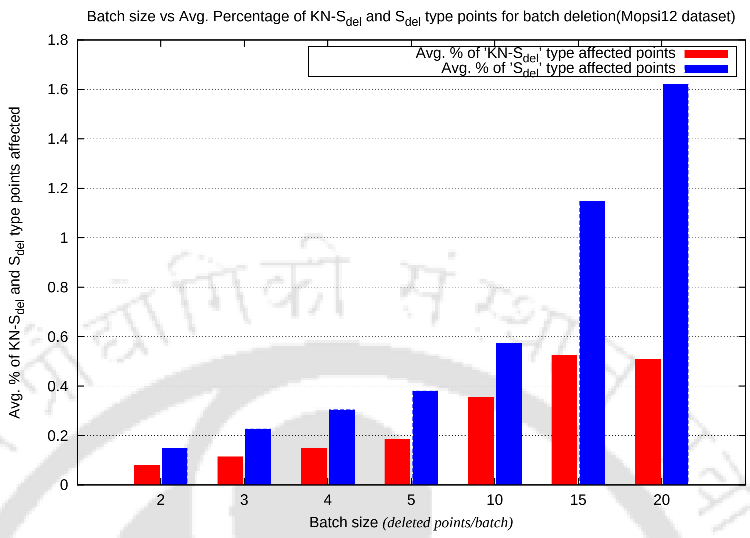
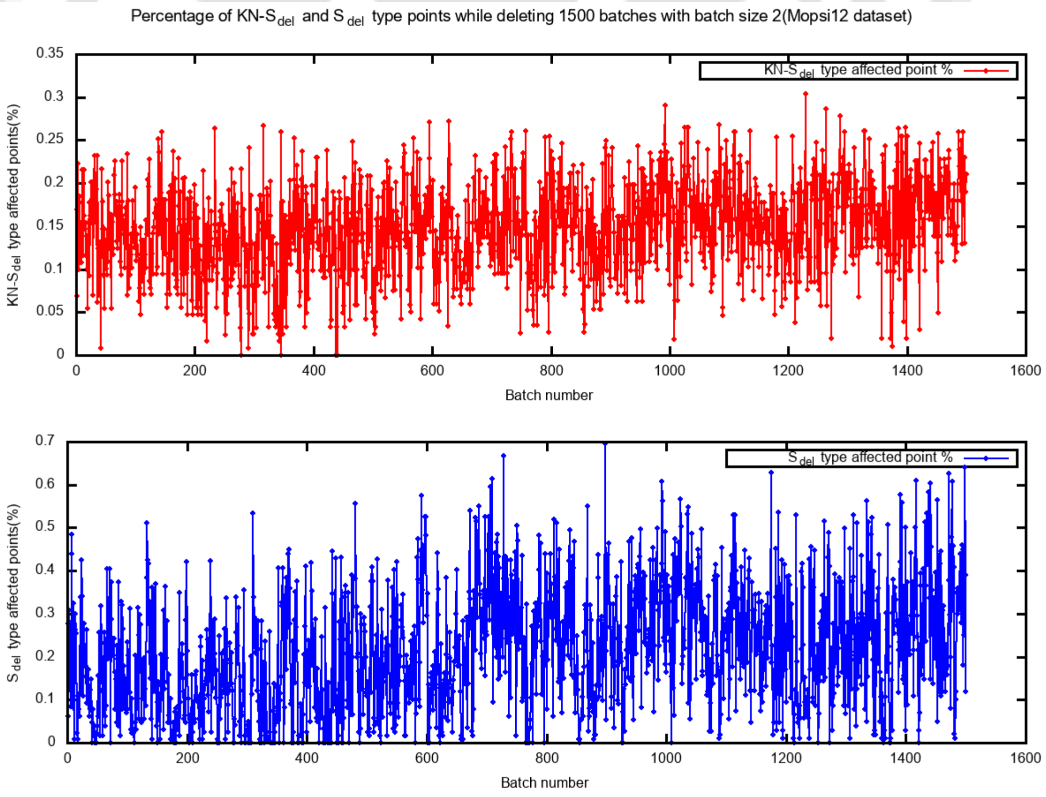


FIGURE 5.5: Mopsi12 dataset: Percentage of $KN - S_{del}$ and S_{del} type points created while deleting 1500 batches ($BISDB_{del}$).



1500 batches (Figure 5.5). The mean percentage of $KN - S_{del}$ and S_{del} type points per batch reached only around 0.5% and 1.6% respectively for a batch of size 20 involving 150 batches in sequence (Figure 5.4).

List of key observation(s) and reason(s) for Mopsi12 dataset (Deletion):

- **Key Observation:** CPU execution time for batch-incremental algorithms reduces with increasing batch size.
  Analysis/Reason(s): With increase in batch size, the total number of batches to be processed decreases. The overall reconstruction time for K-SNN graph, KNN lists, detecting core and non-core points incrementally reduces.

- **Key Observation:** $BISDB_{del}$ achieves the best efficiency.
  Analysis/Reason(s): Constructing updated KNN lists, K-SNN graph, finding core and non-core points incrementally.

- **Key Observation:** Average percentage of $KN - S_{del}$ and S_{del} type points increases with increasing batch size. However the average percentage of $KN - S_{del}$ type points is less than that of S_{del} type points.
  Analysis/Reason(s): With increase in batch size, the number of existing points $(D \cap D')$ affected due to any batch deletion increases.
  Given any batch update, the number of points changing their previous KNN list is less than those affected points which do not change their KNN list. Therefore the average percentage of $KN - S_{del}$ type points is less than that of S_{del} type points.

- **Key Observation:** Percentage of $KN - S_{del}$ or S_{del} type affected points while processing 1500 batches (deletion) remains less than 1%.
  Analysis/Reason(s): The number of points altering their KNN list or those forming new links and breaking existing ones due to any batch update are less than 1% of the size of updated base dataset.

2. **5D (Deletion):** The parameters were set as: K = 4, $\delta_{sim} = 2$, $\delta_{core} = 2$ and $w = 2$. The size of base dataset was taken to be 100000. We deleted 20000 points from the base dataset in multiple batches. While deleting points, we varied the batch size from 2 to 20. Experimentally we observed that $BISDB_{del}$ performed most efficiently (Figure 5.6) out of the three deletion algorithms. When the batch size was 2, $BISDB_{del}$ processed 10000 batches

sequentially. A maximum of around 0.0207% of $KN-S_{del}$ type and 0.0149% of S_{del} type affected points were identified (Figure 5.8) while processing batch number 9030 and 9790 respectively. The average percentage of $KN-S_{del}$ and S_{del} type points per batch was found to be 0.017% and 0.013% respectively for a batch of size 20 involving 1000 batches (Figure 5.7).

FIGURE 5.6: 5D dataset: Efficiency comparison between batch incremental deletion algorithms.

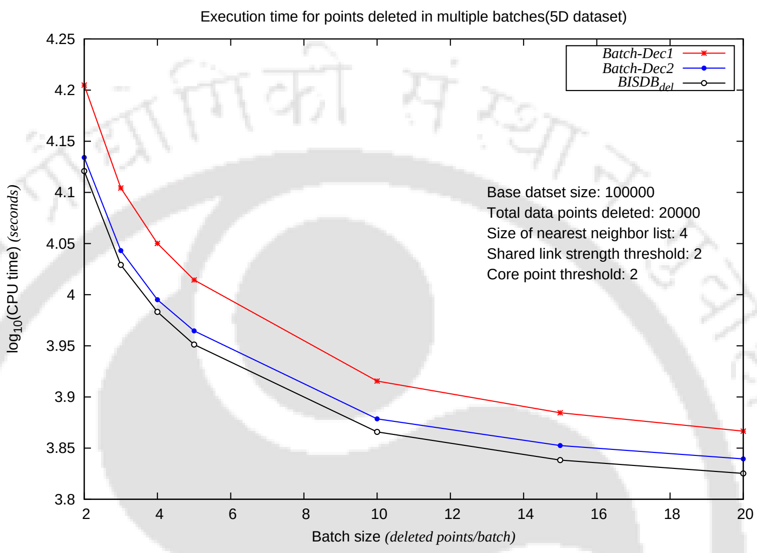
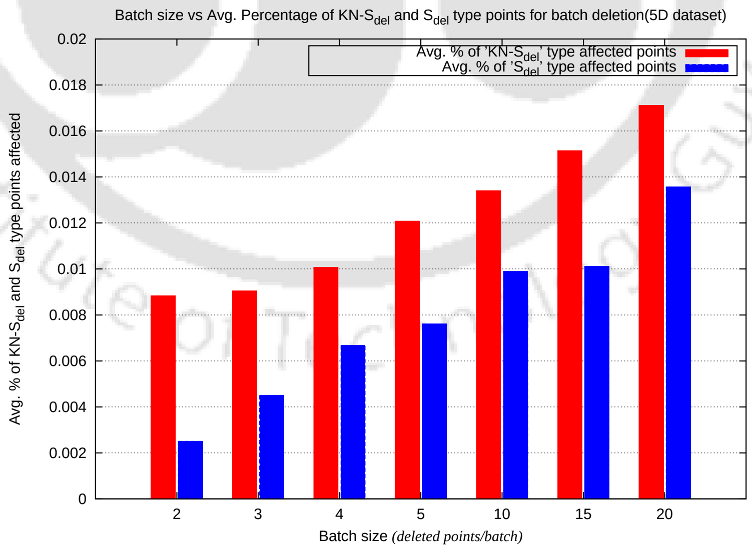


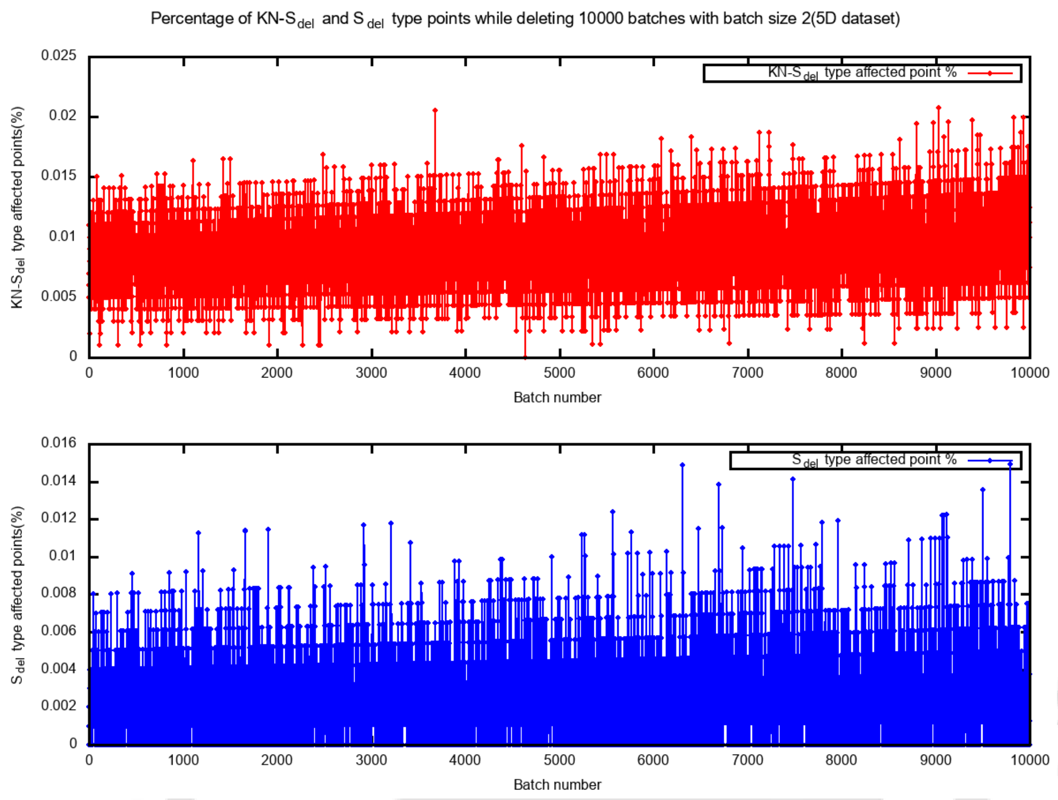
FIGURE 5.7: 5D dataset: Average percentage of $KN-S_{del}$ and S_{del} type points for multiple batch insertion of varying batch size ($BISDB_{del}$).



List of key observations and reasons (Deletion) for 5D dataset:

- **Key Observation:** Average percentage of $KN - S_{del}$ and S_{del} type points increases with increasing batch size. However the average percentage of $KN - S_{del}$ points is more than that of S_{del} type points.

FIGURE 5.8: 5D dataset: Percentage of $KN-S_{del}$ and S_{del} type points created while deleting 10000 batches ($BISDB_{del}$).



Analysis/Reason(s): With increase in batch size, the number of existing points affected due to any batch deletion increases.

Given any batch update, the number of points changing their previous KNN list is greater than those affected points which do not change their KNN list. Therefore the average percentage of $KN - S_{del}$ type points is more than that of S_{del} type points.

- **Key Observation:** Percentage of $KN - S_{del}$ or S_{del} points while processing 10000 batches (deletion) remains less than 0.05%.

  Analysis/Reason(s): The number of points altering their KNN list or those forming and breaking shared links due to any batch update are less than 0.05% of the size of updated base dataset.

3. **Birch3 (Deletion):** The values of parameters were set as: K = 5, δ_{sim} = 2, δ_{core} = 2 and w = 2. We deleted 10000 points from a base dataset of size 100000. We varied the batch size from 2 to 20. $BISDB_{del}$ proved to be the most efficient method out of the three deletion variants (Figure 5.9). For a batch of size 2, exactly 10000 points were deleted in 5000 batches

sequentially. A maximum of around 0.0154% of $KN - S_{del}$ type points and 0.0207% of S_{del} type affected points were identified (Figure 5.11) while processing batch number 4593 and 1707 respectively. The average percentage of $KN - S_{del}$ and S_{del} type points per batch was found to be 0.04% and 0.07% respectively for a batch of size 20 involving 500 batches (Figure 5.10).

FIGURE 5.9: Birch3 dataset: Efficiency comparison between batch incremental deletion algorithms.

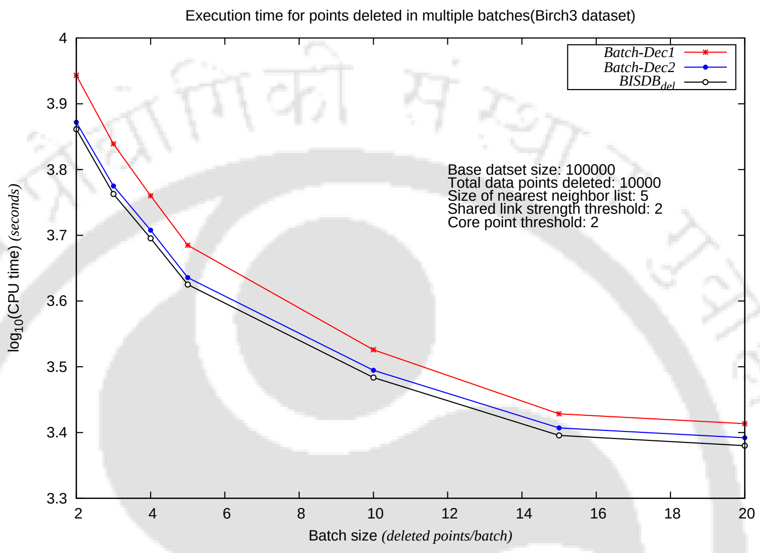
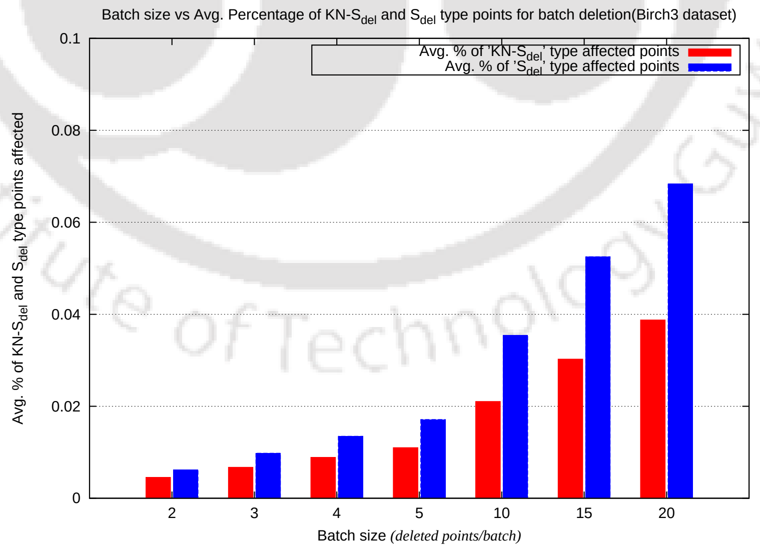


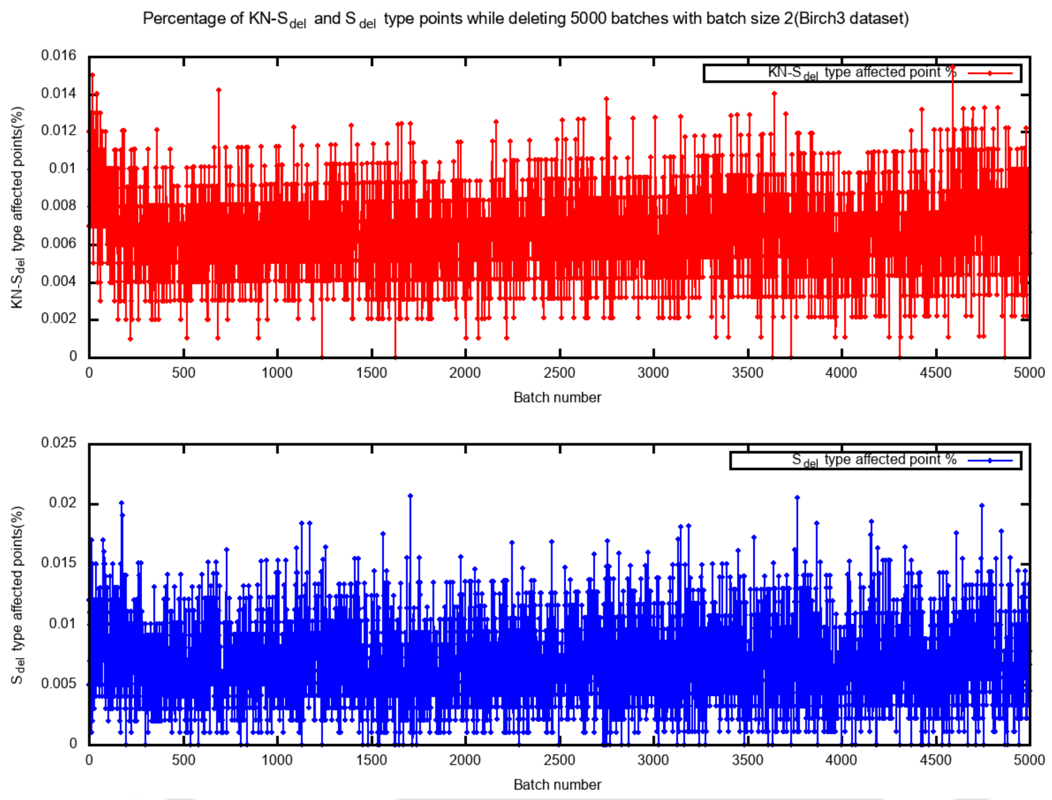
FIGURE 5.10: Birch3 dataset: Average percentage of $KN - S_{del}$ and S_{del} type points for multiple batch insertion of varying batch size ($BISDB_{del}$).



List of key observations and reasons (Deletion) for Birch3 dataset:

- **Key Observation:** Average percentage of $KN - S_{del}$ and S_{del} type points increases with increasing batch size. However the average percentage of $KN - S_{del}$ points is less than that of S_{del} points.

FIGURE 5.11: Birch3 dataset: Percentage of $KN - S_{del}$ and S_{del} type points created while deleting 5000 batches ($BISDB_{del}$).



Analysis/Reason(s): With greater batch size, the number of existing points affected due to any batch deletion increases.

Given any batch update, the number of points changing their previous KNN list is less than those affected points which do not change their KNN list. Therefore the average percentage of $KN - S_{del}$ points is less than that of S_{del} type points.

- **Key Observation:** Percentage of $KN - S_{del}$ or S_{del} points while processing 5000 batches (deletion) remains less than 0.05%.

  Analysis/Reason(s): The number of points changing their KNN list or those forming and breaking shared links due to any batch update are less than 0.05% of the size of updated base dataset.

$BISD_{del}$ therefore proves to be the most efficient batch incremental algorithm (deletion) out of the three proposed variants (Figure 5.12). The selective handling of affected data points in $D \cap D'$ while reconstructing the algorithmic components incrementally leads to a grater efficiency of $BISDB_{del}$. The clusters are detected dynamically with minimal interference on the base dataset.

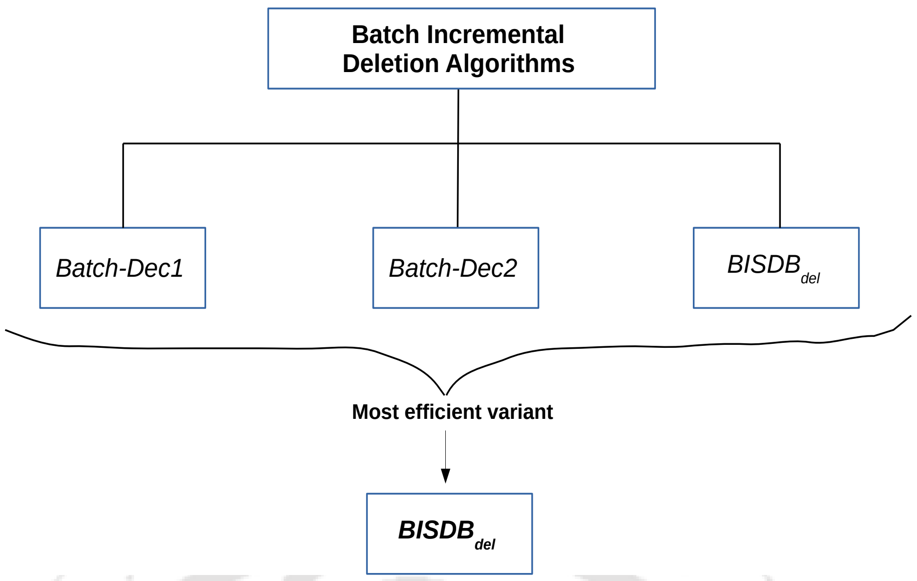


FIGURE 5.12: Summary of Phase-1 experiments for deletion.

5.8.2 Phase-2: Prove efficiency of the most effective batch incremental variant (deletion) over point-based deletion

In this phase, we establish that $BISDB_{del}$ is more efficient than the point-based deletion scheme (executing $BISDB_{del}$ with 1 point/batch) for constant and variable updates made to the base dataset.

Constant updates: In case of constant updates, a fixed number of points were deleted from the base dataset in multiple batches of identical batch size. We compared the efficiency of $BISDB_{del}$ with point-based deletion scheme executing the same number of deletions in a point wise manner (Table 5.6).

TABLE 5.6: Performance comparison of $BISDB_{del}$ and point wise deletion for constant updates.

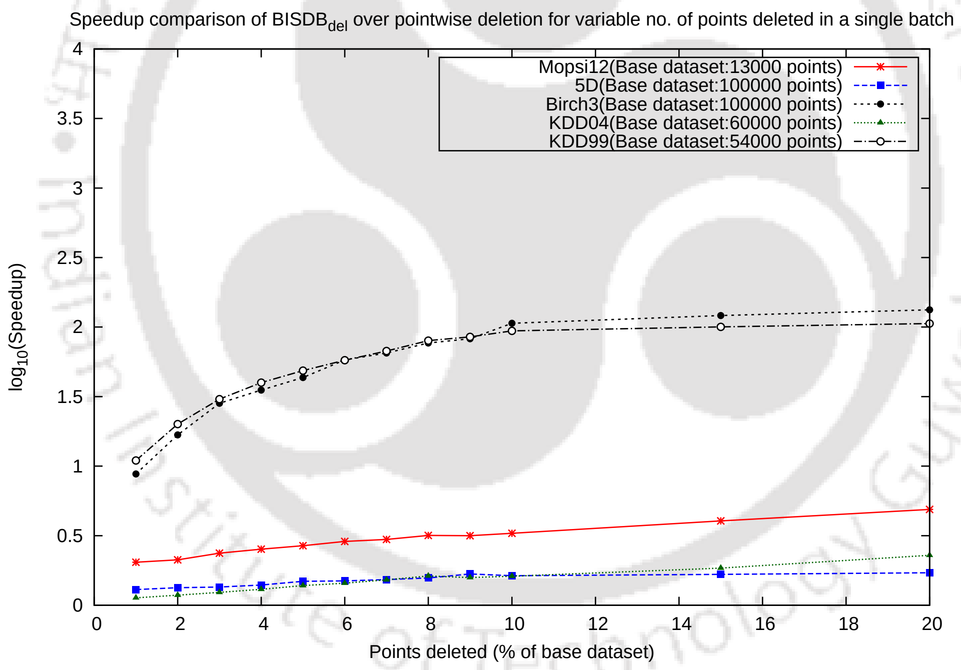
Dataset	Base dataset size	Deleted	Remaining	$BISDB_{del}$ 1p/b (sec)	$BISDB_{del}$ 20p/b (sec)	Speedup ratio
Mopsi2012	13000	3000	10000	1360.02	247.88	5.48
5D points set	100000	20000	80000	19728.95	6686.86	2.95
Birch3	100000	10000	90000	12016.71	2399.10	5.01

Key observation(s): $BISDB_{del}$ deleted 20 points per batch for three datasets: Mopsi12, 5D points set and Birch3 to make up for the total number of points to be removed. For each dataset, the batch incremental method proved to be more efficient than the point based deletion approach.

Analysis/Reason(s): In case of point-wise deletion, the data points are removed one at a time. The construction of KNN lists and the K-SNN graph take place after every deleted point. The detection of core, non-core points and the clusters is also repetitive of the number of points removed. Although the size of base dataset decreases, the time required to remove existing links, compute the altered weights of retained links and split clusters after every point deleted makes the overall process slow as compared to $BISDB_{del}$. If $T_{BISDB_{del}}$ and $T_{point-del}$ are the final CPU execution times after requisite number of points have been removed, then experimentally we showed that $T_{BISDB_{del}} < T_{point-del}$ (Refer Table 5.6).

Variable updates: A variable number of deletions were made from the base dataset in a single batch. We executed $BISB_{del}$ by making deletions ranging from 1% to 20% of the base dataset in one batch. Each time an update was inflicted, the efficiency of $BISB_{del}$ was compared with that of the point-based deletion approach (Figure 5.13).

FIGURE 5.13: Speedup comparison of $BISDB_{del}$ with point-based deletion for variable number of points deleted in a single batch



Key Observation: On measuring the speedup of $BISDB_{del}$ with point-based deletion (Figure 5.13), we observed a tendency of increase as the percentage of updates (deletion) made to the dataset increases.

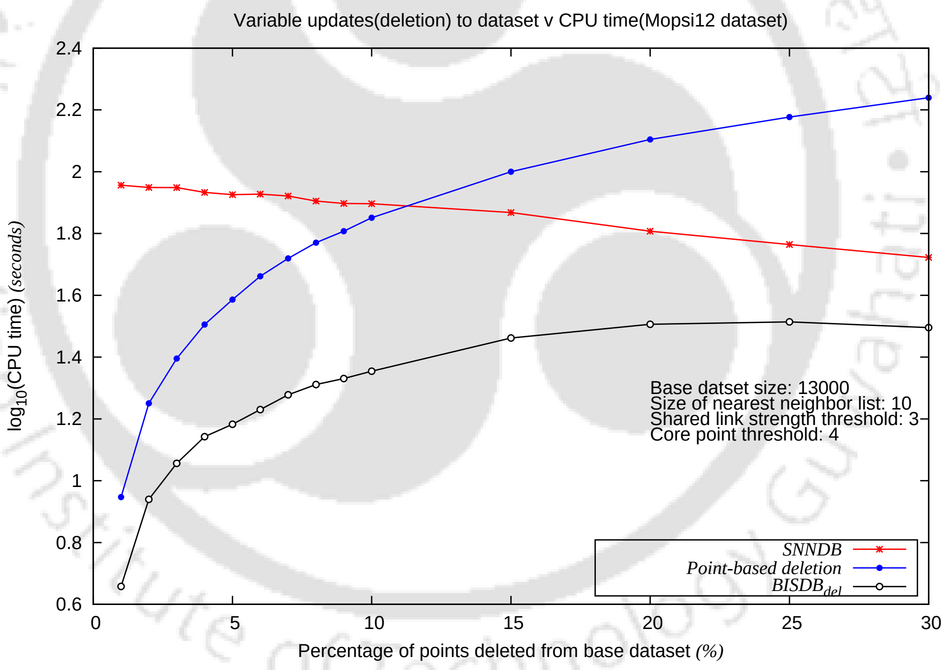
Analysis/Reason(s): For a point-based deletion approach, the process of removing old links, splitting clusters, finding new link strengths of existing ones happen as many times as there are total number of deletions. Due to these computations, $BISDB_{del}$ being a batch processing algorithm, scores on efficiency over point-based scheme as the percentage of update increases.

The $BISDB_{del}$ algorithm therefore outperforms point-based technique for constant or variable updates made to the base dataset, $\therefore T_{batch-del} < T_{point-del}$.

5.8.3 Phase-3: $BISDB_{del}$ and SNNDB are more effective than point-based deletion when large changes are made to the base dataset

Through experimental observations, we establish that when larger changes are made to the base dataset, the naive method (SNNDB) and the batch incremental algorithm ($BISDB_{del}$) outperform the point-based deletion scheme. For illustration, we used the Mopsi12 dataset to demonstrate this property (Figure 5.14).

FIGURE 5.14: Execution time of SNNDB, point-based deletion and $BISDB_{del}$ for variable updates (deletion) in a single batch.



We implemented the $BISDB_{del}$ algorithm and point-based deletion scheme by taking a base dataset of size 13000. While executing $BISDB_{del}$, we removed data points in a single batch, with batch size varying from 1% to 30% of the base dataset. Identical number of deletions were made in a point wise manner. Corresponding to every batch deletion, the CPU execution time of $BISDB_{del}$ was compared with the point based approach.

We also executed the SNNDB [24] algorithm and compared its efficiency with $BISDB_{del}$ and the point based method. From Figure 5.14, we identify that both $BISDB_{del}$ and the point-based scheme maintained a better efficiency than SNNDB [24] till a certain stage. However when the extent of deletion exceeded 11% of the base dataset, the non-incremental SNNDB started achieving a better efficiency than the point-based method. The $BISDB_{del}$ algorithm consistently outperformed both the point-based technique and SNNDB for deletions of all batch sizes. Point-based incremental approach therefore fails to achieve a better efficiency when larger updates are inflicted upon the base dataset.

5.8.4 Phase-4: Prove the efficiency of $BISDB_{del}$ over SNNDB

In this phase, both constant and variable updates were made to the base dataset in multiple batches. For constant updates, a fixed number of points were deleted from the base dataset in multiple batches. The variable updates were made in a single batch with the batch size varying from 1% to 20% of the base dataset. We then measured the speedup of $BISDB_{del}$ with the SNNDB algorithm.

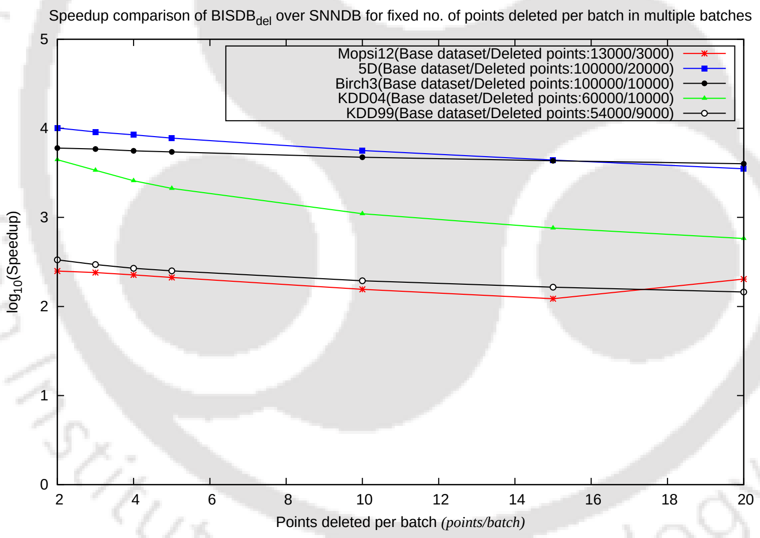


FIGURE 5.15: Speedup of $BISDB_{del}$ over SNNDB for constant deletion.

Key observation(s): $BISDB_{del}$ achieves maximum speed up for smaller updates while the speed up gradually diminishes with increase in size of batch deletions (Figure 5.15, 5.16).

Analysis/Reason(s): While processing smaller updates, $BISDB_{del}$ deals with insignificant percentage of $KN-S_{del}$ and S_{del} type points. As the rest of the points retain their AlgoComp values, post new deletions the time required to reconstruct KNN lists, K-SNN graph and detect clusters incrementally is very less.

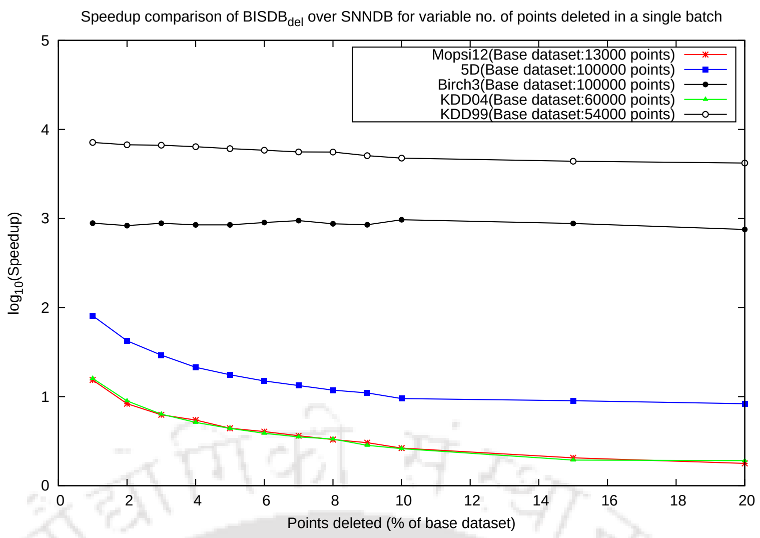


FIGURE 5.16: Speedup of $BISDB_{del}$ over SNNDB for variable deletion.

$BISDB_{del}$ therefore outperformed SNNDB for both constant and variable updates made to the base dataset, $\therefore T_{batch-del} < T_{naive}$. However, the efficiency was obtained at the cost of memory overhead. $BISDB_{del}$ consumed a maximum of 60% more memory than SNNDB for Mopsi12 dataset. The average memory overhead due to $BISDB_{del}$ over SNNDB is around 41.08%, $\therefore |Mem_{batch-del} - Mem_{naive}| \approx 0.41$.

5.9 Cluster analysis

In this section we present the details of clusters that were obtained after executing the naive method (SNNDB [24]), point-based deletion and the batch incremental algorithm $BISDB_{del}$. We compared the number of clusters, core and non-core points along with the outliers that were obtained from executing respective algorithms (Tables 5.7,5.8). Based on the tabular results, it is evident that the set of clusters obtained from running the naive method and the incremental methods for deletion are identical.

TABLE 5.7: Cluster details of SNNDB for all datasets to compare with the incremental deletion algorithms.

Dataset	Algorithm	Input size	#Clusters	#Core points	#Non-core points	#Outliers
Mopsi12	SNNDB	10000	815	3999	1249	4752
5D		80000	807	1339	1023	77638
Birch3		90000	10302	61491	17639	10870
KDD'99		45000	428	2312	744	41944
KDD'04		50000	108	153	224	49623

TABLE 5.8: Cluster details of point-based deletion and $BISDB_{del}$ for all datasets.

Dataset	Algorithm	Base dataset/ Deleted points	#Clusters	#Core points	#Non-core points	#Outliers
Mopsi12	$BISDB_{del}$ / point-based deletion	13000 / 3000	815	3999	1249	4752
5D		100000 / 20000	807	1339	1023	77638
Birch3		100000 / 10000	10302	61491	17639	10870
KDD'99		54000 / 9000	428	2312	744	41944
KDD'04		60000 / 10000	108	153	224	49623

Next we analyze the correctness of clusters obtained from $BISDB_{del}$ w.r.t., SNNDB [24] and point-based deletion based on the following Lemmas.

Lemma 5.1. *Post k number of updates (deletion), the base dataset D ($|D| = n$) changes to D'. KNN(D') obtained from $BISDB_{del}$, SNNDB [24], point-based deletion are identical.*

Proof: After k^{th} update, only the $KN - S_{del}$ type affected points change their KNN lists and find the appropriate top K closest points ($BISDB_{del}$). Similarly, if all the $n - k$ points are taken into consideration, only the top K closest points would occupy the appropriate positions of KNN(x) $\forall\, x \in D'$ (SNNDB). Point-based approach adopts a similar incremental policy of KNN list computation post any deletion.

Lemma 5.2. *$\forall\,(x, y) \in D'$ (updated dataset), $sim(x, y)$ obtained from $BISDB_{del}$, SNNDB [24], point-based deletion are identical.*

Proof: KNN(D') post k number of updates are identical for each class of aforementioned algorithms (Lemma 5.1), therefore the number of shared data points ($sim(x, y)$) between the KNN lists $\forall\,(x, y) \in D'$ for $BISDB_{del}$, SNNDB and point-based scheme are also identical.

Lemma 5.3. *$\forall\, x \in D'$, $Core(x), Non{-}core(x)$ obtained from $BISDB_{del}$, SNNDB [24], point-based deletion are identical.*

Proof: $\forall\,(x, y) \in D'$, $sim(x, y)$ is identical for each class of aforementioned algorithms (Lemma 5.2). As a result, the number of strong links adjacent to a point is also identical for these algorithms, $\therefore \forall\, x \in D'$, x will retain its same core or non-core status for $BISDB_{del}$, SNNDB and point-based deletion.

On the basis of Lemmas 5.1 5.2, 5.3, it can be concluded that the set of clusters obtained from $BISDB_{del}$, SNNDB [24] and point-based deletion are identical. Therefore we may have the following observation(s):

- $C_{batch-del}(C_{BISDB_{del}}) = C_{naive}(C_{SNNDB})$
- $C_{batch-del}(C_{BISDB_{del}}) = C_{point-del}$

where C_A represents the set of clusters for a given algorithm A.

5.9.1 Clustering results in brief

The Mopsi dataset contained about 40% core points. However post deletion in batch mode, nearly 47.5% of points remained as outliers. This is because removal of data points leads to splitting of clusters along with simultaneous breaking of shared strong links. As a result most of the non-core data points fail to attach with a core point so as to obtain a cluster membership. For 5D synthetic dataset, around 97% of points are categorized as noise points. In case of Birch3 dataset, about 87.9% of data points belong to a cluster with an average cluster size of 7.68.

For KDD'99 and KDD'04 dataset, a large share of data points comprised of outliers. In KDD'04 around 99% of points were outliers with a maximum cluster size of eight. The clusters obtained from the algorithms are parameter dependent. The parameter values were specified while conducting experimental evaluation for individual datasets (Refer Section 5.8.1). The results provided are based on parameter values set prior to any execution. Any change in these values may alter the clustering output along with the set of core, non-core and noise points.

5.10 Conclusion

InSDB [1], an existing incremental extension of SNNDB fails to handle dynamic deletion of data points. This is a major flaw on the part of InSDB which acts as a motivation behind designing of the $BISDB_{del}$ algorithm.

The SNNDB clustering algorithm computes the KNN list, similarity matrix, core and non-core points while detecting clusters of arbitrary shapes, sizes and densities. In order to produce an incremental extension to SNNDB in batch mode, each of these components needs to be computed incrementally. $Batch - Dec1$ and $Batch - Dec2$ correspond to the first two batch incremental extensions (deletion) while $BISDB_{del}$ proved to be most effective algorithm out of three incremental variants.

We also showed that for smaller updates $BISDB_{del}$ maintains a higher efficiency. The efficiency gradually diminishes with increase in the size of data updates. We also observed that for variable updates made to the base dataset, $BISDB_{del}$ consistently outperformed both SNNDB and point-based deletion, $\therefore T_{batch-del} < T_{naive}$ and $T_{batch-del} < T_{point-del}$. The cost involved in average memory consumption was about 41.08%. The set of clusters produced by SNNDB and the incremental methods viz. $BISDB_{del}$ and point-based deletion are identical thereby proving the objectives as stated in Section 5.4. For each of the five datasets, we computed the clusters, core and non-core points along with the outliers. While Mopsi12 dataset involved a significant percentage of clustered points, KDD'99 and KDD'04 consisted mostly of outliers.

Chapter 6

KAGO: An Approximate Incremental Grid-Based Outlier Detection Approach using Kernel Density Estimate

In the previous three chapters, we focused on incremental density based clustering algorithms. These algorithms dealt with point-wise or batch mode updates. However, with increasing necessity of finding exceptions in many real time applications (Refer Section 1.3.2), in this contribution we only focus on extracting outliers from dynamic data.

A prominent non-incremental method viz. the K-Nearest Neighbor (KNN) Outlier Detection [5] (KNNOD) algorithm relies on distance based approach to extract outliers. Use of distance based measures for any given data mining task in a multi-dimensional space may produce inappropriate results [24, 64]. Moreover, the KNNOD algorithm involves a quadratic running time for its completion. As a result, any form of redundant computation due to frequent updates may lead towards inefficiency in detecting outliers. In order to address these challenges, we propose an approximate incremental outlier detection algorithm viz. KAGO (Adaptive Grid based Outlier detection using Kernel Density Estimate). KAGO facilitates single point insertions. The proposed algorithm computes the point density using a kernel function. KAGO prunes the inliers and filters the candidate grid cells with local outliers upon any new insertion. The grid cells containing potential outliers are aggregated to extract at most top-N global outliers.

6.1 Motivation

Outlier detection finds its importance in a wide range of applications such as network intrusion detection [38, 77], credit card transactions [39], healthcare [41], detecting faults in safety critical systems [42] to name a few. In all the aforementioned applications, there exists a possibility of frequent data updates in a dynamic environment [78, 40, 79, 11]. For example, consider the following scenario: a credit card transaction in a place far from its usual location of use may indicate a fraud. Similar transactions carried out in such unexpected locations in course of time may reaffirm the involvement of fraudulent means. On the contrary, as the count of such transactions increases from a particular new location, the prior usage of credit card from this new place may appear legal instead of being suspicious. In both the cases, a new transaction can be mapped to the entry of a data point upon base dataset. As the number of transactions increases with the passage of time, we may either have new outliers (fraudulent transactions from a new location) or conforming patterns of data (expected transactions from a usual place).

However against every new insertion, involving all the data points in their entirety may lead to following disadvantages:

- With every insert inflicted on the base dataset, the run time of the outlier detection algorithm may increase disproportionately.
- Increase in updates may lead to a higher consumption of computing resources.
- A delayed extraction of outliers due to processing of data in totality against frequent changes.

The KNNOD [5] algorithm relies on distance based measure to extract outliers. However with the size of data growing continuously, the discovery of outliers in an automated manner becomes crucial. Against every new update made to the base dataset, a KNN based approach may require re-computation of the K-Nearest Neighbors for each data item. Since KNNOD has a quadratic time complexity, such an approach may suffer from having high response time in a dynamic environment.

In our pursuit of developing an incremental solution to KNNOD [5], we aim to extract density based outliers using a probabilistic measure. This is due to the fact that in variable density regions, distance based methods are often rendered

Table 6.1: Motivation behind developing the KAGO algorithm.

Motivation	Description
High response time	Non-incremental algorithms suffer from the issue of redundant computation while handling dynamic data. They involve the entire dataset against every new update leading to a high response time.
Small frequent Updates	When fewer number of insertions are inflicted upon a larger base dataset, the changes in output (anomalies) are also expected to be minimal. As a result, there is no need to process the dataset in its totality.
KNNOD [5] fails in efficient hanling of point insertion dynamically	KNNOD [5] involves quadratic time in detecting outliers. Due to addition of data points one at a time, the anomaly detection process may get slower as the volume of base dataset increases. The time of checking outliers is bound to increase with the size of base dataset. As a result, there is a need to process updates intelligently to quicken the outlier detection process against new insertions.
KNNOD [5] relies on inefficient distance based technique	Distance based methods show their limitations in extracting outliers [23, 80, 5] in regions of variable density. Therefore adopting a density based approach by relying on statistical properties of data (eg:Kernel Density Estimate (KDE)) may lead towards robustness.

inappropriate [5, 80] while filtering outliers. Our proposed technique therefore revolves around the use of probability density approximation method viz. Kernel Density Estimation (KDE) [29, 51, 81, 82, 83] while finding point densities. Table 6.1 provides a brief description about the motivation behind our work.

6.1.1 Chapter contributions

The key contribution(s) made in this chapter may be summarized as follows:

1. We propose an approximate incremental solution to KNNOD [5] in form of the KAGO algorithm (Table 6.2).
2. Use grid structure for subspace creation instead of any expensive clustering technique.
3. Facilitate inlier pruning by focusing on a set of candidate outlier grids (COG). This prevents unnecessary checking of inlier points for not being anomalous.
4. Introduce an opportunistic scheme for kernel center (representative points, see Section 6.3) selection. This enables a cost effective design of the KAGO algorithm.
5. A brief outlier analysis is also provided.

Table 6.2: Brief overview of the baseline method and our proposed approach (Refer Section 6.3 for definitions of related concepts).

Algorithm	Brief working mechanism	Limitations	Improvement
KNNOD [5]	Finds the distance to the Kth nearest neighbor of any point. On the basis of this distance, the outlierness status of a point is determined.	Quadratic time complexity. Re-computation of outliers may be expensive. High resource consumption with every update. Inappropriate for variable density subspaces.	KAGO
KAGO	Divides data-space into grids. Computes point density using KDE. Prunes inlier grids to improve efficiency. Extracts local outliers to find the top-N global outliers incrementally without much loss of accuracy.	Possible memory overhead. Sensitive to size of grid cells.	

6.2 Related work and background

A study regarding incremental outlier detection known as iLOF [47] provides an insight about handling high velocity data streams. A similar work I-IncLOF [48] considers sliding window to designate a set of points as inliers or outliers. The concept of KDE is employed in another method involving data streams [51] for detecting local outliers. Use of K-nearest neighbor classification is also done [84] for detecting outliers from HTTP traffic data. The approach in this work incrementally learns the new traffic anomalies with advent of more data samples. An alternative novel incremental outlier detection approach known as GEM [85] efficiently detects anomalies at a given level of significance. Moreover, for detecting online anomalies in unmanned vehicles, another research [86] has been carried out with encouraging results. We also studied a traditional KNN based anomaly detection approach for finding outliers in a large scale traffic data [5]. Few prominent distance based algorithms were also proposed [87, 88] to extract outliers from large datasets and are scalable over voluminous data streams [89].

Most of these existing outlier detection techniques adopt a distance based approach and involve streaming data. Moreover, a lot of these methods also end up detecting only local outliers. Our grid based approach coupled with the usage of KDE provide a global outlier detection algorithm without neglecting regional sub-spaces.

TABLE 6.3: Major notations used in the chapter (fourth contribution).

Notation	Description
O	Set of outliers prior to any changes in dataset
O'	Set of outliers after dataset is changed
D	Original (Base) dataset
D'	Changed dataset after a point insertion.
i	Number of new insertions
$Outlier$ (.)	Set of outliers
Lc_j	j^{th} kernel center
p	No. of partitions per dimension
d	No. of dimensions
S	The set of points within a grid cell

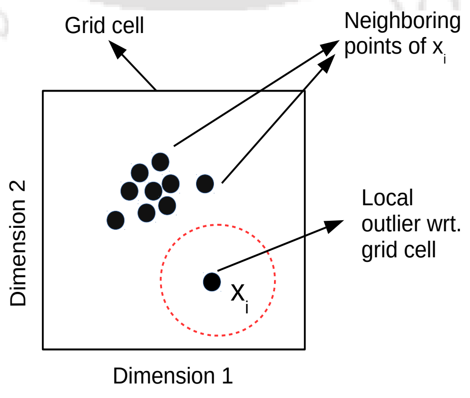
6.3 Preliminaries and Definitions

In this section, we define the key terms and concepts used in this contribution (Refer Table 6.3 for the meaning of notations used henceforth).

6.3.1 Local outlier:

A point x_i is termed as a local outlier if the data density at x_i is substantially lower than the densities at x_i's neighboring points. As shown in Figure 6.1, the density at x_i appears relatively lesser than that of its neighboring points. Therefore point x_i is more likely to be a local outlier, where the neighborhood taken into consideration is the containing grid cell.

FIGURE 6.1: Illustration of local outliers within a grid cell using localized density.



Following steps are involved in designating a point x_i as local outlier:

1. Compute the density (local density)[1] at point x_i along with the densities at x_i's neighboring points.
2. Estimate x_i's local outlierness score. This is based on the deviation of density at x_i contrary to those lying in its neighborhood.

6.3.2 Neighborhood:

For any point $x_i \in D$ (base dataset) $[1 \leq i \leq n, |D| = n]$ and a grid cell $g_c, c = 1, 2, 3, \ldots, p^d$ (d being the number of dimensions), x_i's neighborhood or local neighborhood is represented by the corresponding g_c that contains x_i.

6.3.3 Kernel centers:

Let S be any sample of data ($S \subseteq D$), then we denote a kernel center by $Lc_j, 1 \leq j \leq m$, such that $|S| = r$ and $m \leq r$ ($j, m, r \in \mathbf{Z}^+$). Typically S represents the set of points within a grid cell g_c and Lc_j is a data point sampled from S.

For any given kernel center, there exists a kernel function K_h. The influence Lc_j on the density of a point $x_i \in S$ $[1 \leq i \leq r]$ is estimated on the basis of distance [2] from the kernel center to the concerned point x_i and is given as: $K_h(|x_i - Lc_j|)$.

6.3.4 Kernel Density Estimate (KDE):

The Kernel Density Estimate (KDE) is a non-parametric method applied to compute the probability density function (PDF) of any data sample $S = \{x_1, x_2, x_3, \ldots, x_r\}$. For any given point $x_i \in S$ $[1 \leq i \leq r]$, the KDE is used to estimate the likelihood of point x_i being drawn from S. The probability estimated through the kernel density estimator may be interpreted as the "point density" at any $x_i \in S$. In context of this work, the overall local density at x_i is given as the average of individual density contributions made by all the chosen kernel centers. The following equation gives the measure of overall local density at x_i.

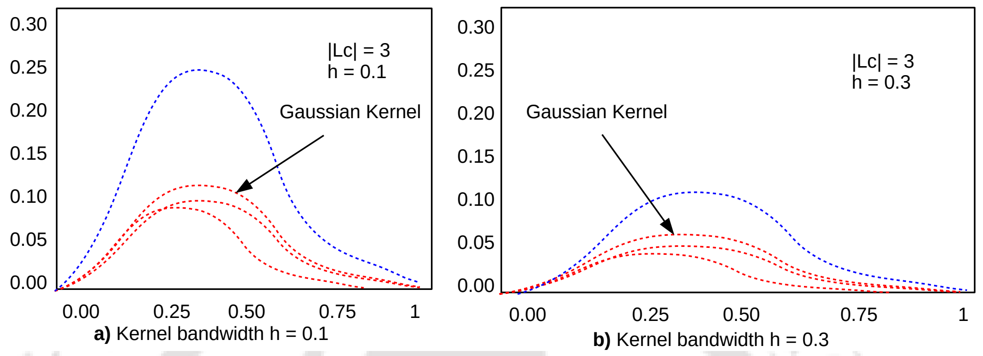
$$f_D(x_i) = \frac{1}{l} \sum_{j=1}^{l} K_h(|x_i - Lc_j|) \tag{6.1}$$

[1]In this contribution, density refers to local density.

[2]We adopt the cosine similarity [90] measure for evaluating the distance between x_i and Lc_j.

where l is the number of influencing kernel centers, $K_h(.)$ represents the kernel function, h is the kernel bandwidth or the smoothing factor and Lc_j represents the j^{th} kernel center.

FIGURE 6.2: Gaussian kernel as univariate KDE with different kernel bandwidths.



For illustrative purpose, in Figure 6.2 we have shown the effect of three kernel centers, each of them a carrying a Gaussian kernel function (red curve). The final density function across all kernels assumes the shape of a curve represented by blue color.

6.3.5 Kernel functions:

A variety of kernel functions may be applied for estimating the density using KDE [29]. The Gaussian kernel [80] is one of the most frequently used kernel functions. In this work, we use the Gaussian kernel as our KDE function (Equation 6.2).

$$K_{Gauss}(v) = \frac{1}{h(\sqrt{2\pi})} exp(-\frac{1}{2}\frac{v^2}{h^2}) \tag{6.2}$$

where v signifies the distance from kernel center Lc_j to the target point x_i. The kernel bandwidth h is also known as the *smoothing factor* that controls smoothness of the curve obtained from the KDE function. A higher value of h ensures a smoother curve of the density function $f_D(.)$. From Figure 6.2(a) (left), we observe that the kernel bandwidth (h) takes a value of 0.1 resulting in sharper curves as compared to Figure 6.2(b) (right) with $h = 0.3$ having flatter curves.

6.3.6 Grid local outlier score ($glos$):

Let $S = \{x_1, x_2, x_3, \ldots, x_r\}$ be the set of points in a grid cell such that $|S| = r$. Then $\forall x_i \in S, i = 1, 2, 3, \ldots, r$, the *glos* value is defined as follows: Given a set of kernel centers $Lc = \{Lc_1, Lc_2, Lc_3, ..., Lc_m\}$ where $Lc \subseteq S$ and $m \leq r$, the *glos* value of a target point x_i is measured as:

$$glos(x_i) = z\text{-}score\left(f_D(x_i), \frac{1}{m}\sum_{j=1}^{m} f_D(Lc_j)\right) \tag{6.3}$$

where $z\text{-}score(P, Q) = \frac{P-Q}{\sigma_Q}$ [91] signifies that if Q is the mean of a set of values, then how many standard deviations the value P is below or above Q. Therefore from Equation 6.3, we observe that the variable P is equivalent to the overall local density at point x_i i.e., $f_D(x_i)$ while variable Q represents the mean local density of all the chosen kernel centers $Lc_j \in Lc$ where $1 \leq j \leq m$. A smaller $glos(x_i)$ value indicates a higher probability of point x_i being an outlier.

6.3.7 Mean grid local outlier score ($mglos$):

Let $S = \{x_1, x_2, x_3, \ldots, x_r\}$ represent the set of points in any grid cell $g_c, c = 1, 2, 3, \ldots, p^d$, then we define the *mglos* value of any g_c by the following equation (Equation 6.4):

$$mglos(g_c) = \frac{1}{r}\sum_{i=1}^{r} glos(x_i) \tag{6.4}$$

6.3.8 Incremental anomaly detection:

Let the initial outlier detection scheme be defined by a mapping f: $D \rightarrow O$ where O represents the set of outliers obtained from the non-incremental algorithm. Let an insertion sequence of k points be made over a base dataset D ($|D| = n, k \ll n$). After k number of insertions let D' be the new dataset, then an incremental outlier detection is defined by a mapping $h : D' \rightarrow O'$, where O' represents the outliers produced from the incremental version. The outliers obtained through $h(D')$ is similar to the one time outliers $f(D')$ produced by the naive algorithm.

6.4 Problem formulation

For k number of insertions where k $\in \mathbf{N}$, let T_{naive} be the total time taken by the non-incremental algorithm and $T_{point-ins}$ be the time taken by the point based approximate incremental method. Let O_{naive} and $O_{point-ins}$ be the respective set of outliers obtained after k number of updates. If $|Mem_{point-ins} - Mem_{naive}|$ be the difference in percentage of average memory consumed, then we aim to establish the following objectives:

- $T_{point-ins} < T_{naive}$
- $O_{point-ins} \approx O_{naive}$
- $|Mem_{point-ins} - Mem_{naive}| \leq \delta$, where δ is a small real number.

6.5 The KNNOD algorithm in brief

The K-Nearest Neighbors outlier detection [5] (KNNOD) algorithm involves distance based approach to extract outliers. For each point $x \in D$ (base dataset), the algorithm identifies the distance of x with its Kth nearest neighbor (d_{K_x}(say)). If $d_{K_{th}}$ is considered as a threshold value, then all the data points whose d_{K_x} value is greater than $d_{K_{th}}$ are considered as outliers while rest of the points remain as inliers.

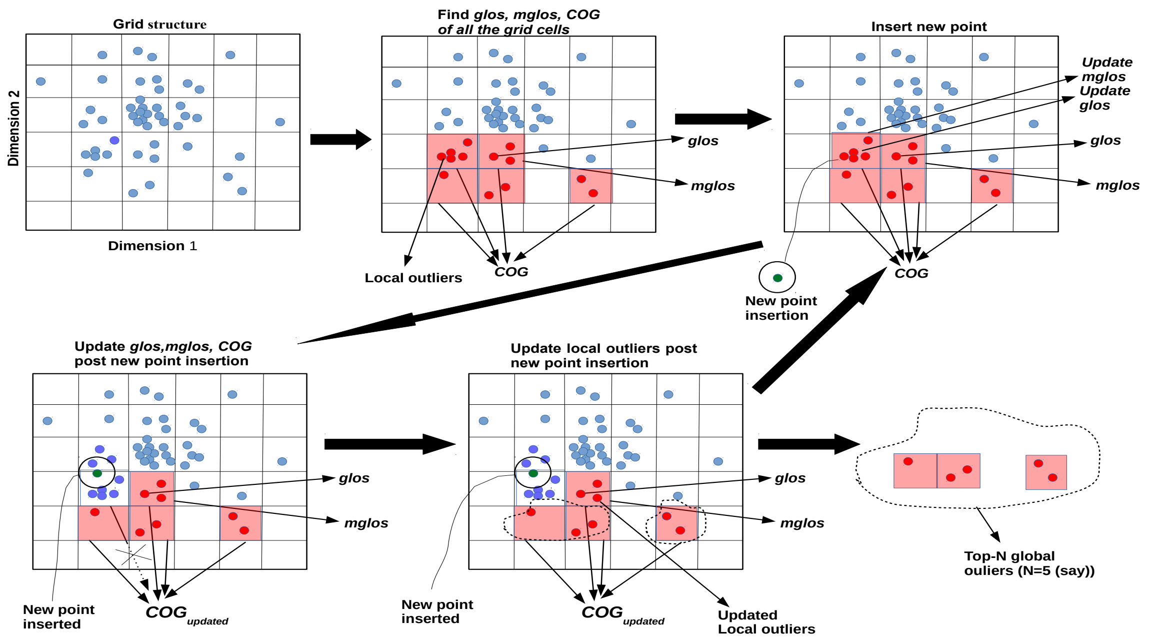
6.6 Framework of the KAGO algorithm

The KAGO algorithm is built around the following framework (Refer Figure 6.3):

1. **Phase-1 Build the grid structure:** Initially, the whole data-space is divided into grids. The usage of grids eliminates the requirement of any clustering [51] approach for separating noiseless points from outliers.

2. **Phase-2 Compute the $glos, mglos$ and COG values**: From the existing grid structure containing base dataset D, the $glos$ value for each data point from all the non-empty grid cells are evaluated using the KDE technique. This is followed by the computation of $mglos$ value for individual grid cell.

Figure 6.3: The KAGO algorithm framework.



Grids with lower *mglos* values within a certain threshold (determined in course of the algorithm) are inducted into the set of candidate outlier grids (*COG*).

3. **Phase-3 New point insertion:** A newly inserted point occupies a single grid cell. As a result, the *glos* and *mglos* values of that concerned grid are updated accordingly. This may result in a change within the set *COG*. A grid cell previously belonging to *COG* may cease to exist within it due to an increased *mglos* value.

4. **Phase-4 Filter top-N global outliers from the updated *COG* set:** From the set of updated *COG*, the potential outlier data points are aggregated. Based on the *glos* value of each data point, at most top-N global outliers are extracted from the current iteration.

5. Repeat Phases 3 and 4 until requisite number data points have been inserted.

6.7 The KAGO algorithm

6.7.1 Theoretical model

Let $O = \{o_1, o_2, o_3, \ldots, o_m\}$ be the set of outliers from base dataset $D = \{x_1, x_2, x_3, \ldots, x_n\}$ such that $O \subseteq D$ and $m \leq n$. Each item in D is represented by a d-dimensional vector $\implies x_i = \{x_{i1}, x_{i2}, \ldots, x_{id}\}$ where $x_{iq} \in \mathcal{R}, 1 \leq i \leq n, 1 \leq q \leq d$. Let x_{n+1} be the new data point added to D, therefore D changes to D'. $D' = \{x_1, x_2, \ldots, x_n, x_{n+1}\}$ becomes the current set of points from which the updated outliers are to be extracted. The new set of outliers can be obtained by applying the KNNOD [5] algorithm on D'. However, we avoid this procedure by developing a less expensive scheme in form of the KAGO algorithm.

Let O_{KNNOD} and O'_{KNNOD} be the set of outliers obtained by executing KNNOD upon dataset D and D'. Let O'_{KAGO} be the set of outliers obtained by executing the proposed KAGO algorithm. If $O_{KNNOD} \leftarrow KNNOD(D)$; $O'_{KNNOD} \leftarrow KNNOD(D'); O'_{KAGO} \leftarrow KAGO(D')$ then we establish the following objectives:

- $Time_{KAGO}(D') < Time_{KNNOD}(D')$.
- $Outlier_{KAGO}(D') \approx Outlier_{KNNOD}(D')$.

6.7.2 Steps of the KAGO outlier detection algorithm

Prior to insertion of any new data point, we execute the first two phases of the framework for KAGO algorithm. This includes creation of the grid structure *GridStruct* (a set of grid cells) containing base dataset D and identification of the set COG wrt. D. We define a new term called Algorithm-Components (Algo-Comp). The *GridStruct* and COG are included as a part of Algo-Comp along with the base dataset D. Once the Algo-Comp values are determined, a new point is inserted. The pre-determined Algo-Comp values are then used by the KAGO algorithm for extracting new set of outliers incrementally.

Next we present the steps of our proposed KAGO algorithm. Each step is supported by necessary explanation as and when required. The only parameter involved is the number of partitions per dimension (p). A grid cell is denoted as g_c where $c = 1, 2, 3, \ldots, p^d$.

1. **Step 1 - Set the Algo-Comp:**

   (a) Load the base dataset D.

   (b) Set the no. of grid cells to p^d.

   (c) Create $GridStruct$ by dividing the dataspace containing D into p^d d-dimensional grid cells.

   (d) Set the variable $VolGrid$ to 2^d+1. $VolGrid$ gives a measure of the minimum number of points per grid cell exceeding which the cell becomes relatively dense.

   (e) $COG = \phi$

   (f) $\forall$ grid cell $g_c \in GridStruct, [1 \leq c \leq p^d]$,

      i. If $(|g_c| = 0)$, process the next grid cell.

      ii. If $(|g_c| > VolGrid)$, where $|g_c| = r(say)$, then the grid cell is considered to be relatively dense. For a dense grid cell g_c, KAGO selects m kernel centers$[m = 3]$ from that cell such that it represents the data distribution within g_c. The chosen kernel centers are:

         - $x_i \in g_c[1 \leq i \leq r]$, and x_i is closest to the centroid of g_c.
         - $x_i \in g_c[1 \leq i \leq r]$, and x_i is closest to the minimal point[3] of g_c.
         - $x_i \in g_c[1 \leq i \leq r]$, and x_i is closest to the maximal point[4] of g_c.

      iii. If $(|g_c| \leq VolGrid)$ with $|g_c| = r(say)$, then the grid cell is considered to be sparse. Under this scenario, m kernel centers$[m \leq r]$ within g_c influence the local density of any point in the grid cell.

      iv. Calculate KDE $\forall x_i \in g_c[i =, 2, 3, \ldots, r](|g_c| = r)$ (Defined in Section 6.3).

      v. $\forall x_i \in g_c[i = 1, 2, 3, \ldots, r]$, compute $glos(x_i)$ (Defined in Section 6.3).

      vi. Compute $mglos(g_c)$ (Defined in Section 6.3).

   (g) Sort the non-empty grid cells in $GridStruct$ in order of their increasing $mglos(g_c)$ value.

   (h) $\forall g_c \in$ top 50% of non-empty grid cells from $GridStruct$ with lowest $mglos(g_c)$ scores (sorted in increasing order), $COG \leftarrow COG \cup \{g_c\}$.

[3]The point within g_c where each co-ordinate in a given dimension is the minimum of all the current points $\in g_c$ in that dimension.

[4]The point within g_c where each co-ordinate in a given dimension is the maximum of all the current points $\in g_c$ in that dimension.

(i) Designate at most top-N points from COG as potential outliers based on their increasing $glos$ value.

2. **Step 2- New point insertion:** A new point $npt(say)$ is inserted upon D. The base dataset D changes to D' where $|D'| = |D| + 1$.

3. **Step 3- Identify the affected grid:** Post insertion of new point npt, KAGO identifies the affected grid cell $g_c[1 \leq c \leq p^d]$ where npt is positioned. As a result the strength of affected g_c, where $|g_c| = r(say)$ increases by one,$\therefore$ $|g_c| = r + 1$.

4. **Step 4- Update the variables related to Algo-Comp wrt. the affected grid cell:**

   (a) For the affected grid cell $g_c \in GridStruct$ where $|g_c| = r + 1$,

      i. If $(|g_c| > VolGrid)$, then the grid cell is considered to be relatively dense. For a dense grid cell g_c, KAGO selects m kernel centers$[m = 3]$ (Similar to Step 1 (f) (ii)) in that cell such that it represents the data distribution within g_c.

      ii. If $(|g_c| \leq VolGrid)$, then g_c continues to be a sparse grid cell. Under this scenario, m kernel centers$[m \leq |g_c|]$ influence the local density of any point within the grid cell.

      iii. Estimate the updated KDE $\forall x_i \in g_c[i = 1, 2, 3, ..., r + 1]$.

      iv. $\forall x_i \in g_c[i = 1, 2, 3, ..., r + 1]$, compute the updated $glos(x_i)$.

      v. Find the updated value of $mglos(g_c)$.

   (b) Sort the non-empty grids cells in $GridStruct$ according to their updated $mglos(g_c)$ value in increasing order.

   (c) Update the set COG by selecting a maximum of top 50% non-empty grid cells[5] from $GridStruct$ according to their increasing $mglos(g_c)$ values. An inlier grid cell prior to addition of npt, will not become a part of COG after insertion, despite having a relatively lesser $mglos$ score.

   (d) If $COG = \phi$, there are no outliers post insertion of npt. Go to **Step 6**.

5. **Step 5- Filter a maximum of top-N global outliers:** Gather all data points from the updated set of COG. Sort the data points in increasing order of their $glos$ score and extract a maximum of top-N global outliers

[5]Post entry of a new point, any grid cell previously belonging to COG might not be a part of it anymore.

(with repeated insertions, the number of existing outliers may be less than N). A point having an inlier status prior to addition of *npt* will not be an outlier after its insertion in spite of having a relatively lesser *glos* value.

6. Set $D = D'$.

7. Repeat **Steps 2 to 6** till k number of points have been inserted.

6.8 Time Complexity of the KAGO algorithm

The first five component steps (Step 1 (a) to 1 (e)) mainly consist of the initialization tasks. These include loading of base dataset (D) and creation of the grid structure. As a result Step 1 involves a constant running time of $\mathcal{O}(1)$. For Step 1(f), two scenarios may arise while processing a certain grid cell g_c. If g_c is dense ($|g_c| > VolGrid$), then the algorithm considers three kernel centers to have an influence for determining the point wise *glos* values. Let *Grids* represent the total number of non-empty grid cells (a constant entity). Considering that the strength of any grid cell does not exceed n ($|D| = n$), a running time of $\mathcal{O}(3r + 3r) \simeq \mathcal{O}(1)$ is required for Step 1 (f (ii)) [$\because r \ll n, |g_c| = r, \therefore \mathcal{O}(3r)$ time each is required for kernel center selection and KDE computation $\forall\, x_i \in g_c,\ i = 1, 2, \ldots, r$ where $r > VolGrid$]. However, for sparse grid cells ($|g_c| \leq VolGrid$) (Step 1 (f (iii))), $\mathcal{O}(r^2) \simeq \mathcal{O}(1)$ time is required given that all the r points within any g_c act as kernel centers. For datasets with certain highly dense grid cells ($r \approx n$), a combined running time of $\mathcal{O}(n * Grids)$ is involved for evaluating $glos(x_i)$ $[1 \leq i \leq n]$.

The *mglos* value for each grid cell takes either $\mathcal{O}(r)$ or $\mathcal{O}(n)$ time (Step 1 (f (vi))) depending on the strength of grid cell. Step 1 (g) involves sorting of non-empty grid cells on the basis of their increasing *mglos* scores, which takes $\mathcal{O}(Grids * \log Grids) \simeq O(1)$ running time [$\because Grids \ll n$]. Step 1 (i) requires a running time of $\mathcal{O}(n \log n)$ or $\mathcal{O}(1)$ depending on the number of potential outliers within *COG*. The following steps: Steps 2 and 3 take $\mathcal{O}(1)$ time where the insertion of new point and its grid cell identification takes place. The next step (Step 4) deals with updation of *glos* value for each data point within the affected cell and *mglos* value for the grid cell itself. Depending on the cell strength, Step (4 (a)) involves a total time of $\mathcal{O}(r^2)$ (sparse grid cell) or $\mathcal{O}(r)$ $[r \leq n]$ (dense grid cell). Sorting of grids post insertion of new point takes $\mathcal{O}(Grids * \log Grids) \simeq O(1)$ time. The new list of *COG* is therefore rebuilt in constant time.

Table 6.4: Datasets and their partition size used in experiments.

Dataset	**#Features**	**#Points**	**Base dataset size**	**#Added points**
NSL-KDD3	3	25000	20000	5000
NSL-KDD4	4	25000	20000	5000
A1-Yahoo! Search	4	18000	15000	3000

In Step 5, all the potential outliers lying within COG are sorted at a cost of $\mathcal{O}(|COG| * r \log(|COG| * r)) \simeq \mathcal{O}(1)$ (Taking an average grid cell strength of r) or $\mathcal{O}(n \log n)[r \approx n]$ based on their increasing $glos$ values. We assume the initial size of top-N outliers as $\sqrt{n}$. Therefore checking a point of its outlier status is done by comparing it with the previous list. This involves a running time of $\mathcal{O}(\sqrt{n * n}) \simeq \mathcal{O}(n)$. The overall time complexity (worst case) of KAGO algorithm is therefore $\mathcal{O}(n + n \log n) \simeq \mathcal{O}(n \log n)$ for a skewed distribution, otherwise $\mathcal{O}(n)$ (given that the strength of each $COG \ll n$). However, with increase in number of insertions, the time taken to extract outliers ($\ll \sqrt{n}$) may reduce significantly with more dense areas in the data space. In this scenario, the final list of outliers may be obtained in constant time.

6.9 Experimental evaluation

6.9.1 Experimental setup and datasets used

We simulated our proposed KAGO algorithm in C++ on a Linux platform (Intel (R) Xeon (R) CPU E5530 @ 2.40GHz) with 32GB RAM. The experiments were performed on two network intrusion detection datasets viz. NSL-KDD (PCA (Principal Component Analysis) [92] reduced to 3 and 4 dimensions respectively)[6] and a bidding data for market advertisement: A1 for Yahoo! Search[7]. For experimental purpose, we used the PCA reduced NSL-KDD dataset by the name of NSL-KDD3 and NSL-KDD4. The Yahoo! search marketing advertiser bidding data has been named as A1-Yahoo! (Refer Table 6.4 for dataset details).

Prior to execution of KAGO, we involve the entire dataset to decide the starting and ending point of each dimension. The minimum and the maximum end for each dimension is determined by the following relations:

[6]https://www.unb.ca/cic/datasets/nsl.html

[7]https://webscope.sandbox.yahoo.com/catalog.php?datatype=a

$$Min = min\{min\{D_1\}, min\{D_2\}, \ldots, min\{D_d\}\} \tag{6.5}$$

$$Max = max\{max\{D_1\}, max\{D_2\}, \ldots, max\{D_d\}\} \tag{6.6}$$

Here, $min\{D_i\}, max\{D_i\}$ $[1 \leq i \leq d]$ represents the minimum and maximum of all the values along i^{th} dimension. The *GridStruct* (Refer Section 6.7) is therefore constructed with its origin containing Min (Equation 6.5) and every dimension extends to an identical length till Max (Equation 6.6). The height of a grid cell (h) in any dimension having $p(p > 0)$ partitions per dimension is given as:

$$h = \frac{Max - Min}{p} \tag{6.7}$$

The purpose of building *GridStruct* with entire dataset is to ensure the positioning of a newly added point within the same grid space as the base dataset.

In case of a dense grid cell, we select three kernel centers (centroid, minimal, maximal) ($\therefore m = 3$, Refer Section 6.7) within the cell to exert influence on each $x_i \in g_c[1 \leq c \leq p^d, 1 \leq i \leq r, r > VolGrid]$ while evaluating x_i's kernel density. In case of a sparse grid cell, we assign each $x_i \in g_c[1 \leq c \leq p^d, 1 \leq i \leq r, r \leq VolGrid]$ to behave as an influential kernel center within $g_c(\therefore m = r)$. Next we describe our observations based on experiments conducted for each dataset.

6.9.2 Experimental Results and Analysis

1. **Observations for NSL-KDD3:** For NSL-KDD3 dataset, the value of parameter p (#partitions/dimension) was taken to be 5. As a result, the total number of grid cells within *GridStruct* was 125 $[\because \#Grids = p^d, d = 3(\#dimensions)]$. After analyzing the entire dataset (25000 points), the Min and Max values were found to be -4.09628 and 111.051 respectively. The origin of the co-ordinate system was therefore located at (-4.09628,-4.09628,-4.09628). The maximal point of *GridStruct* was positioned at (111.051,111.051,111.051). For each of the 125 grid cells, the grid height (h) in every dimension was identical to a value of 23.0295 (Using Equation 6.7).

   Key Result(s): The base dataset size was taken to be 20000 upon which 5000 additional points were inserted one at a time. Prior to insertion of any

FIGURE 6.4: NSL-KDD3 dataset: Efficiency comparison between KAGO and KNNOD.

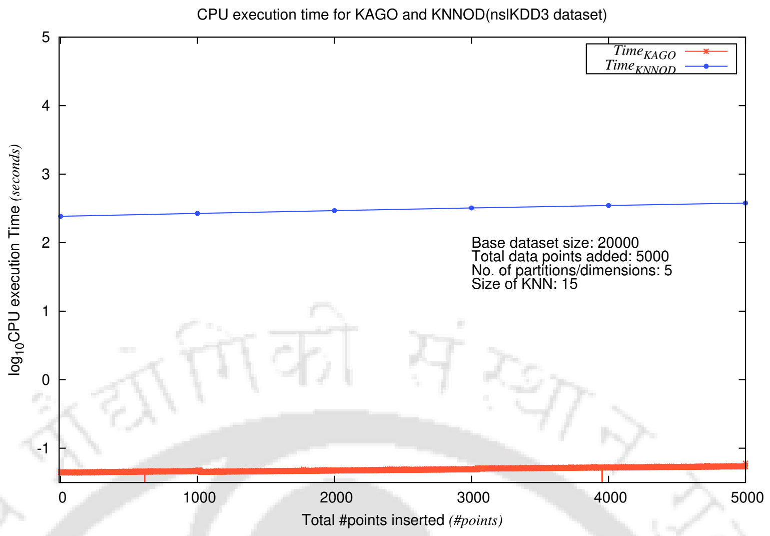
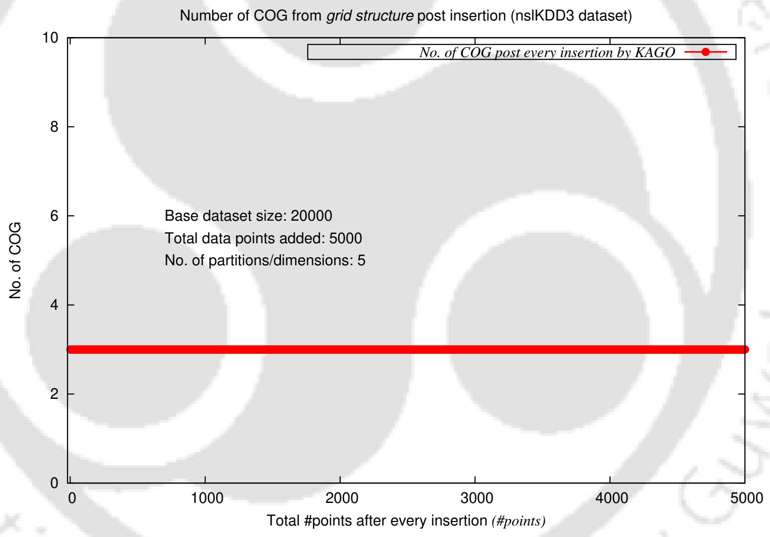


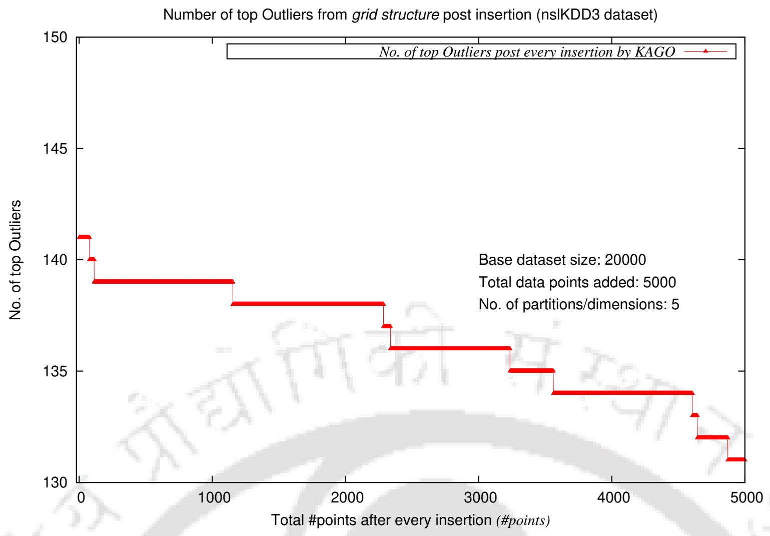
FIGURE 6.5: NSL-KDD3 dataset: Number of *COG*.



new point npt, five grid cells numbered: 1, 26, 51, 76 and 101 were filled with data points. Grid #1 was the only dense grid cell with more than 9 ($\because VolGrid = 2^d + 1$) points. From Figure 6.4, we observe that our proposed KAGO algorithm (lower curve) consistently outperformed KNNOD [5] till the entry of final point. Based on our observation, a maximum speedup of about 6660 ($\approx$ order of 3.8) was achieved by KAGO when the 4000^{th} point was inserted. However a curve dip (Figure 6.4) was observed after 615^{th} and 3954^{th} insertion in case of KAGO.

Reason(s): The high speedup of KAGO can be attributed to its grid based approach of dealing with subspaces instead of the entire dataset scan. The efficiency curve dip (Figure 6.4) for KAGO occurred because two new grid

Figure 6.6: NSL-KDD3 dataset: Maximum of top-N outliers post every insertion for NSL-KDD3 (NSL-KDD PCA reduced to 3 dimensions).



cells (previously empty) viz. Grid# 101 and Grid# 76 were affected when 615th and 3954th point were inserted. These newly affected grid cells remain sparse ($\because |g_{76}| = 4, |g_{101}| = 3$). Therefore the time required to compute the *glos* values of points within the affected cells are much less as compared to the denser grid cells.

Key Result(s): The number of grid cells in COG (set of candidate outlier grids) was consistent throughout the insertion process of all data points (Figure 6.5).

Reason(s): A constancy in the number of grid cells in COG implies that no new cells were affected due to point insertions and that the list of initial COG with probable outliers remain untouched even after repeated addition of points. It may also be the case that a single grid cell in a COG of any iteration had been dismantled by a newly affected cell while rest of the grid cells in erstwhile COG retain their position. No deviation in the number of cells in COG also means that the potential outliers (at most top-N, N=$\lfloor\sqrt{|D|}\rfloor, |D| = 20000$) were selected post every insertion from the three grid cells within COG.

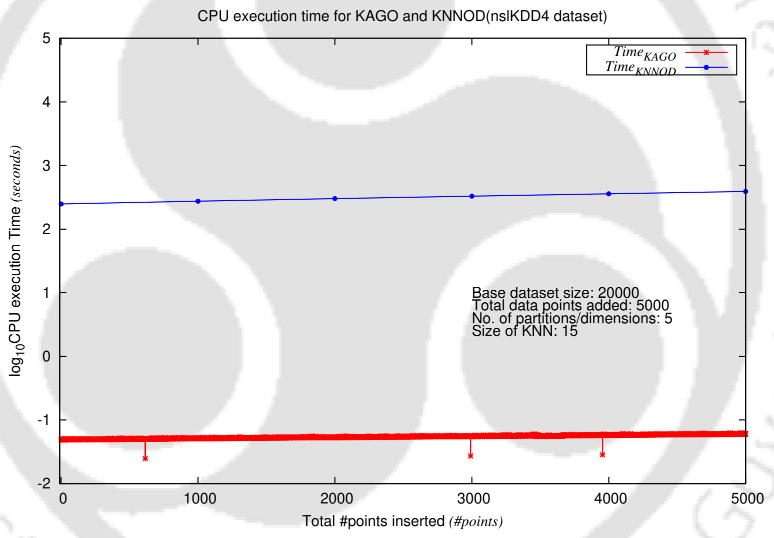
Key Result(s): We observed a steady decrease in the number of top-N outliers from 141 to 131 after all the insertions were made (Figure 6.6).

Reason(s): The reason for this steady decrease in the number of outliers may be attributed to the increase in *glos* values of data points. The *glos* scores increase due to a dense neighborhood within the affected grid cells.

Redundant insertions on certain cells initiate this phenomenon where a previous outlier point may gradually lose its outlier status. This results into a sufficiently filled parent grid cell having a higher *mgols* value.

2. **Observations for NSL-KDD4:** The parameter p was set to 5 resulting in a total of 625 grid cells ($\because d = 4$). The base dataset size was taken to be 20000 facilitating insertion of 5000 points. The *Min* and *Max* values for *GridStruct* were found to be -6.28682 and 111.051 respectively. The grid height equaled 23.4676 in each of the four dimensions. Prior to insertion of any new data point, a total of five grids were filled, with Grid #1 being the densest containing 19985 points.

FIGURE 6.7: NSL-KDD4 dataset: Efficiency comparison between KAGO and KNNOD.



Key Result(s): On observing the results for subsequent insertions from Figure 6.7, we found that KAGO achieved a maximum speedup of around 6445 times ($\approx$ order of 3.8) over KNNOD post insertion of the last point. However a curve dip (Figure 6.7) was observed after 615^{th}, 2991^{st} and 3955^{th} insertion was made. No change was was observed in the number of *COG* throughout all point insertions (Figure 6.8).

Reason(s): Similar reasons as applicable to the previous dataset. The newly affected grid cells in this case were: Grid# 101, Grid# 126 and Grid# 76 respectively.

Key Result(s): For the set of outliers (Figure 6.9), we observed that upon entry of first point, a total of 141 outliers were extracted which reduced to

Figure 6.8: NSL-KDD4 dataset: Number of *COG*.

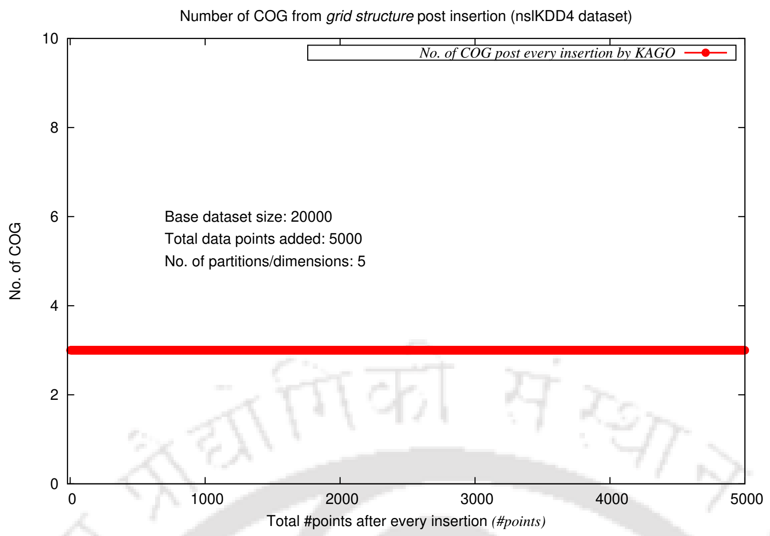
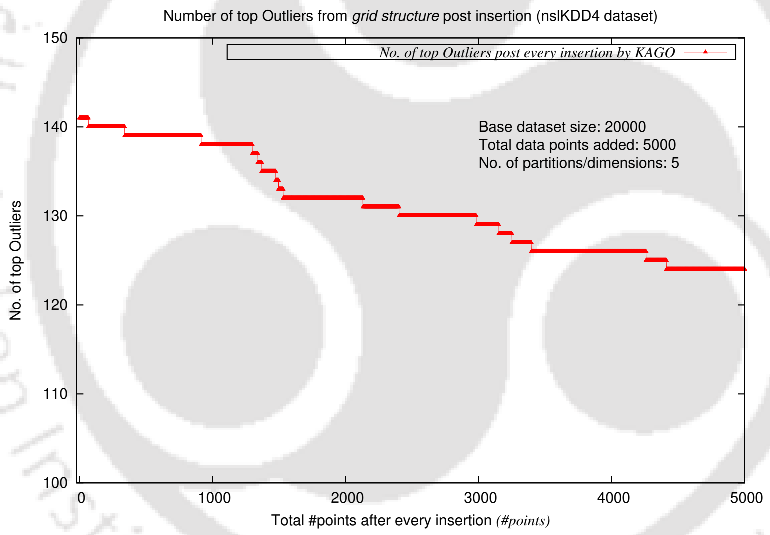


Figure 6.9: NSL-KDD4 dataset: Maximum of top-N outliers post every insertion for NSL-KDD4 (NSL-KDD PCA reduced to 4 dimensions).



124 post entry of the 5000th point.
Reason(s): This is because upon entry of any new point, a sparse grid cell may turn into a dense resulting in an overall increase in the *mglos* score of that affected grid cell. This results in a significant decrease in the number of top-N global outliers.

3. **Observations for A1-Yahoo!:** The parameter p was assigned a value of 5. The dataset being used consisted of four dimensions resulting in a total of 625 grid cells. The base dataset size was fixed at 15000 with an additional 3000 points being added one at a time. The Min and Max values were calculated to be 0 and 1133 respectively with a grid height h equaling 226.6

per dimension. After insertion of all the data points, noticeably for A1-Yahoo! dataset (Refer Figure 6.10), KAGO achieved a maximum speedup of about 8403 times (≈ 3.92) over KNNOD.

FIGURE 6.10: A1 - Yahoo! dataset: Efficiency comparison between KAGO and KNNOD.

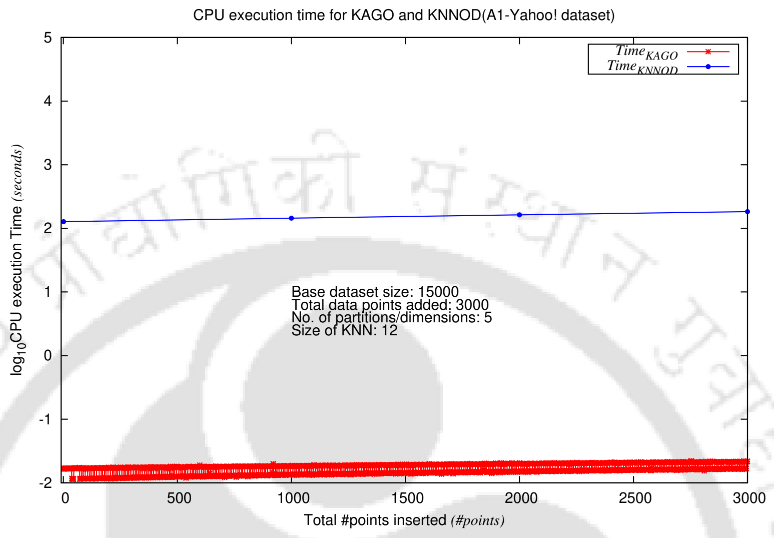
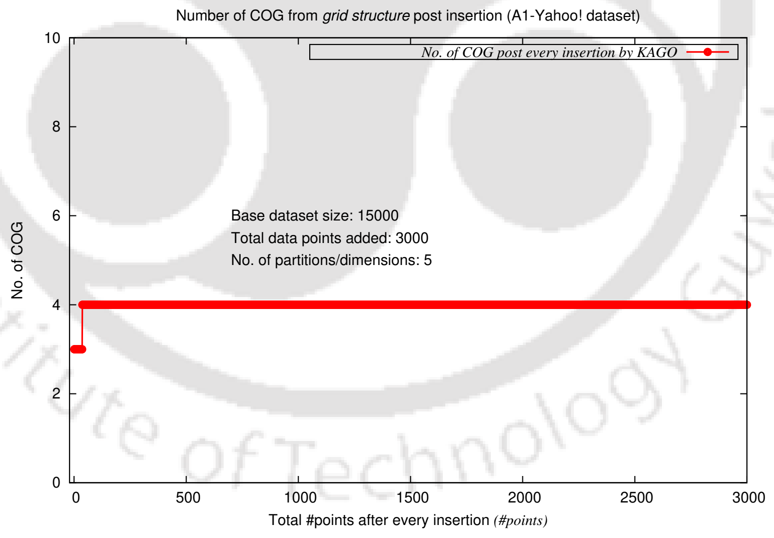


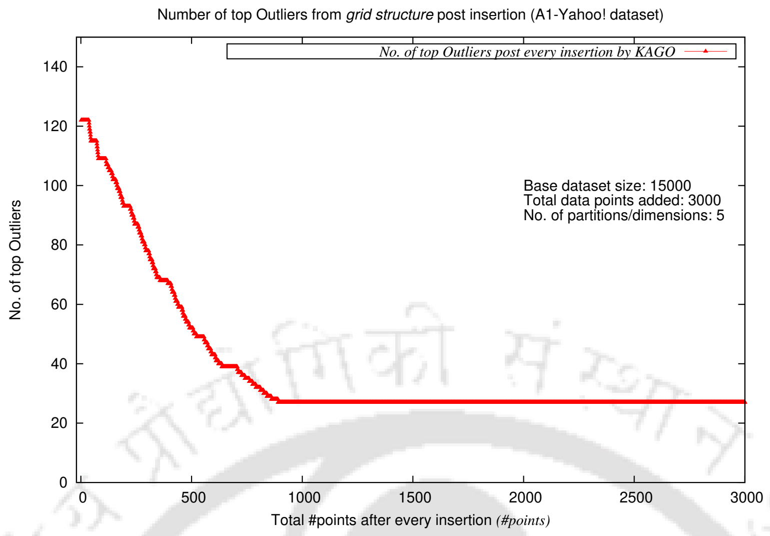
FIGURE 6.11: A1 - Yahoo! dataset: Number of *COG*.



Key Result(s): We observed that there is a periodic dip in the efficiency curve (Figure 6.10) for KAGO over subsequent point insertions.

Reason(s): Initially preceding any insertion, a total of 6 grid cells were filled with data points: Grid #46 (321 points), Grid #17 (404 points), Grid #22 (227 points), Grid #47 (3928 points), Grid #16 (5779 points) and Grid #21 (4341 points). Once the new insertions were made, a previously empty grid cell (Grid# 52) started filling up from the 37th insertion periodically. The

FIGURE 6.12: A1 - Yahoo! dataset: Maximum of top-N outliers post every insertion for A1 - Yahoo!.



new grid cell continued to be sparse till the entry of first seventeen points with each point behaving as kernel centers. Further insertion into the same cell transformed Grid# 52 into a relatively denser grid cell. However, the total number of points within Grid# 52 was still less than the other affected cell (Grid #47) during repeated insertions. As per KAGO algorithm, a running time of $\mathcal{O}(3r+3r)$ (Refer Section 6.8) is required for computing the *glos* value of individual points within a dense grid cell. Let $|g_{47}| = r1$ and $|g_{52}| = r2$, $\because r2 < r1$ across all insertions, we observe a periodic dip in the efficiency curve of KAGO (Figure 6.10) when Grid #52 is affected.

Key Result(s): Contrary to the previous datasets, we observed that the number of cells in *COG* (Figure 6.11) remained as three till the insertion of 36^{th} point. When the 37^{th} point was inserted, $|COG|$ increased to 4.

Reason(s): This may be possible when new grid cells are affected (previously empty) post insertion of data points. We observed that a new cell (Grid #52) was targeted when the 37^{th} point was entered. Upon observing the set of populated cells prior to any data insertion, we clearly noticed that Grid #52 was previously empty. This triggers a re-shuffle in the new *COG* list resulting in an addition of one or more grid cells as candidate outlier grids based on the updated *mglos* values.

Key Result(s): The set of top-N (N=122) outliers after the entry of first point experienced a steep decrease (Refer Figure 6.12) till the 890^{th} point (N=28) was inserted. During this phase, Grid #47 and #52 were repeatedly affected. From the insertion of 891^{st} point onward, no more reduction in number of outliers was observed.

Reason(s): This may happen when the *glos* values of existing points are at least as low as the value necessary for retaining the outlier status due to previous insertion and the new points are a part of the dense grid cells. Redundant positioning of new points in previously affected cells ensures a higher *glos* value for the contained points within the cell itself. This phenomenon ensures a decrease or constancy in the number of outliers.

TABLE 6.5: Key experimental results of the KAGO algorithm

Dataset	**#Grid cells**	**#Added points**	**Time KAGO (final point) (sec)**	**Time KNNOD (all points) (sec)**	**Speedup$_{max}$ (#insertion)**
NSL-KDD3	125	5000	0.06034	378.805	6660.12 (4000)
NSL-KDD4	625	5000	0.06064	390.89	6445.33 (5000)
A1-Yahoo!	625	3000	0.021	183.134	8403.72 (3000)

A brief summarization of experimental results comparing KAGO with KNNOD [5] on all the datasets have been shown in Table 6.5. Next we compare both the algorithms in terms of their memory consumption.

6.9.3 Memory usage

The high efficiency of KAGO over KNNOD [5] in terms of CPU execution time was achieved along with a significant reduction in memory consumption (Refer Table 6.6). For each dataset, we observed that the memory usage due to KAGO was approximately half as that of the KNNOD algorithm. On an average, the KAGO algorithm consumed about 51.57% less memory as compared to KNNOD, $\therefore |Mem_{point-ins} - Mem_{naive}| \approx .51$. Contrary to KAGO, a larger share of memory consumption due to KNNOD results from its storage of K-nearest neighbors for each data point in form of KNN matrix. The KAGO algorithm on the other hand allocates space only to the filled up grid cells within *GridStruct* facilitating the storage of *glos* value for each point. The grid cell indices and outliers occupy additional memory space.

6.9.4 Brief outlier analysis

For KNNOD, the size of K-Nearest Neighbor (KNN) and a cutoff threshold determine the outliers for a given dataset. In our experimental procedure, we initially

Table 6.6: Memory usage comparison between KAGO and KNNOD [5].

Dataset	KAGO memory usage (MB)	KNNOD memory usage (MB)	Gain factor
NSL-KDD3	9.88	19.74	**2.016**
NSL-KDD4	10.42	20.13	**1.93**
A1-Yahoo!	6.64	15.76	**2.37**

identified the distance $d_{K_{x_i}}$ (say) of each point $x_i \in D[1 \leq i \leq n]$ with its Kth nearest neighbor. The mean of all such distances is treated as the threshold $d_{K_{th}}$ (say). The value of K (K=$\lceil\sqrt{n}/10\rceil$) [93] was chosen to be 15,15 and 12 for NSL-KDD3, NSL-KDD4 and A1-Yahoo! datasets respectively. Any point whose $d_{K_{x_i}}$ value is greater than $d_{K_{th}}$, obtains an outlier status.

Table 6.7: Algorithm correctness evaluation.

Dataset	*#Classes*	KAGO		KNNOD	
		RI	F1-score	RI	F1-score
NSL-KDD3	22	**0.47563**	**0.64464**	0.38091	0.55168
NSL-KDD4	22	**0.40783**	**0.57938**	0.39762	0.55168
A1-Yahoo!	2	**1.0**	**1.0**	1.0	1.0

We evaluated the RI measure and F1-score for comparing the quality of results related to KAGO and KNNOD [5] (Refer Table 6.7). For NSL-KDD3 dataset, KAGO differs from KNNOD with approximately 9% more correct outliers' detection (RI). In case of NSL-KDD4, the percentage accuracy of both the algorithms are almost identical with around 1% difference in favor of KAGO. No change in accuracy was observed in case of A1-Yahoo! dataset. The reason for obtaining a better accuracy for the KAGO algorithm can be attributed to its repeated inlier pruning based on *glos* value instead of filtering outliers based on any predetermined threshold. The exactness wrt. both the algorithms have been compromised to a certain extent. However the usage of KAGO has ensured a more efficient and effective outlier detection scheme as compared to KNNOD (Refer Table 6.5, 6.7) $\therefore T_{point-ins}(Time_{KAGO}) < T_{naive}(Time_{KNNOD})$ and $O_{point-ins}(Outlier_{KAGO}) \approx O_{naive}(Outlier_{KNNOD})$.

6.10 Key properties of the KAGO algorithm

In this section, we dwell on the probable reasons behind certain assumptions made for our proposed algorithm. We also present few proofs related to some key possibilities in this work.

1. In the KAGO algorithm, any grid cell $g_c[1 \leq c \leq p^d]$ is considered as sparse ($VolGrid \leq 2^d + 1$) or relatively dense ($VolGrid > 2^d + 1$) depending on the number of contained points within it.
   Analysis/Reason(s): If the average number of points per grid ($|D|/\#Grids$) were taken instead of $VolGrid$, then there exists a possibility that a reasonably filled up subspace (grid cell) might be potentially inducted into COG in spite of being an inlier cell. In order to prevent this scenario, we incorporated the threshold $VolGrid$. Any $d-$dimensional grid cell contains 2^d corners and a center point. Since a combination of these points can exert distinct influence as kernel centers on any other point within g_c, we assumed $VolGrid$ to be the threshold between a sparse and a relatively denser grid cell. By no means we impose the fact that for any g_c ($|g_c| > VolGrid$), the cell is an absolute dense grid cell and hence we use term "relatively dense" wrt. our KAGO algorithm.

2. Three kernel centers are involved in a dense grid cell while all the data points within any sparse cell are treated as kernel centers.
   Reason(s): For any sparse grid cell $g_c[1 \leq c \leq p^d]$, we have $|g_c| \leq VolGrid$. As compared to the size of base dataset D, $|g_c| \ll |D|$. Therefore a selective choice of kernel centers from g_c may not represent the local data distribution within it. Moreover due to sparse nature of the concerned grid cell, the included points within g_c might be spatially dispersed. As result, $\forall x_i \in g_c\,(1 \leq i \leq r, r \leq VolGrid)$, x_i can potentially act as a kernel center having a distinct impact on KDE (local density) of any other point within g_c. Due to these reasons, each data point within a sparse grid cell are treated as kernel centers.

   In case of a dense grid cell ($|g_c| > VolGrid$), the points within g_c are very close to each other. Effectively, a similar influence of such close neighboring points on KDE of any $x_i \in g_c\,(1 \leq i \leq r, r > VolGrid)$ may lead to redundant computation. In order to efficiently compute the KDE $\forall x_i \in g_c$, we chose to represent the set of points in a dense grid cell with three kernel centers. These kernel enters include data points that are closest to minimal, maximal and centroid points of the cell (Refer Section 6.7).

3. A relatively denser grid cell with higher number of points may belong to the set COG.

   Analysis/Reason(s): The COG is identified based on increasing $mglos$ value of the grid cells. However after sorting all the non-empty cells, a certain grid cell g_c ($|g_c| > VolGrid$) may not obtain a sufficiently higher $mglos$ so as to be pruned like an inlier cell. As a result g_c continues to be a part of COG.

4. Selected grid cells in COG may not always produce the top-N global outliers after point insertions.

   Reason(s): This is an exceptional scenario which may arise if repeated insertions are made to an empty or single point grid cells. Under such a scenario, the dense grid cells will be pruned as inlier cells. However due to increase in the number of sparse grid cells, the list of cells in COG may not include such potential outlier cells. This may result in reduction of top-N global outliers initially. On the contrary, if we increase the percentage of grid cells in COG, then unnecessary computation may be involved in accessing the inlier points having a higher $glos$ value.

6.10.1 Lemmas related to the KAGO algorithm

Lemma 6.1. *Let* $q_{max} = max\{|COG|\} = \lceil \frac{\#Grids}{2} \rceil$ *produced from* D*. Let* q ($q < q_{max}$) *be the size of* COG *prior to any insertion. Then for each point insertion* $x_i, 1 \leq i \leq k$ *, we have* $0 \leq |COG| \leq q_{max}$.

Proof: With every insertion x_i, the $mglos$ value of the affected grid cell g_c ($1 \leq c \leq p^d$) gets updated. The new list of grid cells is generated to produce an updated COG. It is possible that the cells in old COG may become sufficiently dense to move out of new COG, but a previously inlier grid cell will not become a part of new $COG \implies |COG| = 0$.

If the old COG retains some of its grid cells after x_i's insertion, while some cells are removed due to increase in $mglos$ value, then $|COG| < q$. If no loss in new COG is observed due to redundant insertion on same grid cell(s), $|COG| = q, \therefore 0 \leq |COG| \leq q$. However, if the previously empty cells are affected across insertions, then at most all the grid cells may become non-empty. In that case $q = q_{max} \therefore 0 \leq |COG| \leq q_{max}$.

Lemma 6.2. *Degree of outlierness* $\forall x \in D \propto 1/glos(p)$.

Proof: From the definition of *glos* value for any $x \in D$ (Section 6.3), we have $glos(x) = f(z\text{-}score(x))$ [91]. $z\text{-}score(P, Q) = \frac{P-Q}{\sigma_Q}$, where P is the density of x wrt. any grid cell $g_c[1 \leq c \leq p^d]$ and Q is the mean local density of the included kernel centers within g_c. Therefore if $z\text{-}score(P, Q) \gg 0 \implies \frac{P-Q}{\sigma_Q} \gg 0 \implies P \gg Q$ $[\sigma_Q \neq 0]$, then a highly dense point x has a higher *glos* value. A low density point $(P \ll Q)$ has lower $glos(x)$ value and an higher probability of becoming an outlier. Effectively, grid cells containing points with lower $glos(x)$ values are more likely to become a part of COG.

6.11 Conclusion

In this contribution, we proposed an approximate point based incremental algorithm known as KAGO. Our proposed approach relies on local density derived through KDE instead of any distance measure while determining the local outlier status of a point. The local outliers obtained from different sub-spaces (grid cells) combine to produce at most top-N global outliers. In a dynamic setup, KAGO offers an efficient outlier detection scheme by selectively handling data points within the affected grid cell instead of entire data space. Experimental results on large network datasets and a search marketing advertiser bidding data showed the greater efficiency of KAGO over KNNOD, $\therefore T_{point-ins} < T_{naive}$. In addition, the KAGO algorithm consumed about half the memory as compared to KNNOD. We also showed that $O_{point-ins} \approx O_{naive}$ thereby proving the objectives as stated in Section 6.4.

Chapter 7

Conclusion and Future Scopes

Real time data analysis finds its importance in various modern day applications. Such applications may range from providing online recommendations to identifying threats in cyber security systems (Refer Section 1.2). A majority of these applications require extraction of patterns or meaningful information from the data with continuous updates. However, the task of extracting information from the ever changing data rests on certain dynamic algorithms specific to a given domain such as clustering, anomaly detection etc. These algorithms need to be curated in a definite manner such that a higher degree of efficiency is achieved along with an exact or near accurate output.

In this thesis, we particularly focused on the area of density based clustering and outlier detection. In our pursuit to produce necessary contributions, we developed four density based incremental mining algorithms. Three of these algorithms are related to clustering and one in the field outlier detection. Out of the three clustering algorithms, two of them provide an exact incremental solution. The remaining algorithm is an approximate incremental extension to the baseline method. The lone outlier detection algorithm leveraged the idea of KDE [29, 51] to find global outliers from a changing data.

While our first and the fourth contribution dealt with point-wise insertion of data [71], the second and third contribution involved both insertion and deletion of points in batch-mode.

7.1 Summary of the contributions

1. In our first contribution, we aimed at developing an approximate incremental version of the MBSCAN [2] clustering algorithm known as $iMass$. Intelligent construction of new mass-matrix, and an efficient design of the $iForest$ using prior node-split criterion enabled $iMass$ to achieve the much desired efficiency. The $iMass$ algorithm was about 191 times faster than MBSCAN (order of $\approx$ 2.28) for Iris dataset. Moreover by retaining the exactness of clusters for certain datasets and maintaining an overall accuracy of about 60.375% for unlabeled data, we showed the effectiveness of $iMass$ over MBSCAN. We further observed that the $iMass$ algorithm maintained a cluster accuracy percentage of about 80.4% for Libras dataset (class labeled) in terms of RI value. Also, for the Iris dataset (class labeled), an improved F1-score of 0.55848 was achieved.

2. For the second contribution, we proposed an exact incremental alternative to the SNN-DBSCAN (SNNDB) [24] algorithm. The proposed scheme supports batch-wise insertion of data points. InSDB [1], an existing incremental extension to SNNDB involves point based insertion. This makes the process extremely slow when updates are made to a larger base dataset. Moreover when the size of updates increases, InSDB fails to detect clusters efficiently as compared to SNNDB. This issue was resolved by our proposed $BISDB_{add}$ algorithm which proved to be the most efficient compared to its sub-variant methods: $Batch - Inc1$ and $Batch - Inc2$. The clusters obtained through $BISDB_{add}$ are identical to that of SNNDB. $BISDB_{add}$ outperformed SNNDB by upto an order of 3 ($\approx$ 1000 times) over three real world and two synthetic datasets. The mean memory overhead due to $BISDB_{add}$ was around 38.87% as compared SNNDB..

3. In our third contribution, the proposed algorithm ($BISDB_{del}$) incrementally extended SNNDB while removing data points. $BISDB_{del}$ proved to be the most efficient as compared to its sub-variant methods: $Batch - Dec1$ and $Batch - Dec2$. The clusters obtained through $BISDB_{del}$ are identical to that of the SNNDB algorithm. Comparisons with SNNDB revealed that the efficiency achieved by $BISDB_{del}$ reached upto an order of 4 ($\approx$ 10000 times) over three real world and two synthetic datasets. The average memory overhead due to $BISDB_{del}$ was around 41.08% as compared to SNNDB.

4. In our final contribution, we proposed a KDE based approach for detecting outliers from a grid partitioned data space. Against every insertion, the local outliers obtained from different sub-spaces combine to produce a set of at most top-N global outliers. The KAGO algorithm outperformed KNNOD by more than an order of 3.91 ($\approx$ 8304 times) (maximum) over two intrusion detection datasets and a bidding data for market advertisement related to a search engine. Outliers' evaluation on these datasets showed a mean improved accuracy of around 3.3% in case of KAGO. For each dataset, we observed that the memory usage due to KAGO is approximately half as that of the KNNOD algorithm. On an average, KAGO consumed about 51.57% less memory as compared to KNNOD.

7.2 Future scopes

The clustering techniques proposed in this thesis comply with the "Impossibility Theorem" of clustering [94]. This theorem suggests that clustering is an ill-posed problem. As a part of our future work, the various similarity measures adopted in the proposed algorithms may be readjusted in an attempt to disprove this theorem.

The incremental algorithms proposed in this thesis provide necessary insights for further work in related direction. The $iMass$ clustering algorithm only facilitates single point insertion of data. This work may be extended to support batch mode updates incrementally. Moreover, through either of the algorithms viz. $BISDB_{add}$ and $BISDB_{del}$, an exact incremental extension of SNNDB [24] was proposed. Knowing the importance of SNNDB in extracting clusters of arbitrary shapes and densities, it may be desirable to provide incremental extensions to more such robust clustering algorithms. We also observed that in Chapters 4, 5, the batch-incremental methods outperformed SNNDB and the point-based schemes after a certain percentage of updates. It will be interesting to see the maximum limit of batch size that can be allowed, beyond which we cannot sustain the efficiency gained due to $BISDB_{add}$ or $BISDB_{del}$.

The KAGO algorithm used the Gaussian kernel function for computing the local density of any point. It will be interesting to observe the influence of other kernels apart from the Gaussian kernel on the overall outliers' extraction process. As a part of future work, a comparison between KAGO and other state of the art outlier detection algorithms may be performed to further evaluate the efficiency.

Moreover, the algorithm may support batch mode updates involving both insertion and deletion of data.

Publications Related to Thesis

Journal (s):

- "*iMass*: An Approximate Adaptive Clustering Algorithm for Dynamic Data Using Probability Based Dissimilarity", **Panthadeep Bhattacharjee**, Pinaki Mitra. *Frontiers of Computer Science* (SCI-e), Springer/Higher Education Press, October 2020. ISSN 2095-2236 (Online). *DOI:*10.1007/s11704-019-9116-y, Volume 15 (2), Article number: 152314 (2021). URL: https://link.springer.com/article/10.1007/s11704-019-9116-y.
- "BISDBx: towards batch-incremental clustering for dynamic datasets using SNN-DBSCAN", **Panthadeep Bhattacharjee**, Pinaki Mitra. *Pattern Analysis and Applications* (SCI-e), Springer, May 2020. ISSN 1433-755X (Electronic), *DOI* : 10.1007/s10044-019-00831-1, Volume 23 (2), pages 975-1009. URL: link.springer.com/article/10.1007/s10044-019-00831-1.
- "A Survey of Density Based Clustering Algorithms", **Panthadeep Bhattacharjee**, Pinaki Mitra. *Frontiers of Computer Science* (SCI-e), Springer/Higher Education Press, September 2020. ISSN 2095-2236 (Online). *DOI:*10.1007/s11704-019-9059-3, Volume 15 (1), Article number: 151308 (2021). URL: https://link.springer.com/article/10.1007/s11704-019-9059-3.
- **Under Review**[†]**:**
  "KAGO: An Approximate Adaptive Grid Based Outlier Detection Approach using Kernel Density Estimate", **Panthadeep Bhattacharjee**, Ankur Garg, Pinaki Mitra. *Pattern Analysis and Applications* (SCI-e), Springer.

Conferences/Workshops/Forums:

- "Incremental Mining Algorithms: Adapting to Dynamic Data", **Panthadeep Bhattacharjee**, Pinaki Mitra. *24th International Conference on Advanced Computing and Communications* (ADCOM 2018), ACCS, PhD Forum, Bangalore, India, September 2018, ISBN 978-93-5321-421-0, pages 110-113. URL: https://accsindia.org/adcom-2018-conference-proceedings/.

[†]The manuscript was in minor revision stage at the time of revised thesis submission.

Bibliography

[1] S. Singh and A. Awekar, “Incremental shared nearest neighbor density-based clustering,” in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management.* ACM, 2013, pp. 1533–1536.

[2] K. M. Ting, Y. Zhu, M. Carman, Y. Zhu, and Z.-H. Zhou, “Overcoming key weaknesses of distance-based neighbourhood methods using a data dependent dissimilarity measure,” in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining.* ACM, 2016, pp. 1205–1214.

[3] A. Asuncion and D. Newman, “Uci machine learning repository,” 2007.

[4] P. Fränti and S. Sieranoja, “K-means properties on six clustering benchmark datasets,” pp. 4743–4759, 2018. [Online]. Available: http://cs.uef.fi/sipu/datasets/

[5] T. T. Dang, H. Y. Ngan, and W. Liu, “Distance-based k-nearest neighbors outlier detection method in large-scale traffic data,” in *2015 IEEE International Conference on Digital Signal Processing (DSP).* IEEE, 2015, pp. 507–510.

[6] D. T. Pham, S. S. Dimov, and C. Nguyen, “An incremental k-means algorithm,” *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, vol. 218, no. 7, pp. 783–795, 2004.

[7] K. M. Hammouda and M. S. Kamel, “Incremental document clustering using cluster similarity histograms,” in *Proceedings IEEE/WIC International Conference on Web Intelligence (WI 2003)*, Oct 2003, pp. 597–601.

[8] M. Charikar, C. Chekuri, T. Feder, and R. Motwani, “Incremental clustering and dynamic information retrieval,” *SIAM Journal on Computing*, vol. 33, no. 6, pp. 1417–1440, 2004.

[9] H. Chang, J. Lin, M. Cheng, and S. Huang, "A novel incremental data mining algorithm based on fp-growth for big data," in *2016 International Conference on Networking and Network Applications (NaNA)*, July 2016, pp. 375–378.

[10] M.-Y. Su, G.-J. Yu, and C.-Y. Lin, "A real-time network intrusion detection system for large-scale attacks based on an incremental mining approach," *Computers & security*, vol. 28, no. 5, pp. 301–309, 2009.

[11] R. Baldoni, L. Montanari, and M. Rizzuto, "On-line failure prediction in safety-critical systems," *Future Generation Computer Systems*, vol. 45, pp. 123–132, 2015.

[12] Wikipedia, *Dynamic data*. [Online]. Available: http://https://en.wikipedia.org/wiki/Dynamic_data

[13] J. Basilico and T. Hofmann, "Unifying collaborative and content-based filtering," in *Proceedings of the twenty-first international conference on Machine learning*. ACM, 2004, p. 9.

[14] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon, "I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '07. New York, NY, USA: ACM, 2007, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/1298306.1298309

[15] G. Linden, B. Smith, and J. York, "Amazon. com recommendations: Item-to-item collaborative filtering," *IEEE Internet computing*, no. 1, pp. 76–80, 2003.

[16] T. Anwar, C. Liu, H. L. Vu, and M. S. Islam, "Roadrank: Traffic diffusion and influence estimation in dynamic urban road networks," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '15. New York, NY, USA: ACM, 2015, pp. 1671–1674. [Online]. Available: http://doi.acm.org/10.1145/2806416.2806588

[17] S. Tuarob, C. S. Tucker, M. Salathe, and N. Ram, "Modeling individual-level infection dynamics using social network information," in *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management*, ser. CIKM '15. New York, NY, USA: ACM, 2015, pp. 1501–1510. [Online]. Available: http://doi.acm.org/10.1145/2806416.2806575

[18] S. Mathe and C. Sminchisescu, "Dynamic eye movement datasets and learnt saliency models for visual action recognition," in *European Conference on Computer Vision*. Springer, 2012, pp. 842–856.

[19] J. Green and J. Schultz, "Dynamic-content web crawling through traffic monitoring," Nov. 28 2006, uS Patent 7,143,088.

[20] G. Folino, F. S. Pisani, and P. Sabatino, "An incremental ensemble evolved by using genetic programming to efficiently detect drifts in cyber security datasets," in *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '16 Companion. New York, NY, USA: ACM, 2016, pp. 1103–1110. [Online]. Available: http://doi.acm.org/10.1145/2908961.2931682

[21] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM computing surveys (CSUR)*, vol. 31, no. 3, pp. 264–323, 1999.

[22] R. Xu and D. C. Wunsch, "Survey of clustering algorithms," 2005.

[23] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *Kdd*, vol. 96, no. 34, 1996, pp. 226–231.

[24] L. Ertöz, M. Steinbach, and V. Kumar, "Finding clusters of different sizes, shapes, and densities in noisy, high dimensional data," in *Proceedings of the 2003 SIAM international conference on data mining*. SIAM, 2003, pp. 47–58.

[25] C. Sammut and G. I. Webb, *Encyclopedia of machine learning and data mining*. Springer Publishing Company, Incorporated, 2017.

[26] R. T. Ng and J. Han, "Clarans: A method for clustering objects for spatial data mining," *IEEE Transactions on Knowledge & Data Engineering*, no. 5, pp. 1003–1016, 2002.

[27] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[28] D. Müllner, "Modern hierarchical, agglomerative clustering algorithms," *arXiv preprint arXiv:1109.2378*, 2011.

[29] B. W. Silverman, *Density estimation for statistics and data analysis*. Routledge, 2018.

[30] G. Gan, C. Ma, and J. Wu, *Data clustering: theory, algorithms, and applications*. Siam, 2007, vol. 20.

[31] C. Domeniconi, D. Papadopoulos, D. Gunopulos, and S. Ma, "Subspace clustering of high dimensional data," in *Proceedings of the 2004 SIAM international conference on data mining*. SIAM, 2004, pp. 517–521.

[32] J. Sander, M. Ester, H.-P. Kriegel, and X. Xu, "Density-based clustering in spatial databases: The algorithm gdbscan and its applications," *Data mining and knowledge discovery*, vol. 2, no. 2, pp. 169–194, 1998.

[33] G. D. Bader and C. W. Hogue, "An automated method for finding molecular complexes in large protein interaction networks," *BMC bioinformatics*, vol. 4, no. 1, p. 2, 2003.

[34] N. R. Chrisman, J. A. Dougenik, and D. White, "Lessons for the design of polygon overlay processing from the odyssey whirlpool algorithm," in *Proc. 5th Int. Symp. on Spatial Data Handling*, vol. 2, 1992, pp. 401–410.

[35] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.

[36] Y. Liao and V. R. Vemuri, "Use of k-nearest neighbor classifier for intrusion detection," *Computers & security*, vol. 21, no. 5, pp. 439–448, 2002.

[37] M. M. Breunig, H.-P. Kriegel, R. T. Ng, and J. Sander, "Lof: identifying density-based local outliers," in *ACM sigmod record*, vol. 29, no. 2. ACM, 2000, pp. 93–104.

[38] Y. Li, B. Fang, L. Guo, and Y. Chen, "Network anomaly detection based on tcm-knn algorithm," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. ACM, 2007, pp. 13–19.

[39] A. Srivastava, A. Kundu, S. Sural, and A. Majumdar, "Credit card fraud detection using hidden markov model," *IEEE Transactions on dependable and secure computing*, vol. 5, no. 1, pp. 37–48, 2008.

[40] A. Brabazon, J. Cahill, P. Keenan, and D. Walsh, "Identifying online credit card fraud using artificial immune systems," in *IEEE Congress on Evolutionary Computation*. IEEE, 2010, pp. 1–7.

[41] S. Haque, M. Rahman, and S. Aziz, "Sensor anomaly detection in wireless sensor networks for healthcare," *Sensors*, vol. 15, no. 4, pp. 8764–8786, 2015.

[42] R. Mitchell and R. Chen, "Behavior-rule based intrusion detection systems for safety critical smart grid applications," *IEEE Transactions on Smart Grid*, vol. 4, no. 3, pp. 1254–1263, 2013.

[43] M. Ester, H.-P. Kriegel, J. Sander, M. Wimmer, and X. Xu, "Incremental clustering for mining in a data ware housing," *University of Munich Oettingenstr*, vol. 67.

[44] N. Chen, A.-z. Chen, and L.-x. Zhou, "An incremental grid density-based clustering algorithm," *Journal of software*, vol. 13, no. 1, pp. 1–7, 2002.

[45] J. Gan and Y. Tao, "Dynamic density based clustering," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 1493–1507.

[46] X. Xu, M. Ester, H.-P. Kriegel, and J. Sander, "A distribution-based clustering algorithm for mining in large spatial databases," in *Proceedings 14th International Conference on Data Engineering*. IEEE, 1998, pp. 324–331.

[47] D. Pokrajac, A. Lazarevic, and L. J. Latecki, "Incremental local outlier detection for data streams," in *2007 IEEE Symposium on Computational Intelligence and Data Mining*, March 2007, pp. 504–515.

[48] S. H. Karimian, M. Kelarestaghi, and S. Hashemi, "I-inclof: Improved incremental local outlier detection for data streams," in *The 16th CSI International Symposium on Artificial Intelligence and Signal Processing (AISP 2012)*, May 2012, pp. 023–028.

[49] M. Salehi, C. Leckie, J. C. Bezdek, T. Vaithianathan, and X. Zhang, "Fast memory efficient local outlier detection in data streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 12, pp. 3246–3260, Dec 2016.

[50] G. S. Na, D. Kim, and H. Yu, "Dilof: Effective and memory efficient local outlier detection in data streams," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery &#38; Data Mining*, ser. KDD '18. New York, NY, USA: ACM, 2018, pp. 1993–2002. [Online]. Available: http://doi.acm.org/10.1145/3219819.3220022

[51] X. Qin, L. Cao, E. A. Rundensteiner, and S. Madden, "Scalable kernel density estimation-based local outlier detection over large data streams." in *EDBT*, 2019, pp. 421–432.

[52] R. A. Jarvis and E. A. Patrick, “Clustering using a similarity measure based on shared near neighbors,” *IEEE Transactions on computers*, vol. 100, no. 11, pp. 1025–1034, 1973.

[53] B. G. Mirkin and A. V. Kramarenko, “Approximate bicluster and tricluster boxes in the analysis of binary data,” in *Rough Sets, Fuzzy Sets, Data Mining and Granular Computing*, S. O. Kuznetsov, D. Slezak, D. H. Hepting, and B. G. Mirkin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 248–256.

[54] R. C. de Amorim, V. Makarenkov, and B. Mirkin, “Core clustering as a tool for tackling noise in cluster labels,” *Journal of Classification*, pp. 1–15, 2019.

[55] D. I. Ignatov, S. O. Kuznetsov, J. Poelmans, and L. E. Zhukov, “Can triconcepts become triclusters?” *International Journal of General Systems*, vol. 42, no. 6, pp. 572–593, Aug. 2013.

[56] D. I. Ignatov, D. V. Gnatyshak, S. O. Kuznetsov, and B. G. Mirkin, “Triadic formal concept analysis and triclustering: searching for optimal patterns,” *Machine Learning*, vol. 101, no. 1-3, pp. 271–302, 2015.

[57] B. Mirkin, “Summary and semi-average similarity criteria for individual clusters,” in *Models, Algorithms, and Technologies for Network Analysis*. Springer, 2013, pp. 101–126.

[58] D. Gnatyshak, D. I. Ignatov, S. O. Kuznetsov, and L. Nourine, “A one-pass triclustering approach: Is there any room for big data?” in *CLA*, 2014, pp. 231–242.

[59] S. O. Kuznetsov and S. A. Obiedkov, “Comparing performance of algorithms for generating concept lattices,” *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 14, no. 2-3, pp. 189–216, 2002.

[60] D. G. Kourie, S. Obiedkov, B. W. Watson, and D. van der Merwe, “An incremental algorithm to construct a lattice of set intersections,” *Science of Computer Programming*, vol. 74, no. 3, pp. 128–142, 2009.

[61] R. Wille, “Restructuring lattice theory: an approach based on hierarchies of concepts,” in *International Conference on Formal Concept Analysis*. Springer, 2009, pp. 314–339.

[62] C. Carpineto and G. Romano, *Concept data analysis: Theory and applications*. John Wiley & Sons, 2004.

[63] P. Lingras and G. Peters, "Rough clustering," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 1, no. 1, pp. 64–72, 2011.

[64] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 604–613.

[65] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 2008, pp. 413–422.

[66] C. L. Krumhansl, "Concerning the applicability of geometric models to similarity data: The interrelationship between similarity and spatial density." 1978.

[67] A. Tversky, "Features of similarity." *Psychological review*, vol. 84, no. 4, p. 327, 1977.

[68] S. Aryal, K. M. Ting, G. Haffari, and T. Washio, "mp-dissimilarity: A data dependent dissimilarity measure," in *Data Mining (ICDM), 2014 IEEE International Conference on*. IEEE, 2014, pp. 707–712.

[69] H. Shojanazeri, D. Zhang, S. W. Teng, S. Aryal, and G. Lu, "A novel perceptual dissimilarity measure for image retrieval," in *2018 International Conference on Image and Vision Computing New Zealand (IVCNZ)*. IEEE, 2018, pp. 1–6.

[70] K. M. Ting, G.-T. Zhou, F. T. Liu, and S. C. Tan, "Mass estimation," *Machine Learning*, vol. 90, no. 1, pp. 127–160, Jan 2013. [Online]. Available: https://doi.org/10.1007/s10994-012-5303-x

[71] S. Bandyopadhyay and M. N. Murty, "Axioms to characterize efficient incremental clustering," in *2016 23rd International Conference on Pattern Recognition (ICPR)*, 2016, pp. 450–455.

[72] X. He, D. Cai, and P. Niyogi, "Laplacian score for feature selection," in *Advances in neural information processing systems*, 2006, pp. 507–514.

[73] Z. Li, J. Tang, and X. He, "Robust structured nonnegative matrix factorization for image representation," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 5, pp. 1947–1960, May 2018.

[74] D. Kong, C. Ding, and H. Huang, "Robust nonnegative matrix factorization using l21-norm," in *Proceedings of the 20th ACM International*

Conference on Information and Knowledge Management, ser. CIKM '11. New York, NY, USA: ACM, 2011, pp. 673–682. [Online]. Available: http://doi.acm.org/10.1145/2063576.2063676

[75] Z. Li and J. Tang, "Unsupervised feature selection via nonnegative spectral analysis and redundancy control," *IEEE Transactions on Image Processing*, vol. 24, no. 12, pp. 5343–5355, 2015.

[76] L. E. Peterson, "K-nearest neighbor," *Scholarpedia*, vol. 4, no. 2, p. 1883, 2009.

[77] M. Thottan and C. Ji, "Anomaly detection in ip networks," *IEEE Transactions on signal processing*, vol. 51, no. 8, pp. 2191–2204, 2003.

[78] M. Xie, J. Hu, S. Han, and H.-H. Chen, "Scalable hypergrid k-nn-based online anomaly detection in wireless sensor networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 8, pp. 1661–1670, 2012.

[79] O. Salem, Y. Liu, A. Mehaoua, and R. Boutaba, "Online anomaly detection in wireless body area networks for reliable healthcare monitoring," *IEEE journal of biomedical and health informatics*, vol. 18, no. 5, pp. 1541–1551, 2014.

[80] C. C. Aggarwal, "Outlier analysis," in *Data mining*. Springer, 2015, pp. 237–263.

[81] L. J. Latecki, A. Lazarevic, and D. Pokrajac, "Outlier detection with kernel density functions," in *International Workshop on Machine Learning and Data Mining in Pattern Recognition*. Springer, 2007, pp. 61–75.

[82] E. Schubert, A. Zimek, and H.-P. Kriegel, "Generalized outlier detection with flexible kernel density estimates," in *Proceedings of the 2014 SIAM International Conference on Data Mining*. SIAM, 2014, pp. 542–550.

[83] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos, "Online outlier detection in sensor data using non-parametric models," in *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 187–198.

[84] M. Kirchner, "A framework for detecting anomalies in http traffic using instance-based learning and k-nearest neighbor classification," in *2010 2nd International Workshop on Security and Communication Networks (IWSCN)*, May 2010, pp. 1–8.

[85] A. O. Hero, "Geometric entropy minimization (gem) for anomaly detection and localization," in *Advances in Neural Information Processing Systems*, 2007, pp. 585–592.

[86] E. Khalastchi, G. A. Kaminka, M. Kalech, and R. Lin, "Online anomaly detection in unmanned vehicles," in *The 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 1*. International Foundation for Autonomous Agents and Multiagent Systems, 2011, pp. 115–122.

[87] E. M. Knorr and R. T. Ng, "Algorithms for mining distance-based outliers in large datasets," in *VLDB*, vol. 98. Citeseer, 1998, pp. 392–403.

[88] S. Ramaswamy, R. Rastogi, and K. Shim, "Efficient algorithms for mining outliers from large data sets," in *ACM Sigmod Record*, vol. 29, no. 2. ACM, 2000, pp. 427–438.

[89] L. Cao, D. Yang, Q. Wang, Y. Yu, J. Wang, and E. A. Rundensteiner, "Scalable distance-based outlier detection over high-volume data streams," in *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 2014, pp. 76–87.

[90] G. Qian, S. Sural, Y. Gu, and S. Pramanik, "Similarity between euclidean and cosine angle distance for nearest neighbor queries," in *Proceedings of the 2004 ACM Symposium on Applied Computing*, ser. SAC '04. New York, NY, USA: ACM, 2004, pp. 1232–1237. [Online]. Available: http://doi.acm.org/10.1145/967900.968151

[91] D. Zill, W. S. Wright, and M. R. Cullen, *Advanced engineering mathematics*. Jones & Bartlett Learning, 2011.

[92] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[93] A. B. Hassanat, M. A. Abbadi, G. A. Altarawneh, and A. A. Alhasanat, "Solving the problem of the k parameter in the knn classifier using an ensemble learning approach," *arXiv preprint arXiv:1409.0919*, 2014.

[94] J. Kleinberg, "An impossibility theorem for clustering," *Advances in neural information processing systems*, vol. 15, pp. 463–470, 2002.