Complexity Results in Some Clustering Algorithms

A dissertation submitted in
partial fulfilment for the degree of
Master of Technology
in
Computer Science

by
Rajdeep Das
Roll No. **CS2315**

Under the supervision of
Proff. Sandip Das
Advanced Computing and Microelectronics Unit (ACMU)

Indian Statistical Institute, Kolkata

June, 2025

CERTIFICATE

This is to certify that the dissertation entitled **"Complexity Results in Some Clustering Algorithms"** submitted by **Rajdeep Das** to the **Indian Statistical Institute, Kolkata**, in partial fulfillment of the requirements for the degree of **Master of Technology in Computer Science**, is an authentic and genuine record of the research work carried out by the candidate under my supervision and guidance.

I affirm that the dissertation has met all the necessary requirements in accordance with the regulations of this institute.

Proff. Sandip Das
ACMU Unit
Indian Statistical Institute
Kolkata – 700108
India

Acknowledgement

I would like to express my heartfelt gratitude to **Proff. Sandip Das**, my advisor at the Advanced Computing and Microelectronics Unit of the Indian Statistical Institute, Kolkata, for his invaluable guidance, unwavering support, and continued encouragement throughout the course of this research. His deep expertise and insightful suggestions have played a pivotal role in shaping both the direction and the depth of this dissertation.

I am also sincerely thankful to **Sangita Saha**, Senior Research Fellow at the Indian Statistical Institute, for her generous assistance in providing ideas crucial to this work. Her steady flow of ideas and constant support has been instrumental in driving this project forward.

I extend my appreciation to all the faculty members of the Indian Statistical Institute for their exceptional instruction, thoughtful feedback, and academic mentorship, all of which contributed meaningfully to my learning journey.

Finally, I owe a deep debt of gratitude to my parents and extended family for their endless support and motivation. I am also thankful to all my friends, whose companionship and encouragement have been a source of strength throughout this academic pursuit. I extend my sincere thanks to everyone who has helped me, directly or indirectly, even if I have not mentioned them by name.

Declaration

I, **Rajdeep Das**, bearing Roll No. **CS2315**, hereby declare that the material presented in the dissertation titled **"Complexity Results in Some Clustering Algorithms"** represents original work carried out by me towards the partial fulfillment of the degree of **Master of Technology in Computer Science** at the **Indian Statistical Institute**, Kolkata.

I further affirm that no part of this dissertation has been copied or reproduced from external sources without proper citation. I am fully aware that any form of plagiarism or unacknowledged use of third-party content will be treated as a serious academic offense in accordance with institutional policies.

Rajdeep Das
M.Tech (CS), Roll No. CS2315
Indian Statistical Institute, Kolkata

Abstract

Density-Based Spatial Clustering of Applications with Noise (DBSCAN) is a prevalent Clustering method without supervision renowned for its capability to recognize arbitrarily shaped clusters and detect noise in spatial data. Unlike partitioning methods such as k-means, DBSCAN operates without inputting a predefined number of clusters and is particularly effective in handling datasets with varying densities.

In this dissertation, we have undertaken an in-depth exploration of the DBSCAN algorithm. We reviewed and analyzed several research papers that build upon or revise the original DBSCAN framework, with the goal of understanding their motivations, design choices, and computational implications.

In addition to studying the foundational principles, we examined traditional spatial data structures that are commonly employed to accelerate DBSCAN, such as R-trees and KD-trees. This background enabled us to identify key computational bottlenecks in both neighbor search and density estimation.

Building on these insights, we proposed two novel algorithms. The first is an approximate algorithm that efficiently replicates standard DBSCAN behavior, and the second is a modified version termed *Box-based DBSCAN*, which operates under a slightly altered definition of neighborhood using axis-aligned bounding boxes. The box-based approach improves clustering performance for geometrically structured data and introduces new ways to identify core regions without relying on exhaustive point-wise comparisons.

Keywords: DBSCAN (Density-Based Spatial Clustering of Applications with Noise), BB (Bounding Box), MBB (Minimum Bounding Box), USEC (Unit-Spherical Emptiness Checking), BCP (Bi-Chromatic Closest Pair)

Contents

Acknowledgement **1**

Abstract **1**

1 Introduction **5**

1.1 Background 5

1.2 DBSCAN Problem Statement 5

1.3 Thesis Organization 7

2 Literature Survey **8**

2.1 Literature Review 8

2.2 Gap Analysis 13

3 Methodology **15**

3.1 Approach 15

3.1.1 Step-by-Step Description of the Algorithm 16

4 Implementation **34**

4.1 C++ Implementation 34

5 Results and Discussion **41**

5.1 Results 41

6 Conclusion and Future Work **47**

6.1 Conclusion 47

6.2 Limitations 48

6.3 Future Scope 48

List of Figures

1.1 Illustration of core and non-core points as defined in Definition 1. Black points are core points (each has MinPts points atleast in its ϵ-radius ball), and white points with black edges are non-core points. Gray circles denote ϵ-neighborhoods. 6

1.2 Connectivity in DBSCAN through a chain of border points. Two dense clusters (Cluster A and Cluster B) are connected via a sequence of intermediate border points. Each arrow indicates a link within ϵ distance, forming a density-connected path as per DBSCAN's definition. 6

3.1 Projection of clustered points onto the X-axis. The dotted lines represent projections from each 2D point onto its corresponding location on the X-axis. 16

3.2 Grouping by range in the X-dimension. Each group is formed by scanning the sorted list of points along the X-axis and collecting those within a fixed-width interval. Vertical dashed lines mark group boundaries and are color-matched to the group points. 17

3.3 Recursive grouping with matching bounding lines. Points are first grouped along the X-axis, and then each group is recursively subdivided along the Y-axis. 17

3.4 Final MBBs obtained through recursive grouping in X and Y dimensions. Each point group is enclosed within a minimum bounding box (MBB) formed by applying fixed-width range grouping first along the X-axis and then along the Y-axis. The bounding lines are color-matched to their respective point groups. 18

3.5 Approximate Bi-Chromatic Closest Pair (BCP) Test: Core points from two MBBs are projected onto the centroid-to-centroid line. Top-k projected points from each MBB are marked, and only these pairs are considered for ε-distance testing. Dotted rectangles represent the Minimum Bounding Boxes (MBBs) of each core set. 26

5.1 Standard DBSCAN on the Jigsaw dataset. 42

5.2 Box-based DBSCAN on Jigsaw dataset. 42

5.3 Comparison of clustering performance on the Jigsaw dataset. 42

5.4 Standard DBSCAN - Spiral 43

5.5 Box-based DBSCAN - Spiral 43
5.6 Comparison of clustering performance on the Spiral dataset. 43
5.7 Standard DBSCAN - Concentric Circles 44
5.8 Box-based DBSCAN - Concentric Circles 44
5.9 Comparison of clustering performance on the Concentric Circles dataset. 44
5.10 Standard DBSCAN - Two Moons 45
5.11 Box-based DBSCAN - Two Moons 45
5.12 Comparison of clustering performance on the Two moon dataset. 45

Chapter 1

Introduction

1.1 Background

Clustering is a fundamental method used in data mining and machine learning , used to partition data points according to some similarity or proximity metrics. Among the clustering algorithms, DBSCAN, introduced by Ester et al. in 1996, is widely used for its capability to discover clusters of even non-convex shapes and to effectively handle noise in spatial data. Unlike partitioning methods such as k-means, DBSCAN does not require the knowledge of the number of clusters beforehand and is capable of identifying clusters of varying sizes and densities.

DBSCAN classifies points as core points and non-core points, based on two critical parameters: *epsilon* (ε), which specifies the neighborhood radius, and MinPts, the minimum number of points needed to qualify as a dense region. By exploring the density connectivity of points, DBSCAN forms clusters by expanding from core points, connecting them to other reachable core or border points. Its simplicity and effectiveness have made it a foundational method in numerous applications, including geospatial data analysis, image segmentation, and anomaly detection.

1.2 DBSCAN Problem Statement

Let there be a set P containing n number of points within a space having dimension d ($\mathbb{R}^d$). For any two points $p, q \in \mathbb{R}^d$, we represent the Euclidean distance among these two points as $\text{dist}(p, q)$. We define $B(p, r)$ as the open ball with center $p \in \mathbb{R}^d$ and radius r. Now for DBSCAN algorithm having two parameters: ϵ and MinPts.

Definition 1. Any $p \in P$ is called a *core point* if the ball $B(p, \epsilon)$ contains at least MinPts number of points from P (point p itself included). p is referred to as a *non-core point* if it i

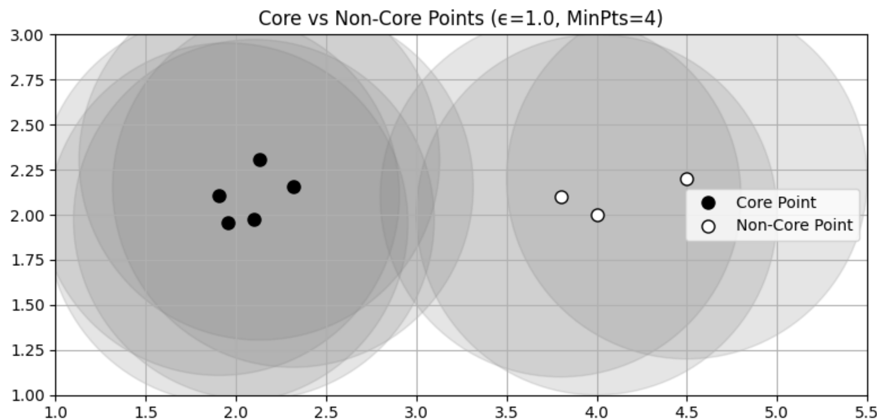

Figure 1.1: Illustration of core and non-core points as defined in Definition 1. Black points are core points (each has MinPts points atleast in its ϵ-radius ball), and white points with black edges are non-core points. Gray circles denote ϵ-neighborhoods.

Definition 2. A point $q \in P$ is said to be *density-reachable* from a point $p \in P$ if there is a sequence of points $p_1, p_2, \ldots, p_t \in P$ (for some integer $t \geq 2$) such that:

- $p_1 = p$ and also $p_t = q$,
- Each of the point belongs to $p_1, p_2, \ldots, p_{t-1}$ is a core point,
- $p_{i+1} \in B(p_i, \epsilon)$ for every $i \in [1, t-1]$,

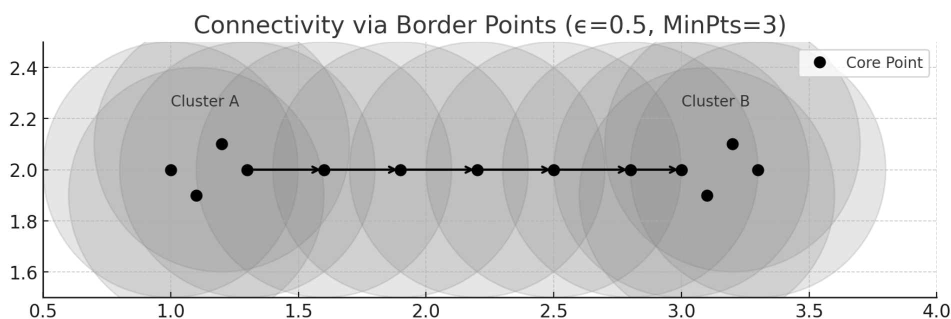

Figure 1.2: Connectivity in DBSCAN through a chain of border points. Two dense clusters (Cluster A and Cluster B) are connected via a sequence of intermediate border points. Each arrow indicates a link within ϵ distance, forming a density-connected path as per DBSCAN's definition.

Definition 3. A cluster $C \subseteq P$, with $C \neq \emptyset$, is defined as a non-empty subset of points that satisfies the following properties:

- **Minimality:** If $p \in C$ is a core point, then every point that is density-reachable from p must also be contained in C.
- **Connectivity:** For any two points $p_1, p_2 \in C$, there exists a point $p \in C$ such that both p_1 and p_2 are density-reachable from p.

The DBSCAN problem is to identify the unique set $\mathcal{C}$ of clusters over the point set P.

1.3 Thesis Organization

- **Chapter 1: Introduction** – Provides the background, motivation, problem statement, objectives, and the structure of the thesis.
- **Chapter 2: Literature Review** – Discusses existing research and developments related to DBSCAN and its variants.
- **Chapter 3: Methodology** – Describes the algorithm, implementation details.
- **Chapter 4: Results and Discussion** – Presents experimental results, and comparative analysis.
- **Chapter 5: Conclusion and Future Work** – Key findings summarized, limitations, and suggested directions for future .

Chapter 2

Literature Survey

2.1 Literature Review

The DBSCAN algorithm was first introduced in the seminal KDD 1996 paper titled *"A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise"*. In this work, the authors formally defined key concepts such as *core points*, *border points*, and *noise*, establishing a new density-based approach to clustering. They use the help of R* Tree (A Spatial query data structure) for query purpose. It was suggested, without formal justification, that DBSCAN achieves an average-case time complexity of $O(n \log n)$, because querying each point on average case takes $O(logn)$ time complexity. However, the paper did not delve into the worst-case time complexity or its behavior in high-dimensional spaces.

we also review the existing literature relevant to DBSCAN , with a special focus on the SIGMOD 2015 paper titled *"DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation"* by Junhao Gan and Yufei Tao. This paper revisits the theoretical and practical efficiency of the DBSCAN algorithm, identifies major issues in the original formulation, and proposes an efficient approximation method. The key points from the literature and the methods used in this review are discussed below.

- **Misclaimed Time Complexity of Original DBSCAN:**
  The original DBSCAN algorithm introduced by Ester et al. in KDD 1996 claimed to operate in $O(n \log n)$ time. However, Gan and Tao [SIGMOD 2015] demonstrate that this claim is incorrect; the actual worst-case time complexity is $O(n^2)$ due to the need for n range queries, each potentially costing $O(n)$ in dense datasets.

- **Review of Propagation of the Error:**
  The incorrect $O(n \log n)$ claim has been widely propagated through:
  - Wikipedia and multiple textbooks [e.g., 12, 20, 25].
  - Numerous academic papers [e.g., 2, 5, 7, 13, 16, 19, 24, 28, 29, 30].
  - Derived results in later works [e.g., 14, 21, 23] that were built upon this incorrect assumption.

  This propagation highlights a critical issue in algorithmic research where foundational claims, if unchecked, can lead to systemic flaws in subsequent studies.

- **Review of Gunawan's 2D Algorithm (2013):**
  Gunawan [2013] addressed the flaw in the original paper and provided a true $O(n \log n)$-time DBSCAN algorithm for two-dimensional space. This method uses:
  - A grid overlay with cells of size $\frac{\varepsilon}{\sqrt{2}}$.
  - Labeling of core and border points based on cell density.
  - Construction of a graph data structure where each and every vertex corresponds to a dense cell (cell that has atleat one core point), and edges represent spatial proximity.
  - Connected component detection to form clusters.

- **Hardness Results from Computational Geometry:**
  To understand the intractability of DBSCAN in dimensions $d \geq 3$, the authors relate the problem to known computational geometry problems:
  - *Bichromatic Closest Pair (BCP):* Best known algorithm has subquadratic time only for $d = 3$ or less.
  - *Unit Spherical Emptiness Checking (USEC):* Requires $\Omega(n^{4/3})$ time in 3D, believed to be optimal.
  - *Hopcroft's Problem:* Also exhibits $\Omega(n^{4/3})$ lower bounds in 2D, implying hardness in higher dimensions.

  A reduction of DBSCAN to USEC shows that an efficient exact DBSCAN algorithm for $d \geq 3$ would lead to breakthroughs in longstanding open problems, which are believed to be impossible to solve.

- **Introduction of ρ-Approximate DBSCAN:**
  The authors proposed the concept of ρ-approximate DBSCAN , To address the computational limitations :

  - Allows for slight inaccuracies, enabling expected linear-time performance ($O(n)$).
  - Offers a quality guarantee: the result is "sandwiched" between the clustering obtained by running DBSCAN with parameters ε and $\varepsilon(1+\rho)$.
  - Suitable for practical clustering of large high-dimensional datasets.

- **Hierarchical Grid-based Approximate Range Counting:**
  For efficient implementation, the paper proposes a grid-based structure that supports:
  - Fast approximate range count queries in $O(1)$ expected time.
  - Space and build time of $O(n)$.
  - Use of multilevel hierarchical grids with hashing to ensure efficiency.

- **Experimental Validation:**
  Extensive experiments were conducted on both synthetic and real datasets. Findings include:
  - Exact DBSCAN variants do not scale well in $d \geq 3$, often failing on datasets with millions of points.
  - ρ-approximate DBSCAN achieves performance improvements of up to three orders of magnitude.
  - Even with $\rho = 0.001$, the approximation returns results indistinguishable from exact DBSCAN in most practical settings.

- **Conclusion of Literature Findings:**
  This paper thoroughly investigates the theoretical and empirical inefficiencies of DBSCAN in higher dimensions. By leveraging advances in computational geometry and introducing principled approximations, it lays the foundation for a practical alternative that preserves clustering quality while achieving scalable performance.

And lastly we review the paper titled *"DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN"* by Eldar Khalil and Thomas Heinis, published in VLDB 2022. This work responds to the criticisms raised in the SIGMOD 2015 paper by Gan and Tao, which questioned the practical feasibility of DBSCAN in high-dimensional spaces. Rather than dismissing DBSCAN based solely on theoretical hardness results, Khalil and Heinis reassess the algorithm's performance in light of modern hardware and engineering optimizations. Their review methodology is grounded in empirical validation, algorithmic synthesis, and practical system design.

- **Revisiting Theoretical Complexity Claims:**
  The paper acknowledges the lower bounds established by Gan and Tao, particularly the result that DBSCAN requires at least $\Omega(n^{4/3})$ time in three or more dimensions due to reductions from the Unit Spherical Emptiness Check (USEC) problem. However, the authors emphasize that these worst-case scenarios do not reflect practical datasets, which are often low-dimensional in structure or sparse in distribution.

- **Criticism of row-Approximate (Row-Approximate) DBSCAN:**
  A core contribution of Gan and Tao was the introduction of row-approximate DBSCAN, which provides a sandwich guarantee between clusters formed at radii ε and $\varepsilon(1+\rho)$. However, Khalil and Heinis critique this approach on multiple grounds:

  - The dimensional complexity of their approximate range query method is still exponential in d, specifically $O((1 + 1/\rho)^d)$, making it unscalable for high dimensions.
  - The algorithm is largely impractical in modern settings, as it does not leverage hardware acceleration, indexing strategies, or multithreading.
  - Real-world clustering tasks typically tolerate minor boundary variations, making such strong approximation guarantees unnecessary in practice.

  The authors thus advocate for pragmatic heuristics and data-aware approximations rather than rigid theoretical bounds.

- **Survey of Optimized DBSCAN Variants:**
  The paper consolidates a number of previous approaches, including:

  - *Grid-based clustering algorithms*, such as those from Gunawan (2013), which partition space into discrete cells.
  - *Index-accelerated DBSCAN variants* using R-trees, KD-trees, and VA-files.
  - *Parallel and GPU-accelerated implementations* to reduce runtime for large-scale datasets.

  These approaches provide a foundation for scalable clustering without compromising result quality.

- **Hardware-Aware Implementation and Benchmarking:**
  The authors perform extensive experimental evaluations, highlighting:

  - Significant speedups from GPU and SIMD-based implementations.
  - Improved cache efficiency and memory locality from grid and index-based methods.

- Near-linear scalability in practice, contrary to theoretical worst-case expectations.

These findings demonstrate that DBSCAN can be viable for millions of points on commodity hardware.

- **Real-World Dataset Evaluation:**
The experimental setup includes:
  - Real datasets from diverse domains, such as satellite imagery and wearable sensors.
  - Synthetic datasets designed to stress-test clustering performance under varying densities and dimensions.

  Evaluation metrics include runtime, memory usage, and clustering accuracy under different parameter settings.

- **Use of Heuristics and Approximation Tolerance:**
Khalil and Heinis argue that real-world applications often do not require exact DBSCAN results. Instead, the following heuristics are effective:
  - Use of approximate nearest neighbor methods, including LSH and product quantization.
  - Early termination in region queries once sufficient density is observed.
  - Relaxation of border point definitions for efficiency gains.

  These relaxations often result in negligible quality loss with major performance benefits.

- **Practical Guidelines and Repositioning of DBSCAN:**
The authors present a set of practical recommendations for choosing and deploying DBSCAN:
  - Do not disregard DBSCAN due to theoretical limitations—benchmark its performance empirically.
  - Use index structures and hardware accelerators for real-time applications.
  - Consider clustering tolerance when selecting ε and MinPts, especially in high-dimensional spaces.

  Overall, the paper reestablishes DBSCAN as a powerful and practical tool in the clustering landscape, especially when tuned with modern techniques.

2.2 Gap Analysis

While the reviewed literature offers valuable insights into the theoretical foundations, practical implementations, and approximation strategies for DBSCAN, a critical limitation remains largely unaddressed: the role of dimensional complexity in performance scalability.

- **Putting emphasis on dimension.** Although some papers briefly mention the impact of high-dimensional data, none of them place adequate emphasis on the fact that in modern machine learning tasks, the number of features (i.e., dimensions) can be arbitrarily large. As a result, algorithmic efficiency must be evaluated not just in terms of data points size n, but also in terms of the dimension d. An algorithm where dimension comes in the power of time complexity analysis , such an algorithm will not be accepted . In [2] while finding core points, the author had to check for points in all its epsilon neighborhood. In higher dimension, such operations would be intractable because the number of neighborhood grids becomes huge as dimension increases.

$$\text{Grid length} = \frac{\varepsilon}{\sqrt{d}}$$

$$\text{Number of 1D grids in } \varepsilon\text{-neighborhood } > \left\lfloor \frac{2\varepsilon}{\varepsilon/\sqrt{d}} \right\rfloor = \left\lfloor 2\sqrt{d} \right\rfloor$$

$$\text{Number of } d\text{-dimensional grids in an } \varepsilon\text{-neighborhood} > \left\lfloor 2\sqrt{d} \right\rfloor^{d}$$

As shown above the number of epsilon neighborhood grids grows in order of

$$O\left(d^{d/2}\right)$$

- **Query-Based vs Grid-Based Methods**

The original DBSCAN algorithm employs a **query-based** method to detect clusters. The key drawback of this approach is its potential inefficiency in the worst-case scenario. Even in the best-case, i.e., output-sensitive conditions, each range query can take up to $O(n)$ time in the worst case.

For example, consider a situation where all points are tightly packed within an ε-diameter ball. In this case, querying the points in its ε-ball of a single point may involve examining all n points, resulting in a time complexity of $O(n^2)$ in worst case for the entire clustering process.

Query-based methods tend to perform better when the dataset is **sparsely distributed**, where neighborhood queries touch fewer points on average.

On the other hand, **grid-based** methods partition the space into uniform cells (or hypercubes) and typically work well when points are concentrated in specific regions. If an appropriate grid length is chosen, such methods can significantly reduce neighborhood search times by localizing computations within adjacent grid cells.

However, the performance of grid-based methods deteriorates as the dimensionality d increases.This is a consequence of the exponential increase in the number of grid cells in a fixed ε-neighborhood, which scales as $O(d^{d/2})$. Hence, while grid-based approaches are advantageous in low-dimensional dense datasets, they face scalability challenges in high-dimensional settings.

To address this gap,To address these gaps, the current thesis proposes an algorithm that is a hybrid of both grid-based and query-based methods. The newly introduced method strikes a balance between theoretical rigor and practical applicability, offering a robust solution for density-based clustering in high-dimensional settings commonly encountered in modern data science and machine learning applications.

Chapter 3

Methodology

3.1 Approach

In this Thesis, we introduce two approximate approaches to calculate DBSCAN: one for computing the exact DBSCAN clustering, and another that implements a variant of DBSCAN optimized for performance.

In our application, the point sets involved in bi-chromatic closest pair (BCP) computations are enclosed within axis-aligned bounding boxes (MBBs). To accelerate the BCP step, we employ an approximate method based on centroid projection.

Instead of performing an exhaustive pairwise distance check between all core points in two MBBs, we compute the centroids of the respective point sets and project all points onto the line connecting these centroids. We then consider only the top k closest point pairs based on projected distance, and perform precise ε-distance checks on those.

This centroid-based projection method significantly reduces computational overhead and is generally accurate in practice. While it may occasionally produce incorrect results in highly adversarial or specially constructed configurations, such pathological structures are extremely rare in real-world machine learning datasets. For most practical purposes, this approximate BCP method is not only efficient but also sufficiently reliable, offering a favorable trade-off between speed and accuracy.

The second method we propose, referred to as **Box-Based Approximate DBSCAN**, simplifies the neighborhood definition in order to achieve greater computational efficiency, particularly in high-dimensional or large-scale datasets.

In this method, we replace the $\varepsilon/\sqrt{d}$ grid length used in the tree construction step with a coarser grid length of ε in each dimension. This transforms the original ε-radius ball criterion into an axis-aligned hypercube (or "box") of side length 2ε. Consequently, core point detection and neighborhood search are carried out with respect to this box, which significantly reduces both tree depth and the number of intersecting grid cells during search.

A major advantage of this approach is that the maximum number of intersecting cells per dimension is limited to just 3 (the current cell and its immediate neighbors), compared to up to $\left\lceil \sqrt{d} \right\rceil + 1$ in the first method. This leads to faster query times and simpler implementation.

While the box-based approximation slightly relaxes the strict DBSCAN definition—potentially overestimating density connectivity near the edges of clusters—it performs well in practice. This makes it particularly suitable for large-scale clustering tasks where speed is prioritized over perfect boundary precision.

3.1.1 Step-by-Step Description of the Algorithm

Step 1: Tree Construction The tree is built recursively by projecting and grouping points dimension-by-dimension using a fixed grid interval. The process can be broken down into the following substeps:

1. **Initial Projection and Sorting:** Project all points onto the 0-th dimension (i.e., their first coordinate). Sort the points based on this dimension.

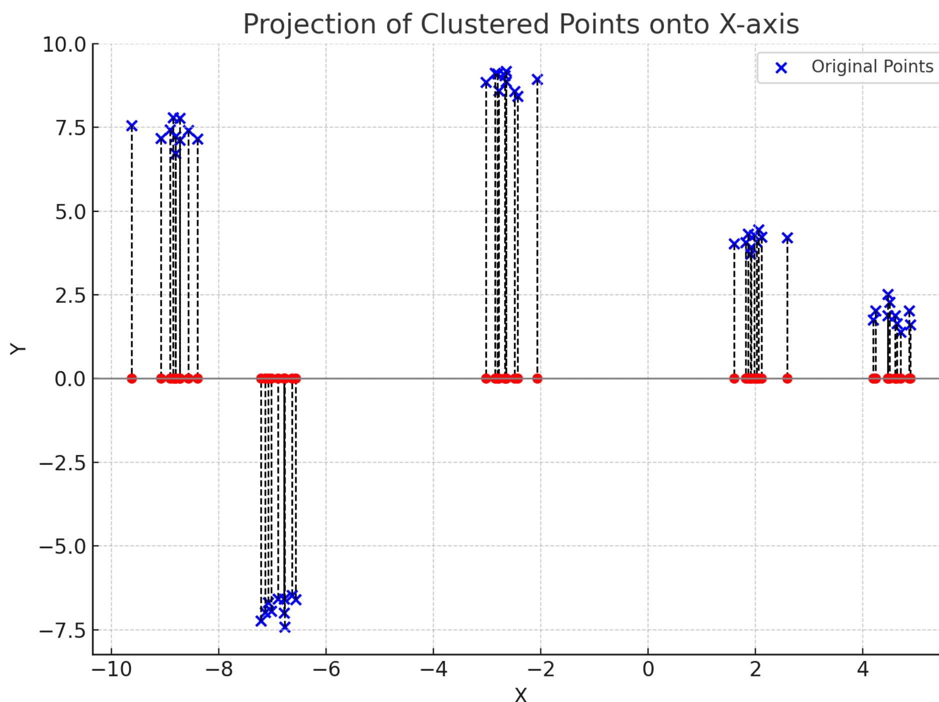

Figure 3.1: Projection of clustered points onto the X-axis. The dotted lines represent projections from each 2D point onto its corresponding location on the X-axis.

2. **Grouping Along 0-th Dimension:** Divide the sorted list of points into groups such that all points in a group lie within an interval of length $\varepsilon/\sqrt{d}$ (or ε in the second method). Each such group becomes a child node of the root.

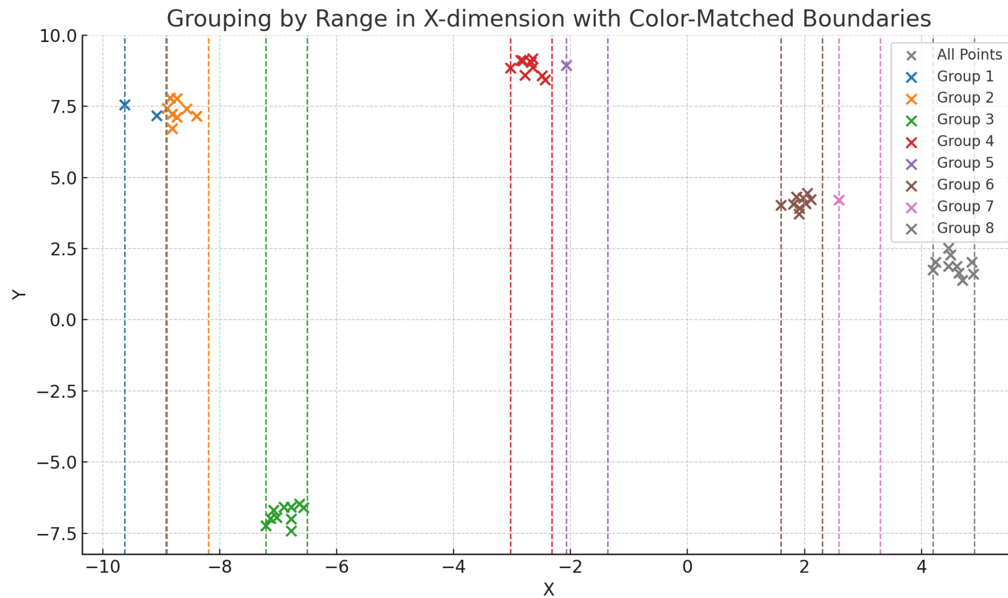

Figure 3.2: Grouping by range in the X-dimension. Each group is formed by scanning the sorted list of points along the X-axis and collecting those within a fixed-width interval. Vertical dashed lines mark group boundaries and are color-matched to the group points.

3. **Recursive Grouping:** For each group formed at the 0-th level, recursively repeat the process:
   - Project the group's points onto the next dimension.
   - Sort and re-group based on the fixed interval.
   - Continue this process up to the $(d-1)$-th dimension.

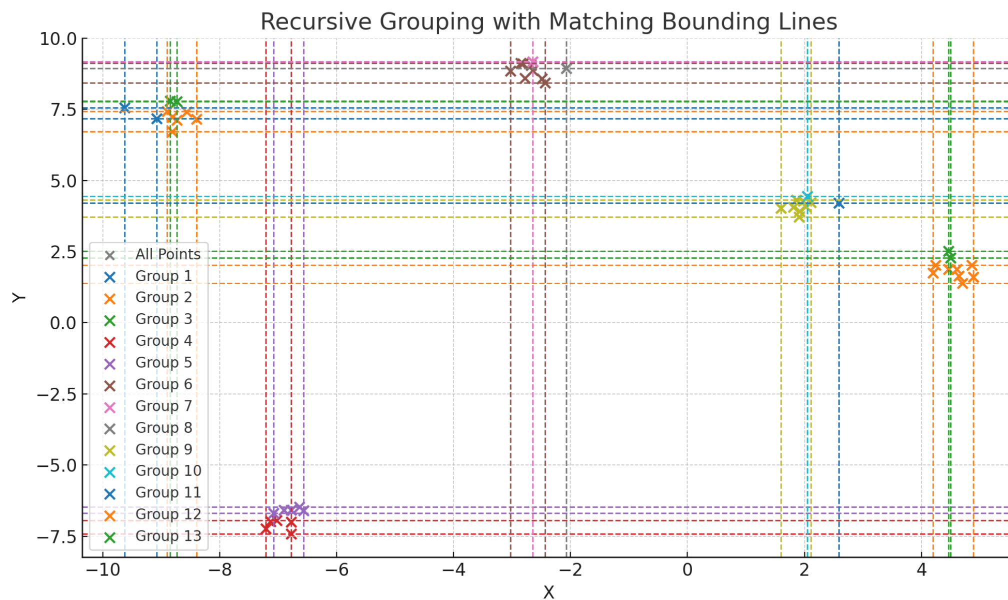

Figure 3.3: Recursive grouping with matching bounding lines. Points are first grouped along the X-axis, and then each group is recursively subdivided along the Y-axis.

4. **Leaf Node Formation:** After processing all d dimensions, each final group becomes a leaf node of the tree. These nodes contain the indices of the points they represent.

5. **Minimum Bounding Box (MBB) Refinement:** Instead of using the interval-derived bounds directly, compute the true Minimum Bounding Box (MBB) by scanning the actual coordinates of all points in the leaf node and determining the minimum and maximum values in each dimension. This ensures tighter spatial bounds for efficient neighborhood search in later steps.

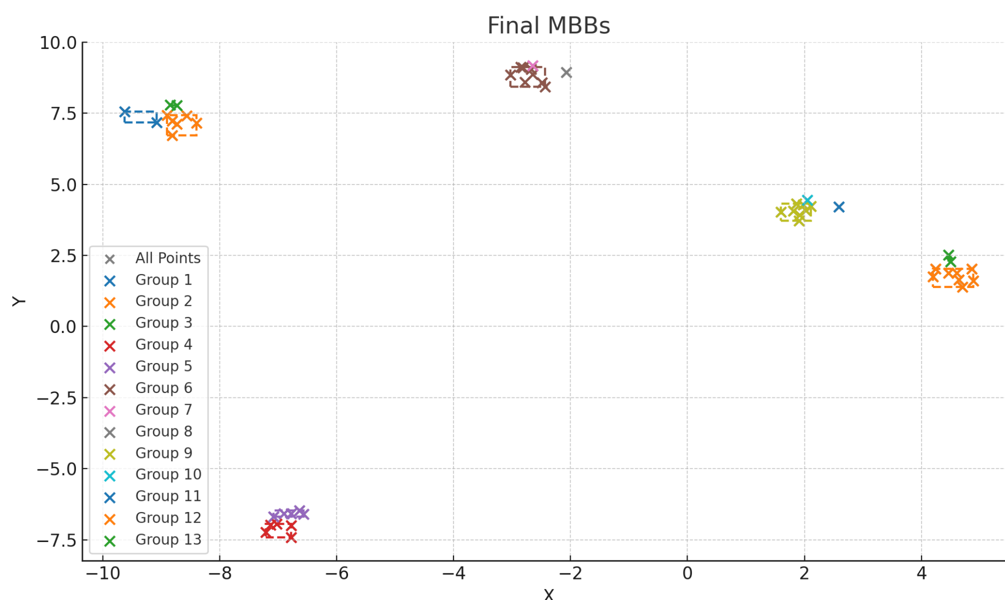

Figure 3.4: Final MBBs obtained through recursive grouping in X and Y dimensions. Each point group is enclosed within a minimum bounding box (MBB) formed by applying fixed-width range grouping first along the X-axis and then along the Y-axis. The bounding lines are color-matched to their respective point groups.

6. **Tree Metadata Storage:** Internal nodes store the dimension along which the split occurred, along with min/max bounds of their child groups. Leaf nodes store the list of point indices and their MBBs.

Data Structures and Pseudocodes:

- `TreeNode`: {dimension, children, min_values, max_values, point_indices, min_bounding_box}
- `MBBNode`: {mbb, neighbors}

Algorithm 1 Argsort by Component

```
1: procedure ArgsortByComponent(points, k)
2:     N ← length of points                              ▷ Number of points
3:     indices ← array of integers from 0 to N − 1
4:     Sort indices using comparison:
5:         a < b when points[a][k] < points[b][k]  ▷ Sort indices based on k-th component
       return indices
6: end procedure
```

Algorithm 2 Group Points by Range

```
1: procedure GroupByRange(sorted_indices, points, k, r)
2:     groups ← empty list of lists
3:     n ← length of sorted_indices
4:     i ← 0
5:     while i < n do
6:         current_group ← empty list
7:         base ← points[sorted_indices[i]][k]        ▷ Get k-th coordinate of first point
8:         threshold ← base + r
9:         while i < n and points[sorted_indices[i]][k] < threshold do
10:            current_group.append(sorted_indices[i])
11:            i ← i + 1
12:        end while
13:        groups.append(current_group)
14:    end while
       return groups
15: end procedure
```

Algorithm 3 Build Spatial Index Tree

```
1:  procedure BuildTree(points, indices, dimension, d, ϵ)
2:      if indices is empty then return null                    ▷ Base case: no points to store
3:      end if
4:      node ← new TreeNode(dimension)                                   ▷ Create new node
5:      if dimension == d then                                              ▷ Leaf node case
6:          subset ← empty list
7:          for idx in indices do
8:              subset.add(points[idx])
9:          end for
10:         (min_corner, max_corner) ← ComputeMinBoundingBox(subset)
11:         node.point_indices ← indices
12:         node.min_bounding_box ← (min_corner, max_corner)
13:         leaf_nodes.add(node)                         ▷ Register leaf node return node
14:     end if
15:                                                                     ▷ Internal node case
16:     r_group ← ϵ/√d                                        ▷ Compute grouping threshold
17:     sorted_indices ← sort indices by points[·][dimension]
18:     groups ← GroupByRange(sorted_indices, points, dimension, r_group)
19:     for group in groups do
20:         if group is empty then
21:         end if
22:         child ← BuildTree(points, group, dimension + 1, d, ϵ)
23:         if child ≠ null then
24:             node.children.add(child)
25:             min_val ← +∞, max_val ← −∞
26:             for idx in group do
27:                 val ← points[idx][dimension]
28:                 min_val ← min(min_val, val)
29:                 max_val ← max(max_val, val)
30:             end for
31:             node.min_values.add(min_val)
32:             node.max_values.add(max_val)
33:         end if
34:     end for
         return node
35: end procedure
```

Algorithm 4 Compute Minimum Bounding Box

```
1:  procedure ComputeMinBoundingBox(points)
2:      if points is empty then return (∅, ∅)        ▷ Return empty pair for empty input
3:      end if
4:      d ← dimension of points                      ▷ Number of components in each point
5:      min_corner ← array of size d initialized to +∞
6:      max_corner ← array of size d initialized to −∞
7:      for each point in points do
8:          for i ← 0 to d − 1 do
9:              min_corner[i] ← min(min_corner[i], point[i])
10:             max_corner[i] ← max(max_corner[i], point[i])
11:         end for
12:     end for
         return (min_corner, max_corner)
13: end procedure
```

Time Complexity Analysis of Step 1: Tree Construction

In Step 1, we construct a hierarchical tree-based spatial index where each level of the tree corresponds to one of the d dimensions of the input data. The construction process operates as follows:

- At each level i (where $1 \leq i \leq d$), the set of points is projected onto the i-th dimension.
- The points are then sorted according to their coordinates in the i-th dimension.
- Based on these sorted values, the points are grouped into ranges, and recursive subtrees are built on each group in the next dimension.

Since sorting n points takes $O(n \log n)$ time and we perform this sorting operation independently at each of the d levels, the total time complexity for building the entire tree is:

$$T(n, d) = d \cdot O(n \log n) = O(d \cdot n \log n) \tag{3.1}$$

Thus, the tree construction phase has a time complexity of $O(d \cdot n \log n)$.

Step 2: Core Point Identification via Neighborhood Search

After constructing the tree, we identify core points by leveraging Minimum Bounding Boxes (MBBs) and performing efficient ε-neighborhood searches. This step is broken into the following substeps:

1. **Neighborhood Expansion:** For each leaf node (denoted as the query node), expand its MBB by subtracting ε from the lower bounds and adding ε to the upper bounds along each dimension. This defines an expanded hyperrectangle representing the ε-neighborhood region.

2. **Neighborhood MBB Search:** For the expanded MBB of the query node, iterate through all other leaf nodes and check whether their MBBs intersect with the expanded region. If so, mark them as potential neighbors. The intersection check is performed dimension-wise:

   $$\text{Check: } [b_{\min}^{(i)}, b_{\max}^{(i)}] \cap [q_{\min}^{(i)} - \varepsilon, q_{\max}^{(i)} + \varepsilon] \neq \emptyset$$

   for each dimension i. This intersection check can be done by binary search on sorted min max arrays present in the parent node.

3. **Core Point Evaluation:** For each point p in the query node:

   - Initialize a neighbor count with the number of points in the same MBB (since intra-MBB points are guaranteed to be within ε distance).
   - For each neighboring MBB, check all contained points. Increment the count for each point q such that distance$(p, q) \leq \varepsilon$.
   - If the total count $\geq$ `MinPts`, label p as a **core point**.

4. **Core Set Aggregation:** Collect all core points found in the current MBB. If non-empty, store:

   - the list of core point indices
   - a reference to the corresponding MBB (tree node)

   for use in the clustering step.

Algorithm 5 Spatial Tree Search

```
1: procedure SearchTree(node, query_min, query_max, r, d, result, skip_node)
2:     if node is null then return                        ▷ Base case: empty node
3:     end if
4:     dim ← node.dimension
5:     expanded_low ← query_min[dim] − r
6:     expanded_high ← query_max[dim] + r
7:     if node is leaf node then
8:         if node ≠ skip_node then
9:             result.append((node.MBB.min, node.MBB.max, node.point_indices, node))
10:        end ifreturn
11:    end if
12:                                             ▷ Binary search for relevant children
13:    lower ← first index where node.max_values ≥ expanded_low
14:    upper ← first index where node.min_values > expanded_high
15:    for i ← lower to upper − 1 do
16:        SearchTree(node.children[i], query_min, query_max, r, d, result, skip_node)
17:    end for
18: end procedure
```

Algorithm 6 Identify Core Points

```
 1: procedure IdentifyCorePoints(points, root, ε, MinPts, d)
 2:     core_points ← empty set                              ▷ Use set to avoid duplicates
 3:     for each node in leaf_nodes do
 4:         r_neigh ← empty list of tuples
 5:         node_min ← node.min_bounding_box[0]
 6:         node_max ← node.min_bounding_box[1]
 7:                                                   ▷ Find all neighboring MBBs within ε
 8:         SearchTree(root, node_min, node_max, ε, d, r_neigh, node)
 9:         for each pi in node.point_indices do
10:             count ← size(node.point_indices)   ▷ Include all points in the same MBB
11:             for each (minb, maxb, indices, ptr) in r_neigh do
12:                 for each idx in indices do
13:                     if EuclideanDistance(points[pi], points[idx]) ≤ ε then
14:                         count ← count + 1
15:                         if count ≥ MinPts then break
16:                         end if
17:                     end if
18:                 end for
19:                 if count ≥ MinPts then break
20:                 end if
21:             end for
22:             if count ≥ MinPts then
23:                 core_points.insert(pi)                          ▷ Mark as core point
24:             end if
25:         end for
26:     end for
        return core_points converted to vector
27: end procedure
```

Time Complexity of Step 2: Core Point Detection via Neighborhood Search

In Step 2, the algorithm identifies core points by checking whether each point has at least `MinPts` neighbors within an ε-radius. The neighborhood queries are optimized using the spatial tree structure built in Step 1. However, in the worst-case scenario, the time complexity of this step can degrade significantly due to certain pathological data configurations.

Worst-Case Time Complexity. In the worst case, each point may need to be compared against $O(n)$ other points to determine its ε-neighborhood. This leads to a time complexity of:

$$T_{\text{step2-worst}} = O(n^2 \cdot d) \tag{3.2}$$

Such a worst-case scenario arises when the data points are structured in a highly adversarial manner. For example, if the points lie at the centers of a dense d-dimensional grid, where each grid cell has side length approximately ε, then the ε-ball around each point may intersect with a large number of other points, effectively triggering $O(n)$ neighborhood checks per point.

Practical Considerations. While the worst-case complexity is quadratic in nature, such pathological configurations are extremely rare in real-world data science applications. In practice:

- Most datasets exhibit sparsity or irregular clustering structure.
- The spatial tree built in Step 1 significantly prunes unnecessary comparisons.
- Additional optimizations (e.g., early stopping once `MinPts` neighbors are found, MBB-level filtering) further reduce the effective runtime.

Therefore, although the theoretical worst-case time complexity is:

$$O(n^2 \cdot d)$$

in practice, the algorithm often runs much faster due to the spatial organization and neighborhood pruning heuristics.

Step 3: Clustering of Core Points Once core points have been identified and grouped into their corresponding MBBs (leaf nodes), we construct clusters by connecting nearby core point sets. The steps involved are as follows:

1. **Graph Construction:** Treat each MBB that contains at least one core point as a node in an undirected graph. Two MBBs are considered connected if they contain at least one pair of core points that are within ε distance of each other.
2. **Neighborhood MBB Detection:** For each MBB M_i, expand its MBB by ε in every dimension and find all other MBBs M_j whose bounding boxes intersect the expanded region. These are potential candidates for connectivity.
3. **Approximate Bi-Chromatic Closest Pair (BCP) Test:** For each intersecting MBB pair (M_i, M_j), apply the following efficient approximation to test whether they should be connected:
   - Compute the centroid of the core points in M_i and M_j
   - Project all core points in both sets onto the line joining the centroids
   - Select the top k closest projected points from each set
   - Perform pairwise ε-distance checks on only the $k \times k$ point pairs
   - If any such pair is within ε, add an undirected edge between M_i and M_j

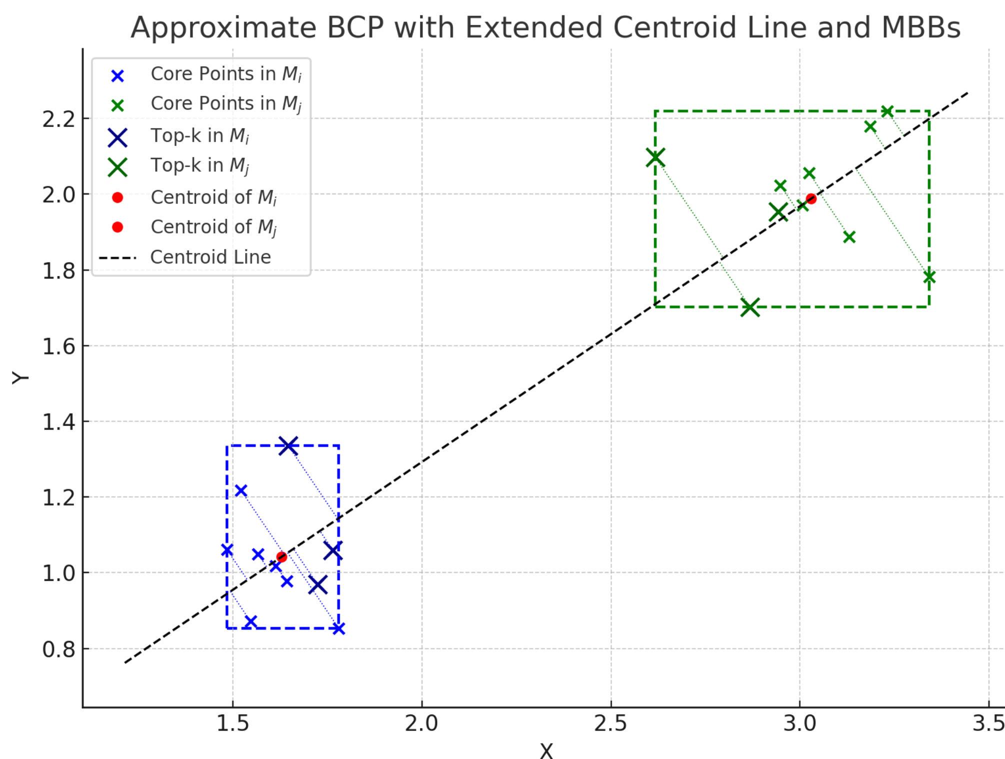

Figure 3.5: Approximate Bi-Chromatic Closest Pair (BCP) Test: Core points from two MBBs are projected onto the centroid-to-centroid line. Top-k projected points from each MBB are marked, and only these pairs are considered for ε-distance testing. Dotted rectangles represent the Minimum Bounding Boxes (MBBs) of each core set.

4. **Connected Component Extraction:** Apply a graph traversal algorithm (e.g., Depth-First Search or Union-Find) to find connected components in the MBB graph. Each connected component represents one DBSCAN cluster.

5. **Cluster Label Assignment:** Assign a unique cluster label to all core points in MBBs belonging to the same connected component.

6. **Border and Noise Classification (Optional):** After labeling all core points, iterate over remaining non-core points:

   - If a non-core point lies within ε of any core point in a cluster, label it as a **border point** and assign the corresponding cluster ID.
   - Otherwise, mark it as **noise**.

Algorithm 7 Find ϵ-Neighbors for MBB

```
1: procedure FindEpsilonNeighbors(target_mbb, root, ϵ, d)
2:     neighbors ← ∅                                    ▷ Initialize empty neighbor list
3:                                                      ▷ Extract target MBB boundaries
4:     target_min ← target_mbb.min_bounding_box[0]
5:     target_max ← target_mbb.min_bounding_box[1]
6:                                                      ▷ Search tree for overlapping MBBs
7:     SearchTree(root, target_min, target_max, ϵ, d, neighbors, target_mbb)
8:     neighbor_indices ← ∅
9:     for each (min_corner, max_corner, point_indices, node_ptr) in neighbors do
10:        idx ← FindIndex(leaf_nodes, node_ptr)
11:        if idx ≠ not found then
12:            neighbor_indices.append(idx)
13:        end if
14:    end for
        return neighbor_indices
15: end procedure
16: function FindIndex(list, item)
17:    return iterator position of item in list or -1 if not found
18: end function
```

Algorithm 8 Optimized Core Pair Check with Centroid Filtering

```
1: procedure HasEpsilonCorePair(mbb1, mbb2, points, core_points, ε, k)
2:     Step 1: Collect Core Points
3:     core_mbb1, core_mbb2 ← ∅
4:     for idx in mbb1.point_indices do
5:         if idx ∈ core_points then core_mbb1.add(idx)
6:         end if
7:     end for
8:     for idx in mbb2.point_indices do
9:         if idx ∈ core_points then core_mbb2.add(idx)
10:        end if
11:    end for
12:    if core_mbb1 is empty or core_mbb2 is empty then return        ▷ Early exit if no
       core points
13:    end if
14:    Step 2: Compute Centroids
15:    c1 ← vector of zeros with dimension d
16:    c2 ← vector of zeros with dimension d
17:    for idx in core_mbb1 do
18:        for dim ← 0 to d − 1 do
19:            c1[dim] ← c1[dim] + points[idx][dim]
20:        end for
21:    end for
22:    for idx in core_mbb2 do
23:        for dim ← 0 to d − 1 do
24:            c2[dim] ← c2[dim] + points[idx][dim]
25:        end for
26:    end for
27:    c1 ← c1/|core_mbb1|                                              ▷ Normalize
28:    c2 ← c2/|core_mbb2|                                              ▷ Normalize
29:    Step 3: Project Points
30:    dir_{c1→c2} ← c2 − c1                                            ▷ Direction vector
31:    dir_{c2→c1} ← −dir_{c1→c2}                                       ▷ Reverse direction
32:    proj_mbb1 ← ProjectPoints(core_mbb1, c1, dir_{c1→c2})
33:    proj_mbb2 ← ProjectPoints(core_mbb2, c2, dir_{c2→c1})
34:    Sort proj_mbb1, proj_mbb2 descending by projection value
```

Algorithm 9 Optimized Core Pair Check with Centroid Filtering(Continued)

1: **Step 4: Check Top-k Pairs**
2: $\epsilon_{sq} \leftarrow \epsilon^2$
3: $k1 \leftarrow \min(k, |proj_mbb1|)$, $k2 \leftarrow \min(k, |proj_mbb2|)$
4: **for** $i \leftarrow 0$ to $k1 - 1$ **do**
5: $(proj1, idx1) \leftarrow proj_mbb1[i]$
6: $p1 \leftarrow points[idx1]$
7: **for** $j \leftarrow 0$ to $k2 - 1$ **do**
8: $(proj2, idx2) \leftarrow proj_mbb2[j]$
9: $p2 \leftarrow points[idx2]$
10: $dist_{sq} \leftarrow 0$
11: **for** $dim \leftarrow 0$ to $d - 1$ **do**
12: $diff \leftarrow p1[dim] - p2[dim]$
13: $dist_{sq} \leftarrow dist_{sq} + diff^2$
14: **if** $dist_{sq} > \epsilon_{sq}$ **then break**
15: **end if**
16: **end for**
17: **if** $dist_{sq} \leq \epsilon_{sq}$ **then return**
18: **end if**
19: **end for**
20: **end forreturn**
21:
22: **function** ProjectPoints(*indices*, *center*, *direction*)
23: $projections \leftarrow \emptyset$
24: **for** idx in $indices$ **do**
25: $proj \leftarrow 0$
26: **for** $dim \leftarrow 0$ to $d - 1$ **do**
27: $proj \leftarrow proj + (points[idx][dim] - center[dim]) \times direction[dim]$
28: **end for**
29: $projections.\text{add}((proj, idx))$
30: **end forreturn** $projections$
31: **end function**

Time Complexity of Approximate Bichromatic Closest Pair (BCP) Algorithm

Let A and B be two sets of core points with cardinalities $m_1 = |A|$ and $m_2 = |B|$, respectively. The goal is to efficiently determine whether there exists a pair $(a, b) \in A \times B$ such that the Euclidean distance between a and b is less than or equal to a given threshold

ε. To reduce the computational cost, an approximate method is used based on projection onto the line joining the centroids of A and B, followed by a top-k brute-force comparison.

Step-by-step Time Complexity:

- **Centroid Computation and Line Calculation:**
  Computing the centroids of sets A and B, and then deriving the vector between them, requires summing each coordinate of all points in both sets:

$$T_{\text{centroid}} = O(m_1 \cdot d + m_2 \cdot d)$$

- **Projection and Sorting:**
  Each point in A and B is projected onto the line connecting the two centroids, which takes $O(d)$ time per point. The projections are then sorted:

$$T_{\text{project+sort}} = O(m_1 \cdot \log m_1 \cdot d + m_2 \cdot \log m_2 \cdot d)$$

- **Top-k Selection and Brute Force Distance Check:**
  After sorting, the top k projected points from each set are selected. A brute-force distance check is performed on all $k \times k$ pairs:

$$T_{\text{brute}} = O(k^2 \cdot d)$$

Total Time Complexity: Combining all the steps, the overall time complexity of the approximate BCP check is:

$$T_{\text{approx_BCP}} = O\left((m_1 + m_2) \cdot d + (m_1 \log m_1 + m_2 \log m_2) \cdot d + k^2 \cdot d\right) \tag{3.3}$$

This approximate method significantly reduces the computation compared to the exact $O(m_1 \cdot m_2 \cdot d)$ brute-force BCP, especially when $k \ll m_1, m_2$.

Algorithm 10 Build MBB Connectivity Graph

```
1: procedure BuildMBBGraph(points, root, ϵ, d, core_points)
2:     Initialize mbb_graph with size |leaf_nodes|
3:     for i ← 0 to |leaf_nodes| − 1 do
4:         current_mbb ← leaf_nodes[i]
5:         mbb_graph[i].mbb ← current_mbb
6:                                              ▷ Find all MBBs within ϵ distance
7:         neighbor_indices ← FindEpsilonNeighbors(current_mbb, root, ϵ, d)
8:         for j in neighbor_indices do
9:             neighbor_mbb ← leaf_nodes[j]
10:            if HasCorePair(current_mbb, neighbor_mbb, points, core_points, ϵ)
   then
11:                mbb_graph[i].neighbors.add(j)
12:            end if
13:        end for
14:    end for
15: end procedure
```

Algorithm 11 Depth-First Search for MBB Graph

```
1: procedure DFS(node_idx, visited, component, graph)
2:     visited[node_idx] ←                       ▷ Mark current node as visited
3:     component.append(node_idx)                ▷ Add to current component
4:     for each neighbor in graph[node_idx].neighbors do
5:         if not visited[neighbor] then
6:             DFS(neighbor, visited, component, graph)    ▷ Recursive exploration
7:         end if
8:     end for
9: end procedure
```

Algorithm 12 Find Connected Components in MBB Graph

```
1:  procedure FindConnectedComponents(graph)
2:      components ← ∅                              ▷ Initialize empty component list
3:      visited ← array of size |graph| initialized to
4:      for i ← 0 to |graph| − 1 do
5:          if not visited[i] then
6:              component ← ∅
7:              DFS(i, visited, component, graph)
8:              components.append(component)
9:          end if
10:     end for
        return components
11: end procedure
```

Algorithm 13 Assign Points to Clusters (Excluding Noise)

```
1:  procedure AssignClusters(components, leaf_nodes, noise_points)
2:      clusters ← ∅                                ▷ Initialize empty cluster list
3:      for each component in components do
4:          cluster ← ∅
5:          for each mbb_idx in component do
6:              for each point_idx in leaf_nodes[mbb_idx].point_indices do
7:                  if point_idx ∉ noise_points then
8:                      cluster.add(point_idx)
9:                  end if
10:             end for
11:         end for
12:         if cluster is not empty then
13:             clusters.add(cluster)
14:         end if
15:     end for
        return clusters
16: end procedure
```

Time Complexity of Step 3: Core-Component Graph Construction

In Step 3, the algorithm constructs a graph over the set of MBBs (Minimum Bounding Boxes) that contain core points. Each MBB is treated as a node, and an undirected edge is added between two MBBs if they contain at least one pair of core points (one from each MBB) within ε-distance of each other.

Worst-Case Time Complexity. The time complexity in this step is similar to that of Step 2. In the worst-case scenario, each MBB may need to be compared with a large number of other MBBs, and the connectivity check between MBBs involves pairwise distance computations between subsets of core points.

If there are $O(n)$ MBBs and each contains up to $O(1)$ to $O(n)$ core points, then the total number of MBB pair comparisons can be quadratic. As a result, the worst-case time complexity is:

$$T_{\text{step3-worst}} = O(n^2 \cdot d) \tag{3.4}$$

Such a worst-case arises in pathological situations, for example when the points are arranged at the centers of a dense d-dimensional grid with spacing close to ε, causing widespread overlapping of neighborhoods and MBB connections.

Variant Method: Box-Based Approximate DBSCAN.

The second method is a variant of the exact DBSCAN approach, designed to improve performance by simplifying the neighborhood criterion.

The only two changes lie in the grouping step during tree construction: instead of partitioning each dimension using grid intervals of length $\varepsilon/\sqrt{d}$, we use a grid length of ε and we use the maximum dimensional gap function instead of the Euclidean distance function.

Algorithm 14 Maximum Dimensional Gap Calculation

```
1: function MaximumDimensionalGap(a, b)
2:     max_gap ← −∞                          ▷ Initialize with minimum possible value
3:     for i ← 0 to length(a) − 1 do
4:         current_gap ← |a[i] − b[i]|                      ▷ Absolute difference
5:         if current_gap > max_gap then
6:             max_gap ← current_gap
7:         end if
8:     end for
        return max_gap
9: end function
```

As a result, rather than checking whether points lie within an ε-radius ball (as in standard DBSCAN), we approximate the neighborhood by checking whether points fall within a hypercube (box) of side length 2ε. This relaxed condition greatly reduces the number of distance computations and improves efficiency in higher dimensions, at the cost of slightly reduced precision in cluster boundaries.All other components of the algorithm—including tree construction logic, MBB neighborhood search, core point identification, and clustering—remain unchanged.

Chapter 4

Implementation

4.1 C++ Implementation

We have implemented the proposed DBSCAN-based clustering algorithms using C++. The following listing provides the complete source code implementing the optimized tree-based structure, core point detection, and clustering logic.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <memory>
#include <sstream>
#include <climits>
#include <cmath>
#include <unordered_set>
#include <unordered_map>

using namespace std;

// ==================== Tree Node Definition ====================
struct TreeNode {
    int dimension;
    vector<shared_ptr<TreeNode>> children;

    // Only for internal nodes
    vector<int> min_values;
    vector<int> max_values;

    // Only for leaf nodes
    vector<int> point_indices;
    vector<vector<int>> min_bounding_box;  // {min_corner, max_corner}

    TreeNode(int dim) : dimension(dim) {}
};

vector<shared_ptr<TreeNode>> leaf_nodes;  // Store pointers to leaf nodes

// ==================== Graph Node for MBBs ====================
struct MBBNode {
    shared_ptr<TreeNode> mbb;  // Pointer to the leaf node (MBB)
    vector<int> neighbors;     // Indices of connected MBBs (in leaf_nodes)
};

vector<MBBNode> mbb_graph;  // Graph representation

// ==================== Helper Functions ====================
vector<int> argsort_by_component(const vector<vector<int>>& points, int k) {
    int N = points.size();
```

```
    vector<int> indices(N);
    for (int i = 0; i < N; ++i)
        indices[i] = i;

    sort(indices.begin(), indices.end(), [&](int a, int b) {
        return points[a][k] < points[b][k];
    });
    return indices;
}

vector<vector<int>> group_by_range(const vector<int>& sorted_indices, const vector<vector<int>>& points, int k,
     double r) {
    vector<vector<int>> groups;
    int n = sorted_indices.size();
    int i = 0;

    while (i < n) {
        vector<int> current_group;
        int base = points[sorted_indices[i]][k];
        int threshold = base + r;

        while (i < n && points[sorted_indices[i]][k] < threshold) {
            current_group.push_back(sorted_indices[i]);
            ++i;
        }
        groups.push_back(current_group);
    }
    return groups;
}

pair<vector<int>, vector<int>> compute_min_bounding_box(const vector<vector<int>>& points) {
    if (points.empty()) return {{}, {}};
    int d = points[0].size();
    vector<int> min_corner(d, INT_MAX);
    vector<int> max_corner(d, INT_MIN);

    for (const auto& point : points) {
        for (int i = 0; i < d; ++i) {
            min_corner[i] = min(min_corner[i], point[i]);
            max_corner[i] = max(max_corner[i], point[i]);
        }
    }
    return {min_corner, max_corner};
}

double euclidean_distance(const vector<int>& a, const vector<int>& b) {
    double sum = 0;
    for (size_t i = 0; i < a.size(); ++i)
        sum += (a[i] - b[i]) * (a[i] - b[i]);
    return sqrt(sum);
}

//For Box Based DBSCAN
double maximum_dimensional_gap(const vector<int>& a, const vector<int>& b) {
    double mx = -10000.0;
    int value=0;
    for (size_t i = 0; i < a.size(); ++i)
        value=abs(a[i]-b[i]);
        if (value>mx)
        {
            mx=value;
        }

    return mx;
}

// ==================== Build Tree Recursively ====================
shared_ptr<TreeNode> build_tree(const vector<vector<int>>& points, const vector<int>& indices, int dimension, int d,
     double epsilon) {
    if (indices.empty()) return nullptr;
    auto node = make_shared<TreeNode>(dimension);

    if (dimension == d) {
        // Leaf node
        vector<vector<int>> subset;
        for (int idx : indices)
            subset.push_back(points[idx]);

        auto [min_corner, max_corner] = compute_min_bounding_box(subset);
        node->point_indices = indices;
```

```
120         node->min_bounding_box = {min_corner, max_corner};
121 
122         leaf_nodes.push_back(node);  // collect leaf node
123         return node;
124     }
125 
126     double r_group = epsilon / sqrt(d);
127 
128     vector<int> sorted_indices = indices;
129     sort(sorted_indices.begin(), sorted_indices.end(), [&](int a, int b) {
130         return points[a][dimension] < points[b][dimension];
131     });
132 
133     vector<vector<int>> groups = group_by_range(sorted_indices, points, dimension, r_group);
134 
135     for (const auto& group : groups) {
136         if (group.empty()) continue;
137         auto child = build_tree(points, group, dimension + 1, d, epsilon);
138         if (child) {
139             node->children.push_back(child);
140 
141             int min_val = INT_MAX, max_val = INT_MIN;
142             for (int idx : group) {
143                 int val = points[idx][dimension];
144                 min_val = min(min_val, val);
145                 max_val = max(max_val, val);
146             }
147             node->min_values.push_back(min_val);
148             node->max_values.push_back(max_val);
149         }
150     }
151     return node;
152 }
153 
154 // ==================== Search Tree ====================
155 void search_tree(const shared_ptr<TreeNode>& node, const vector<int>& query_min, const vector<int>& query_max,
156                  double r, int d, vector<tuple<vector<int>, vector<int>, vector<int>, shared_ptr<TreeNode>>>& result,
157                  shared_ptr<TreeNode> skip_node = nullptr) {
158     if (!node) return;
159 
160     int dim = node->dimension;
161     int expanded_dim_low=query_min[dim]-r;
162     int expanded_dim_high=query_max[dim]+r;
163 
164 
165     if (node->children.empty()) {
166         if (node != skip_node) {
167             result.push_back({node->min_bounding_box[0], node->min_bounding_box[1], node->point_indices, node});
168         }
169         return;
170     }
171 
172     int lower = lower_bound(node->max_values.begin(), node->max_values.end(), expanded_dim_low) -
         node->max_values.begin();
173     int upper = upper_bound(node->min_values.begin(), node->min_values.end(), expanded_dim_high) -
         node->min_values.begin();
174 
175     for (int i = lower; i < upper; ++i) {
176         search_tree(node->children[i], query_min, query_max, r, d, result, skip_node);
177     }
178 }
179 
180 // ==================== Identify Core Points ====================
181 vector<int> identify_core_points(const vector<vector<int>>& points, shared_ptr<TreeNode> root, double epsilon, int
         MinPts, int d) {
182     unordered_set<int> core_points;
183 
184     for (const auto& node : leaf_nodes) {
185         vector<tuple<vector<int>, vector<int>, vector<int>, shared_ptr<TreeNode>>> r_neigh;
186         search_tree(root, node->min_bounding_box[0], node->min_bounding_box[1], epsilon, d, r_neigh, node);
187 
188         for (int pi : node->point_indices) {
189             int count = node->point_indices.size(); // Include all points in the same MBB
190 
191             for (const auto& [minb, maxb, indices, ptr] : r_neigh) {
192                 for (int idx : indices) {
193                     if (euclidean_distance(points[pi], points[idx]) <= epsilon) {
194                         ++count;
195                         if (count >= MinPts) break;
196                     }
```

```
                }
                if (count >= MinPts) break;
            }

            if (count >= MinPts) core_points.insert(pi);
        }
    }

    return vector<int>(core_points.begin(), core_points.end());
}

// ==================== Find   -Neighbor MBBs ====================
vector<int> find_epsilon_neighbors(
    const shared_ptr<TreeNode>& target_mbb,
    const shared_ptr<TreeNode>& root,
    double epsilon,
    int d)
{
    vector<tuple<vector<int>, vector<int>, vector<int>, shared_ptr<TreeNode>>> neighbors;
    search_tree(root, target_mbb->min_bounding_box[0], target_mbb->min_bounding_box[1], epsilon, d, neighbors,
     target_mbb);

    vector<int> neighbor_indices;
    for (const auto& [minb, maxb, indices, ptr] : neighbors) {
        auto it = find(leaf_nodes.begin(), leaf_nodes.end(), ptr);
        if (it != leaf_nodes.end()) {
            neighbor_indices.push_back(it - leaf_nodes.begin());
        }
    }
    return neighbor_indices;
}

// ==================== Check Core Point Pairs ====================
// ==================== Centroid-Based Top-k Optimization ====================
bool has_epsilon_core_pair(
    const shared_ptr<TreeNode>& mbb1,
    const shared_ptr<TreeNode>& mbb2,
    const vector<vector<int>>& points,
    const unordered_set<int>& core_points,
    double epsilon,
    int k = 5)  // Top-k points to consider
{
    // --- Step 1: Collect core points in both MBBs ---
    vector<int> core_mbb1, core_mbb2;
    for (int idx : mbb1->point_indices)
        if (core_points.count(idx)) core_mbb1.push_back(idx);
    for (int idx : mbb2->point_indices)
        if (core_points.count(idx)) core_mbb2.push_back(idx);

    if (core_mbb1.empty() || core_mbb2.empty())
        return false;  // No core points in one MBB

    // --- Step 2: Compute centroids C1 and C2 ---
    vector<double> c1(points[0].size(), 0.0), c2(points[0].size(), 0.0);
    for (int idx : core_mbb1)
        for (int dim = 0; dim < c1.size(); ++dim)
            c1[dim] += points[idx][dim];
    for (int idx : core_mbb2)
        for (int dim = 0; dim < c2.size(); ++dim)
            c2[dim] += points[idx][dim];

    for (double& val : c1) val /= core_mbb1.size();
    for (double& val : c2) val /= core_mbb2.size();

    // --- Step 3: Project core points onto  C 1 C2  and  C 2 C1  ---
    auto project_points = [&](const vector<int>& indices, const vector<double>& center, const vector<double>&
     direction) {
        vector<pair<double, int>> projections;
        for (int idx : indices) {
            double projection = 0.0;
            for (int dim = 0; dim < direction.size(); ++dim) {
                projection += (points[idx][dim] - center[dim]) * direction[dim];
            }
            projections.emplace_back(projection, idx);
        }
        // Sort by projection value (descending)
        sort(projections.begin(), projections.end(), greater<pair<double, int>>());
        return projections;
    };

```

```
275     // Direction vector  C 1 C2
276     vector<double> dir_c1_c2(c1.size());
277     for (int dim = 0; dim < c1.size(); ++dim)
278         dir_c1_c2[dim] = c2[dim] - c1[dim];
279
280     // Direction vector  C 2 C1  (reverse of  C 1 C2 )
281     vector<double> dir_c2_c1 = dir_c1_c2;
282     for (double& val : dir_c2_c1) val *= -1;
283
284     // Get top-k projected points for each MBB
285     auto proj_mbb1 = project_points(core_mbb1, c1, dir_c1_c2);
286     auto proj_mbb2 = project_points(core_mbb2, c2, dir_c2_c1);
287
288     // Take top-k (or all if fewer than k)
289     int k1 = min(k, static_cast<int>(proj_mbb1.size()));
290     int k2 = min(k, static_cast<int>(proj_mbb2.size()));
291
292     // --- Step 4: Check pairs between top-k points ---
293     const double epsilon_sq = epsilon * epsilon;
294     for (int i = 0; i < k1; ++i) {
295         const auto& [proj1, idx1] = proj_mbb1[i];
296         const auto& p1 = points[idx1];
297         for (int j = 0; j < k2; ++j) {
298             const auto& [proj2, idx2] = proj_mbb2[j];
299             const auto& p2 = points[idx2];
300
301             double dist_sq = 0.0;
302             for (int dim = 0; dim < p1.size(); ++dim) {
303                 double diff = p1[dim] - p2[dim];
304                 dist_sq += diff * diff;
305                 if (dist_sq > epsilon_sq) break;  // Early termination
306             }
307             if (dist_sq <= epsilon_sq)
308                 return true;
309         }
310     }
311     return false;
312 }
313
314 // ==================== Build MBB Graph ====================
315 void build_mbb_graph(
316     const vector<vector<int>>& points,
317     shared_ptr<TreeNode> root,
318     double epsilon,
319     int d,
320     const unordered_set<int>& core_points)
321 {
322     mbb_graph.resize(leaf_nodes.size());
323
324     for (size_t i = 0; i < leaf_nodes.size(); ++i) {
325         mbb_graph[i].mbb = leaf_nodes[i];
326         vector<int> neighbors = find_epsilon_neighbors(leaf_nodes[i], root, epsilon, d);
327
328         for (int j : neighbors) {
329             if (has_epsilon_core_pair(leaf_nodes[i], leaf_nodes[j], points, core_points, epsilon)) {
330                 mbb_graph[i].neighbors.push_back(j);
331             }
332         }
333     }
334 }
335
336 // ==================== DFS for Connected Components ====================
337 void dfs(int node_idx, vector<bool>& visited, vector<int>& component, const vector<MBBNode>& graph) {
338     visited[node_idx] = true;
339     component.push_back(node_idx);
340
341     for (int neighbor : graph[node_idx].neighbors) {
342         if (!visited[neighbor]) {
343             dfs(neighbor, visited, component, graph);
344         }
345     }
346 }
347
348 // ==================== Get All Connected Components ====================
349 vector<vector<int>> find_connected_components(const vector<MBBNode>& graph) {
350     vector<vector<int>> components;
351     vector<bool> visited(graph.size(), false);
352
353     for (size_t i = 0; i < graph.size(); ++i) {
354         if (!visited[i]) {
```

```
355             vector<int> component;
356             dfs(i, visited, component, graph);
357             components.push_back(component);
358         }
359     }
360     return components;
361 }
362
363
364 // ==================== Assign Clusters ====================
365
366
367 vector<vector<int>> assign_clusters(
368     const vector<vector<int>>& components,
369     const vector<shared_ptr<TreeNode>>& leaf_nodes)
370 {
371     vector<vector<int>> clusters;
372
373     for (const auto& component : components) {
374         vector<int> cluster_points;
375         for (int mbb_idx : component) {
376             // Add all points from MBBs in this component
377             cluster_points.insert(
378                 cluster_points.end(),
379                 leaf_nodes[mbb_idx]->point_indices.begin(),
380                 leaf_nodes[mbb_idx]->point_indices.end()
381             );
382         }
383         clusters.push_back(cluster_points);
384     }
385     return clusters;
386 }
387
388 // ==================== Main ====================
389 int main() {
390     int n = 50, d = 3, MinPts = 3;
391     double epsilon = 10.0;
392     vector<vector<int>> points = {
393     // Cluster 0 (around [0, 0, 0])
394     {0, 1, 1}, {1, 0, 1}, {1, 1, 0}, {0, 0, 1}, {1, 1, 1}, {2, 0, 0}, {0, 2, 0}, {0, 0, 2}, {1, 2, 1}, {2, 1, 1},
395
396     // Cluster 1 (around [10, 10, 10])
397     {10, 10, 11}, {11, 10, 10}, {10, 11, 10}, {10, 10, 9}, {9, 10, 10}, {10, 9, 10}, {11, 11, 11}, {9, 9, 10}, {10,
         11, 11}, {11, 9, 10},
398
399     // Cluster 2 (around [20, 0, 0])
400     {20, 1, 0}, {21, 0, 0}, {20, 0, 1}, {19, 0, 0}, {20, -1, 0}, {19, 1, 0}, {21, -1, 1}, {20, 0, -1}, {21, 1, 0},
         {20, -1, -1},
401
402     // Cluster 3 (around [0, 20, 20])
403     {0, 21, 20}, {1, 20, 20}, {0, 20, 21}, {-1, 20, 20}, {0, 20, 19}, {-1, 21, 20}, {1, 19, 20}, {0, 19, 19}, {1, 20,
         21}, {-1, 20, 21},
404
405     // Some noise points (optional)
406     {50, 50, 50}, {51, 51, 51}, {52, 52, 52}, {60, 0, 0}, {0, 60, 0}, {0, 0, 60}, {100, 100, 100}, {101, 101, 99},
         {99, 100, 101}, {55, 55, 56}
407 };
408
409     vector<int> all_indices(n);
410     for (int i = 0; i < n; ++i) all_indices[i] = i;
411
412     // Build the tree
413     auto root = build_tree(points, all_indices, 0, d, epsilon);
414
415     // Identify core points
416     vector<int> core_pts = identify_core_points(points, root, epsilon, MinPts, d);
417     unordered_set<int> core_set(core_pts.begin(), core_pts.end());
418
419
420     // Build MBB graph
421     build_mbb_graph(points, root, epsilon, d, core_set);
422
423     // Find connected components in the MBB graph
424     vector<vector<int>> components = find_connected_components(mbb_graph);
425
426     // Assign clusters (excluding noise points)
427     vector<vector<int>> clusters = assign_clusters(components, leaf_nodes);
428
429     // Print results
430     cout << "===== Core Points =====\n";
```

```
431      for (int idx : core_pts) {
432          cout << "Point " << idx << ": ";
433          for (int v : points[idx]) cout << v << " ";
434          cout << "\n";
435      }
436
437
438      cout << "\n===== Clusters =====\n";
439      for (size_t i = 0; i < clusters.size(); ++i) {
440          cout << "Cluster " << i << ": ";
441          for (int point_idx : clusters[i]) {
442              cout << point_idx << " ";
443          }
444          cout << "\n";
445      }
446
447      return 0;
448  }
```

Listing 4.1: C++ Implementation of the Proposed DBSCAN Variant

Chapter 5

Results and Discussion

5.1 Results

To evaluate the effectiveness of the proposed Box-based DBSCAN algorithm, we compare it against the standard DBSCAN across four challenging synthetic datasets: *Jigsaw*, *Spiral*, *Concentric Circles*, and *Two Moons*. Each clustering result is visualized with cluster assignments in distinct colors and noise points in black.

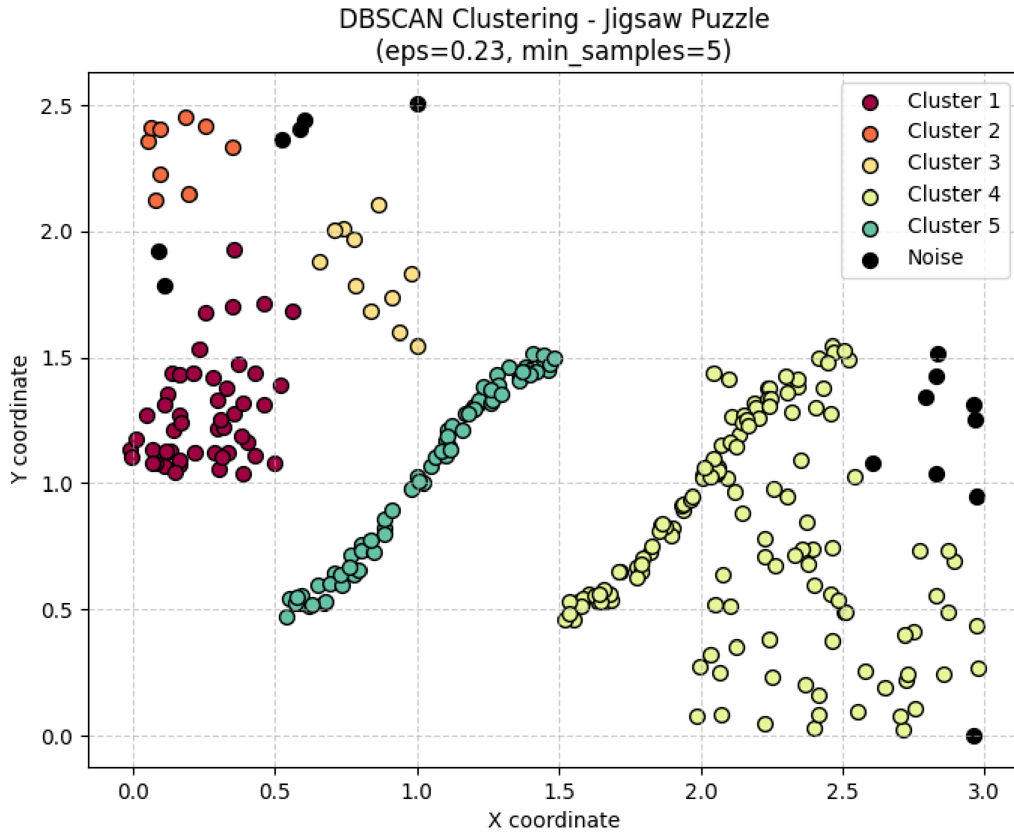

Figure 5.1: Standard DBSCAN on the Jigsaw dataset.

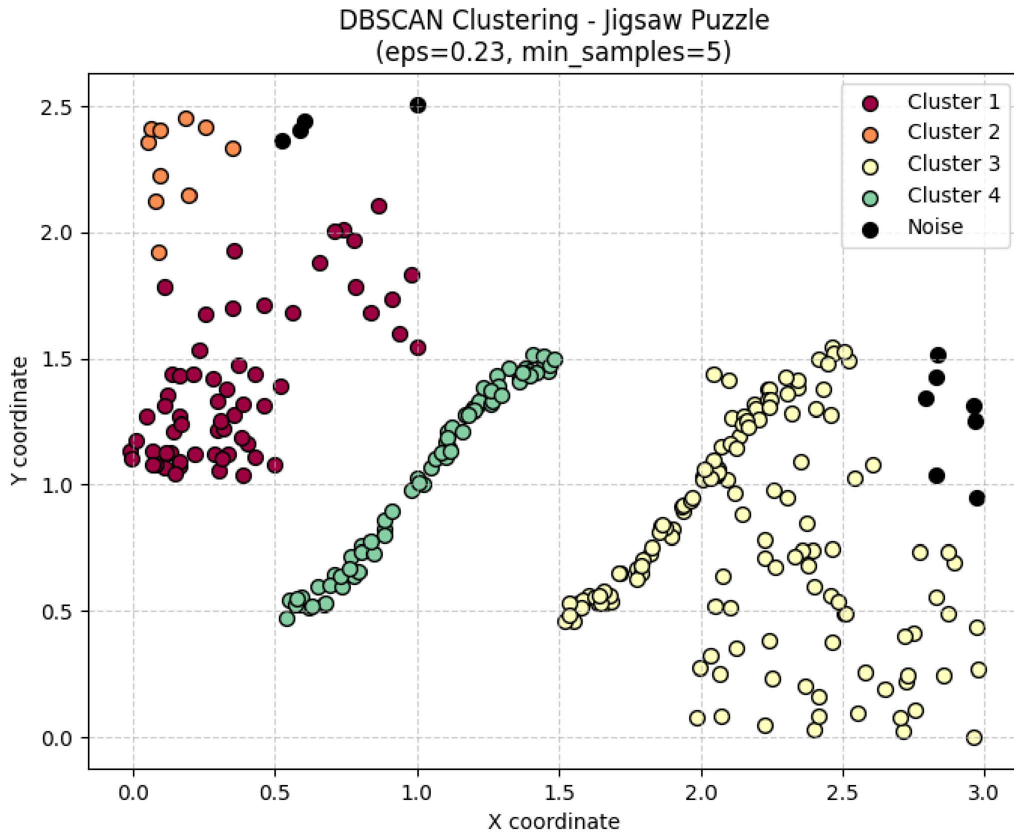

Figure 5.2: Box-based DBSCAN on Jigsaw dataset.

Figure 5.3: Comparison of clustering performance on the Jigsaw dataset.

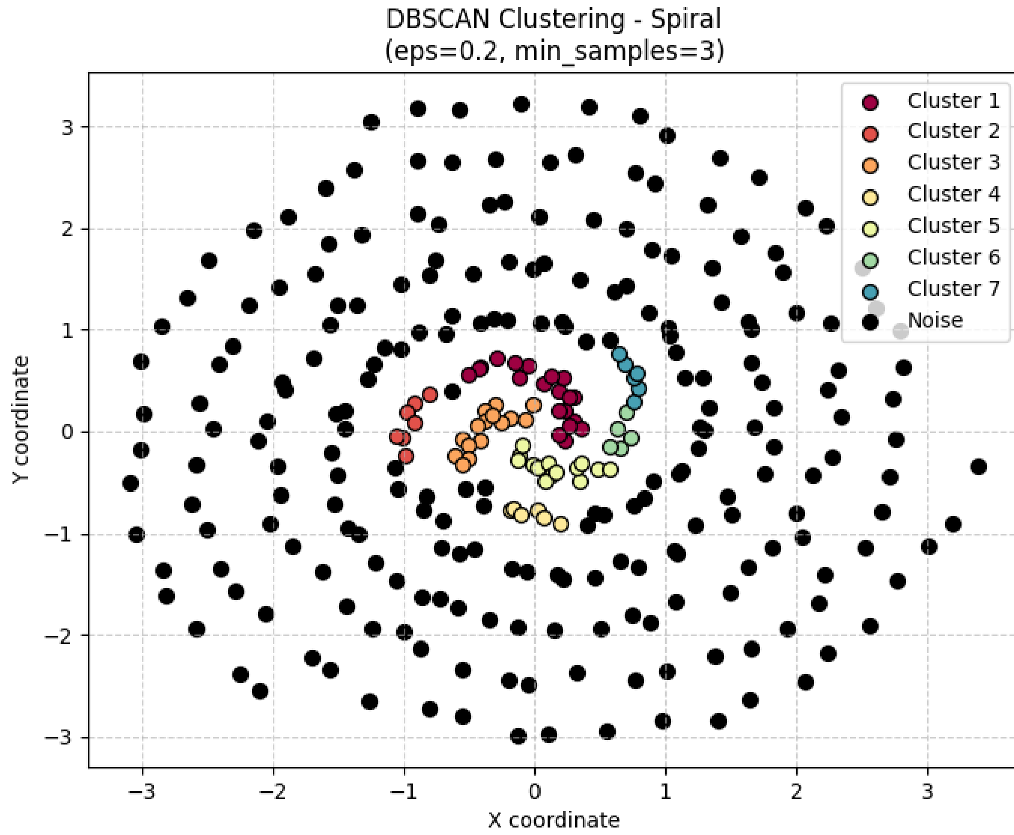

Figure 5.4: Standard DBSCAN - Spiral

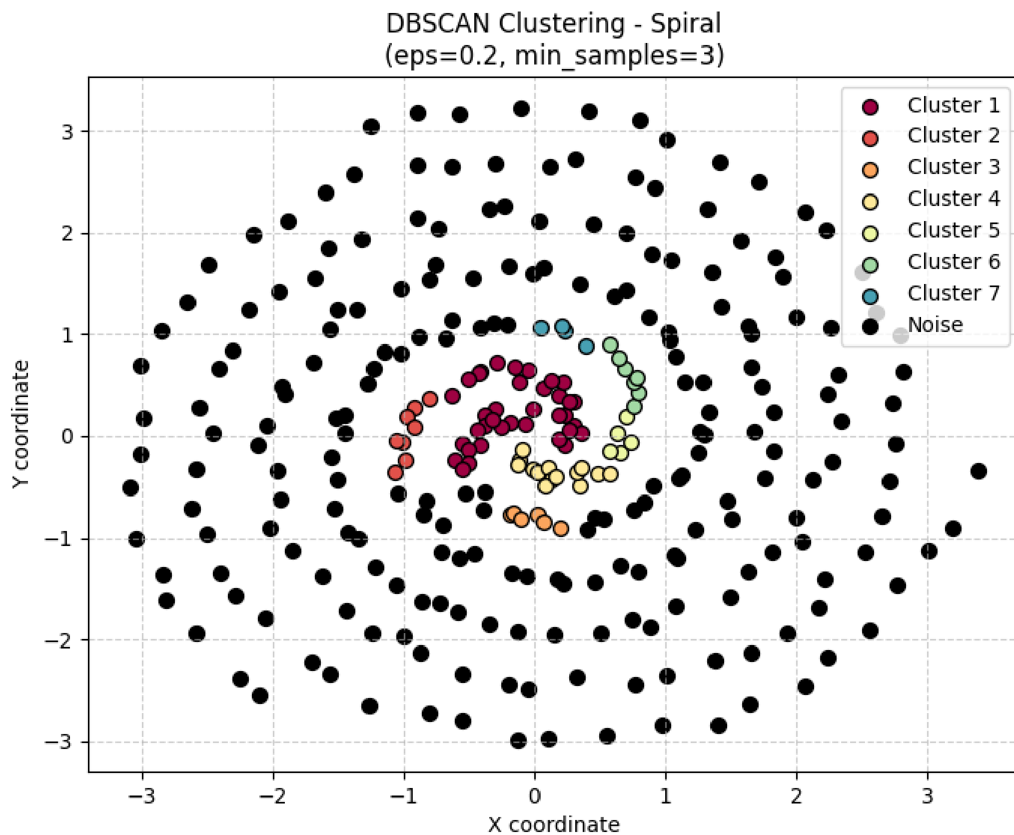

Figure 5.5: Box-based DBSCAN - Spiral

Figure 5.6: Comparison of clustering performance on the Spiral dataset.

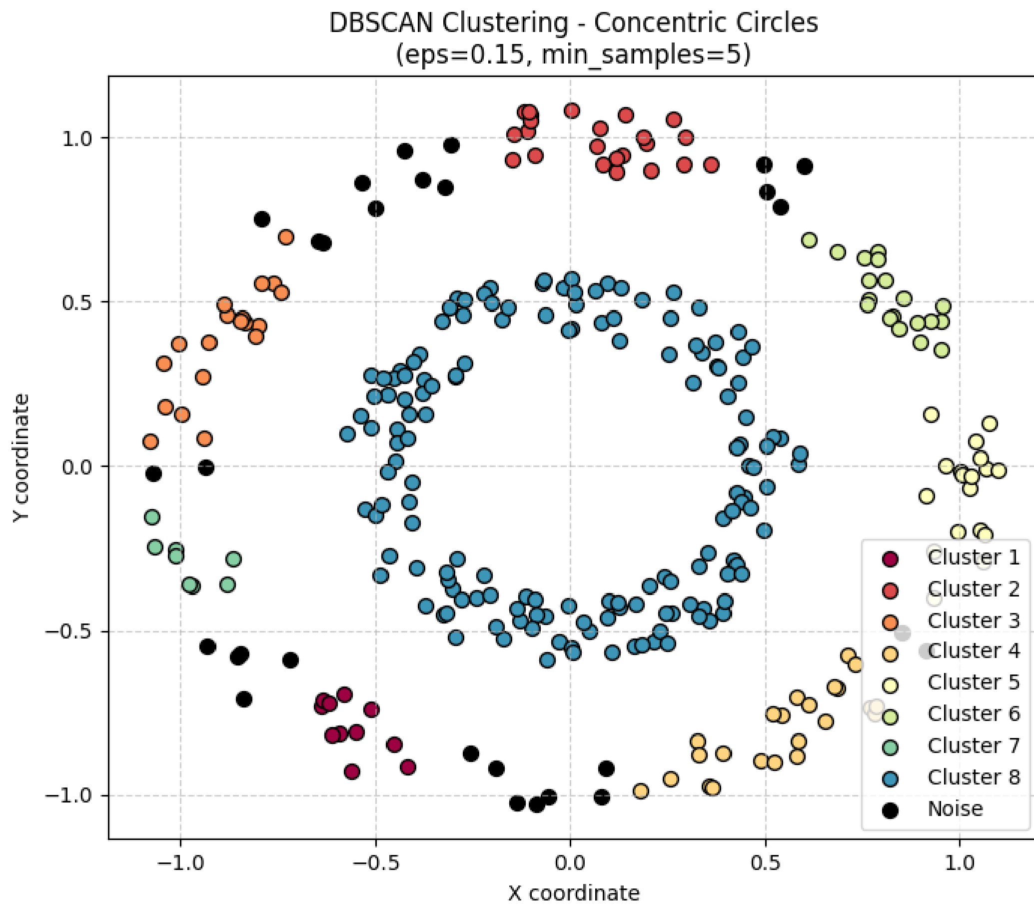

Figure 5.7: Standard DBSCAN - Concentric Circles

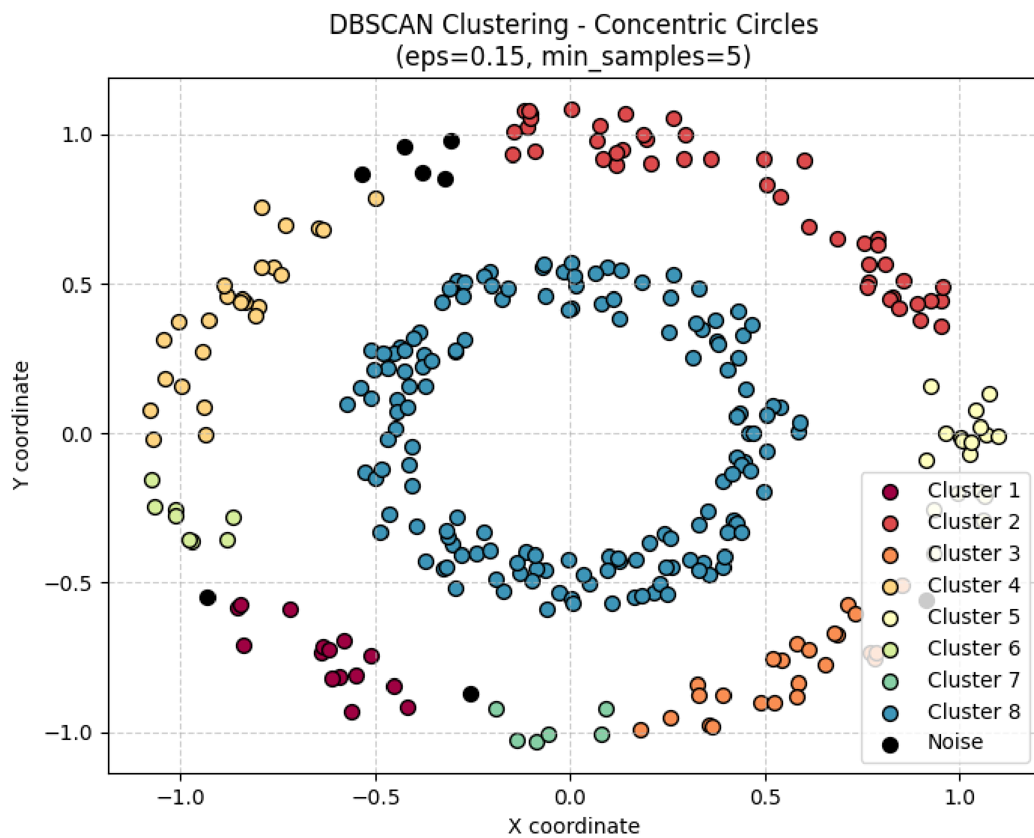

Figure 5.8: Box-based DBSCAN - Concentric Circles

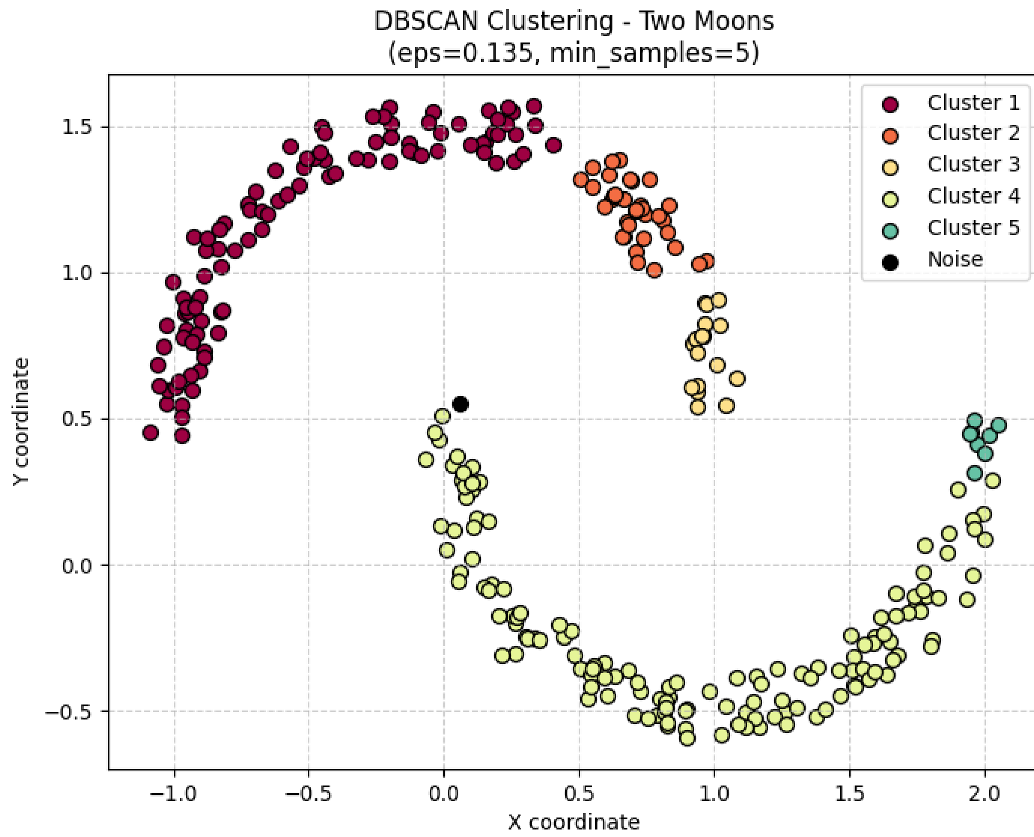

Figure 5.10: Standard DBSCAN - Two Moons

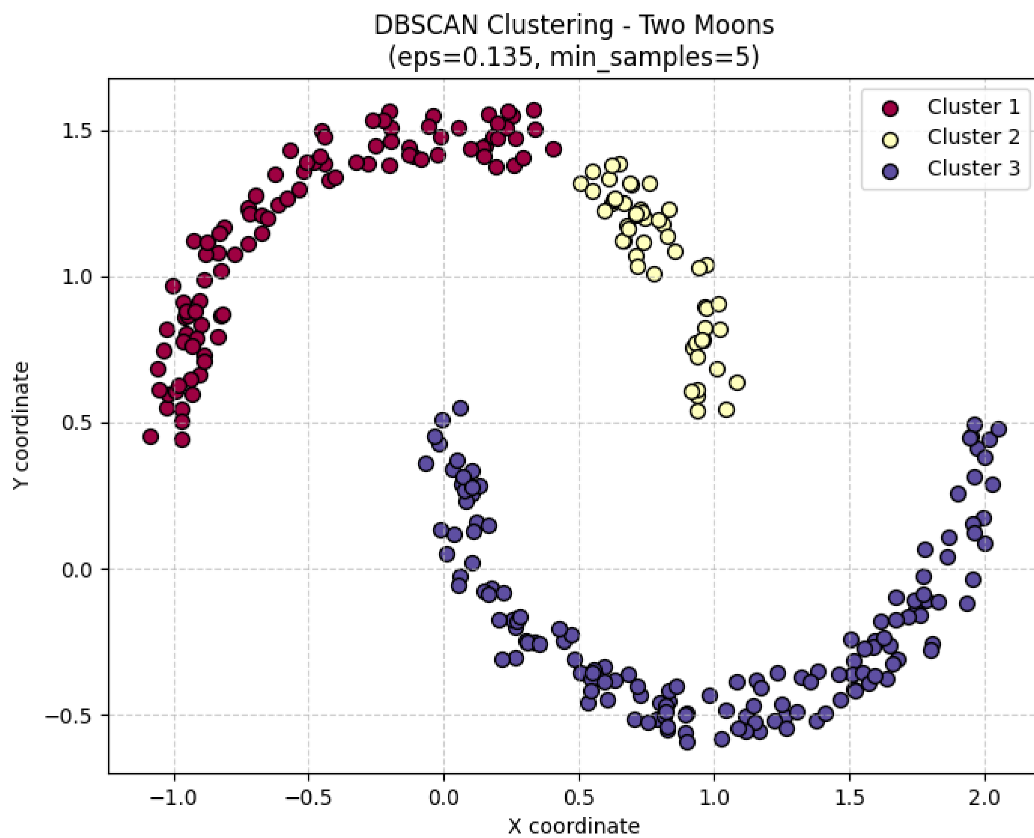

Figure 5.11: Box-based DBSCAN - Two Moons

Figure 5.12: Comparison of clustering performance on the Two moon dataset.

Observations

Across all benchmark datasets, the Box-based DBSCAN algorithm is not coming far different from the DBSCAN algorithm, indicating that it's safe to use for machine learning purposes with a high number of features.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this dissertation, we conducted a comprehensive study of the DBSCAN algorithm, beginning with a deep dive into foundational literature and extensions proposed in prior research. We analyzed the core ideas behind DBSCAN and explored the spatial data structures—such as R-trees and KD-trees—that are typically used to accelerate its computations.

Motivated by the algorithm's inherent challenges in high-dimensional or spatially complex data, we designed two new algorithms: an approximate variant of standard DBSCAN and a modified version called *Box-based DBSCAN*. Both methods aim to improve performance while maintaining clustering fidelity. The approximate DBSCAN algorithm stays faithful to the original DBSCAN definition but uses bounding boxes to optimize core point detection and neighborhood queries.

The Box-based DBSCAN introduces a slight but significant variation in how clusters are formed. Rather than relying solely on point-to-point -distance comparisons, it groups and merges axis-aligned bounding boxes based on spatial density criteria. This box-centric approach improves cluster shape sensitivity and reduces unnecessary computations, especially in geometrically structured datasets.

Through a series of experiments on benchmark datasets, we demonstrated that both proposed methods achieve comparable or improved clustering outcomes compared to standard DBSCAN, while also showing computational advantages. Overall, our contributions show that even minor modifications to the core definition of DBSCAN can lead to meaningful improvements in clustering quality and efficiency.

6.2 Limitations

While the proposed algorithms demonstrate strong empirical performance and improved geometric sensitivity, they are not without limitations:

1. Although the Box-based DBSCAN algorithm typically performs efficiently in practice, its worst-case time complexity can degrade to $O(n^2 \cdot d)$ in certain pathological configurations. For example, when points are distributed near the centers of dense grid cells with side length close to ϵ, bounding box grouping may offer little advantage over brute-force approaches.

2. The centroid-based approximation used for the Bi-Chromatic Closest Pair (BCP) test in core-point connectivity is designed for efficiency, but may produce incorrect results in adversarial cases. Specifically, if two clusters have complex shapes or uneven internal distribution, the projection-based shortcut may miss close point pairs, resulting in false negatives in cluster merging.

These limitations highlight the importance of understanding dataset characteristics when deploying the proposed method. Future work may involve mitigating these edge cases using adaptive heuristics or hybrid verification steps.

6.3 Future Scope

There are several directions in which this work can be extended:

- We aim to further investigate and address the limitations identified in this thesis, particularly the worst-case performance scenarios and the robustness of centroid-based approximations. If time permits, we plan to develop fallback mechanisms or hybrid strategies to ensure better runtime even in adversarial cases.

- Another important area for future research is the development of better strategies for automatic parameter selection in DBSCAN, especially in high-dimensional spaces. The performance of DBSCAN is highly sensitive to the choice of ϵ and `MinPts`, and heuristic or learning-based methods could provide more reliable and adaptive configurations.

Bibliography

[1] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu.
A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise.
Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), 1996.

[2] Junhao Gan, Yufei Tao.
DBSCAN Revisited: Mis-Claim, Un-Fixability, and Approximation.
Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data, 2017.

[3] Schubert, Erich, Jörg Sander, and Martin Ester.
DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN.
ACM Transactions on Database Systems (TODS), 2017.

[4] Antonin Guttman.
R-Trees: A Dynamic Index Structure for Spatial Searching.
Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, 1984.

[5] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger.
The R-Tree: An Efficient and Robust Access Method for Points and Rectangles.*
Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, 1990.