Scalable Approximate Nearest Neighbour Algorithms for Machine Learning and Data Mining

Camilla Birch Okkels

Advisors: Martin Aumüller and Arthur Zimek
Submitted: November 2025

IT UNIVERSITY OF COPENHAGEN

Abstract

In this thesis we explore different uses of techniques for *approximate nearest neighbor search* (ANN search). Finding a set of close points is necessary for many diverse tasks such as clustering and outlier detection, but finding the exact closest points with guarantee can be expensive, often quadratic in running time complexity. Considering the increasing size of data available, this is a problem that will only get more relevant as time goes on.

ANN search can be used to speed up the process of finding near neighbors at the cost of accuracy by introducing a degree of approximation.

We introduce implementations of algorithms using Locality Sensitive Hashing (LSH) and Hierarchical Navigable Small Worlds (HNSW) for ANN search for different tasks and explore the trade-offs they provide between accuracy and running time. The main contributions of the thesis are:

- We introduce an implementation of an LSH-based algorithm for approximate DBSCAN clustering. Fast algorithms for DBSCAN clustering exist in literature, but they often lack theoretical guarantees or rely on the dataset being low-dimensional to work optimally. The proposed algorithm was compared to other state-of-the-art DBSCAN algorithms. We show that while other algorithms failed in some settings for different reasons, ours was consistently either the fastest implementation or performing competitively.
- We explore the benefits and limitations of HNSW graphs for outlier detection. We evaluate direct implementations that compute the kNN and LOF scores from approximate neighborhoods as well as white-box methods. The white-box methods compute outlier-scores directly from the underlying graph and serve as an example to break new ground to make more task-aware tools based on ANN search techniques.
- We present three algorithms for approximate Single-Linkage clustering. As with DBSCAN, the issue lies in the quadratic scaling of the problem. All three utilize HNSW graphs to varying degrees of approximation. We explore their respective trade-offs in accuracy vs. running time as well as compare how they perform against exact implementations. One of the algorithms in particular explores recent methods using heap-of-searchers for incremental ANN search.

Resumé

I denne afhandling udforsker vi forskellige anvendelser af teknikker til *approksimativ nærmeste nabo-søgning* (ANN-søgning). At finde et sæt af nærliggende punkter er nødvendigt for mange forskellige opgaver såsom clustering og detektion af outliers, men at finde de præcist nærmeste punkter med garanti kan være dyrt og ofte have kvadratisk tidskompleksitet. I betragtning af den stigende mængde af data, er dette et problem, der kun bliver mere og mere relevant over tid.

ANN-søgning kan bruges til at gøre processen med at finde nærliggende naboer hurtigere på bekostning af nøjagtighed ved at indføre en grad af approksimation.

Vi introducerer implementeringer af algoritmer, der bruger Locality Sensitive Hashing (LSH) og Hierarchical Navigable Small Worlds (HNSW) til ANN-søgning til diverse opgaver og undersøger de tradeoffs, der er mellem nøjagtighed og tid. Afhandlingens bidrag er:

- Vi introducerer en implementering af en LSH-baseret algoritme til approksimativ DBSCAN-clustering. Hurtige algoritmer til DBSCAN-klyngedannelse findes i litteraturen, men de mangler ofte teoretiske garantier eller afhænger af, at datasættet er lavdimensionelt for at fungere optimalt. Den foreslåede algoritme blev sammenlignet med andre state-of-the-art DBSCAN-algoritmer. Vi viser, at mens andre algoritmer fejlede i visse scenarier af forskellige grunde, var vores enten konsekvent den hurtigste implementering eller ydede på et kompetitivt niveau.
- Vi undersøger fordelene og begrænsningerne ved HNSW-grafer til detektion af afvigere (outlier detection). Vi evaluerer direkte implementeringer, der beregner kNN- og LOF-scorer ud fra approksimative nabolag, såvel som white-box-metoder. White-box-metoderne beregner outlierscorer direkte ud fra den underliggende graf og tjener som et eksempel på, hvordan man kan bane vejen for mere opgavebevidste værktøjer baseret på teknikker til ANN-søgning.
- Vi præsenterer tre algoritmer til approksimativ Single-Linkage-clustering. Ligesom i DBSCAN ligger udfordringen i den kvadratiske skalering. Alle tre anvender HNSW-grafer til varierende grad af approksimation. Vi undersøger deres respektive trade-offs mellem nøjagtighed og tid samt sammenligner, hvordan de præsterer i forhold til eksakte implementeringer. En af algoritmerne specifikt udforsker nyere metoder, der anvender heap-of-searchers til trinvis ANN-søgning.

Acknowledgements

I have had the pleasure of having two excellent supervisors - Martin Aumüller and Arthur Zimek. They provided invaluable support throughout my PhD, giving feedback and helping in my overall growth as a researcher. I am very grateful for the role they have played in my journey.

I would also like to thank the people in the algorithms group I have interacted with on a daily basis - past and present. They created a very warm and welcoming environment. It has meant very much to me that I've had such a nice place of work with great people during these three years.

A thanks to the ITU PhD Club as well. It was a pleasure interacting with great people and getting to know fellow PhD students on the board as well as at social events we organized. It was nice to have people to talk to in the same situation as myself.

I also had the privilege of visiting professor Erich Schubert and his group at TU Dortmund in Germany. I want to thank him and his group for hosting me and providing a very warm and welcoming environment for me to work. It was a very memorable experience and a productive collaboration.

Lastly, I would like to thank all my friends and family who supported me and helped me get through the tough times during my PhD. A special thanks to my fiancé who kept me calm and focused up to several important deadlines. He played an absolutely crucial role.

Contents

1 Introduction 1
- 1.1 Approximate Nearest Neigbor Search . . . 3
- 1.2 Contributions and thesis outline . . . 5

2 On the Design of Scalable Outlier Detection Methods using ANN Graphs 9
- 2.1 Introduction . . . 10
- 2.2 Related work . . . 12
- 2.3 Preliminaries . . . 13
- 2.4 k-NN and LOF computation using HNSW . . . 14
- 2.5 Evaluation . . . 15
- 2.6 Discussion . . . 23

3 High-Dimensional Density-Based Clustering Using LSH 25
- 3.1 Introduction . . . 26
- 3.2 Related work . . . 28
- 3.3 Preliminaries . . . 30
- 3.4 Proposed algorithm . . . 33
- 3.5 Implementation Choices . . . 43
- 3.6 Experimental Evaluation . . . 44
- 3.7 Conclusion . . . 55

4 Approximate Single-Linkage Clustering Using Graph-Based Indexes 57
- 4.1 Introduction . . . 58
- 4.2 Background and Related Work . . . 60
- 4.3 Single-Link Clustering with HNSW . . . 62
- 4.4 Evaluation . . . 66
- 4.5 Conclusion . . . 71
- 4.6 Current work towards future extension . . . 73

5 Conclusion and open problems 75

Bibliography 79

Chapter 1

Introduction

The task of finding similar points to some query point also referred to as nearest neighbor search, is used in a multitude of systems such as recommendation systems[1] to google's image search retrieving similar images to the users query. There are many different ways to measure similarity between objects. Let x and y be data points (represented as vectors) with dimension dim. The definitions of some common distance measures are as follows (see figure 1.1 for a visualization):

- Manhattan distance: $d_m(x,y) = \sum_i^{dim} |x_i - y_i|$
- Euclidean distance: $d_e(x,y) = \sqrt{\sum_i^{dim} (x_i - y_i)^2}$
- Chebyshev distance: $d_{che}(x,y) = \max_i(|x_i - y_i|)$
- Cosine distance: $d_{cos}(x,y) = \cos(\theta) = \frac{x \cdot y}{\|x\|\|y\|}$

Similarity search is fundamental to many larger objectives such as clustering, classification, density estimation or outlier detection. k-Nearest Neighbors (k-NN), Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [EKSX96], Ordering Points To Identify the Clustering Structure (OPTICS) [ABKS99] are examples of widely used clustering algorithms where nearest neighbor search is used in different ways. k-NN use close points directly to cluster a new query point. In DBSCAN clusters are formed by distinguishing between dense and less dense neighborhoods - the dense areas constituting the clusters. We will go into further detail with

[1] Examples include Amazon (and other online shops) recommending similar products to online shoppers or Netflix's "more like this" feature.

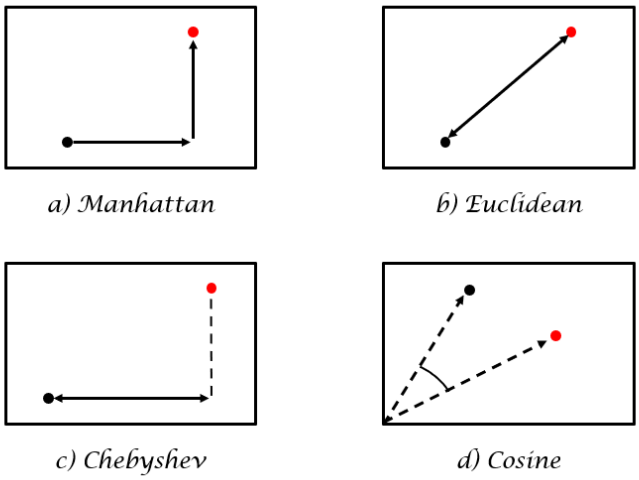


Figure 1.1: A simplistic visualization of different distance measures

DBSCAN and give a brief introduction to its hierarchical clustering variant HDBSCAN in chapter 3. OPTICS uses the reachability distance - a distance measure based on the distance to the *minPts* nearest neighbor, with *minPts* being some parameter representing the desired density threshold.

A limitation of these kinds of algorithms dependent on nearest neighbor search is that finding exact nearest neighbors can be very time-intensive. Consider the most straightforward solution to just calculate the distance from the query point to every other point in the data set. With n data points and dimension d, this would require $\mathcal{O}(nd)$ time. Repeating this for all points the time cost scales quadratically with the size of the data set ($\mathcal{O}(n^2d)$). With modern data sets only appearing to grow larger, already reaching billions of entries it quickly becomes unfeasible to have algorithms that scale quadratically with n. At such scale, even finding similar points to a single query point can take minutes if not hours.

DBSCAN is an example of an algorithm dependent on finding the neighborhoods around points, and where the simplest implementation computes all pair-wise distances between data points. There exists fast implementations, particularly for low-dimensional datasets [WGS20, GT15], but there remains a need for an efficient algorithm for very large *and* high-dimensional datasets. We propose an implementation for such an algorithm in chapter 3.

1.1 Approximate Nearest Neigbor Search

Approximate nearest neighbor search (ANN search) is a means of making algorithms more scalable at the cost of accuracy. There are a diverse range of techniques from the graph-based Hierarchical Navigable Small Worlds to the hashing-based Locality-Sensitive Hashing (LSH). Common for these techniques is they create an index structure to allow for faster query to find near points. Many ANN-based algorithms use the index structure of these ANN techniques and simply replace the exact neighborhood with the approximate one. As we demonstrate in chapter 2 and 4, these indexes and their properties can also be used more directly for the given task. To give a deeper look at the different advantages of ANN techniques, we go into more detail with LSH and HNSW.

1.1.1 Locality Sensitive Hashing

As opposed to typical hashing, where we want to minimize the chance of collisions, with LSH we want a high chance of collisions between *similar* points and a low chance between dissimilar points. More formally, we use the following definition for LSH, also found in definition 3.3.5 used in chapter 3:

Definition 1.1.1 (Locality-Sensitive Hash Family [IM98])**.** *Let* $\Pr(\chi)$ *denote the probability of an event* χ*. Given a distance function* d*, a family of hash functions* $\mathcal{H} = \{h : \mathbb{R}^d \to R\}$ *is said to be* (d_1,d_2,p_1,p_2)*-sensitive if for any pair of points* $p,q \in \mathbb{R}^d$ *the following statements are true:*

(i) *For* $d(q,p) \leq d_1$, $\Pr_{h\sim\mathcal{H}}(h(q) = h(p)) \geq p_1$.
(ii) *For* $d(q,p) \geq d_2$, $\Pr_{h\sim\mathcal{H}}(h(q) = h(p)) \leq p_2$.
(iii) $p_1 \geq p_2$ *and* $d_1 \leq d_2$.

Let $\rho = \frac{log(1/p_1)}{log(1/p_2)}$. This parameter captures the efficiency of the LSH family - a lower value means a better separation between similar and dissimilar points.

One advantage of LSH is the fact that it is not defined on any particular distance measure, it could be hamming distance, euclidean distance, cosine similarity or something else making it useful in many different domains. Furthermore, due to the clear criteria in the definition, LSH provides good theoretical bounds to be derived for ANN algorithms taking advantage of this technique. Consider a family of hash functions

$\mathcal{H}$. Suppose we define a new family $\mathcal{G}$ of hash functions g, where each function $g \in \mathcal{G}$ is build as a concatenation of k hash functions h_1,h_2...,h_k from $\mathcal{H}$. In other words $g(x) = [h_1(x),h_2(x),....,h_k(x)]$. If we then hash L times, each time using a new function g to construct a hash table. For a query, q, we iterate over all L hash tables and look at the points hashed to the same buckets as q. This simple search gives a probability of at least $1-(1-p_1^k)^L$ of finding a point within distance d_1 (if such a point exists). As we demonstrate in section 3.4.5 in chapter 3, this can be extended in specific ANN algorithms to produce bounds on quality. We also use the LSH properties to derive expected sizes on the buckets in the hash tables, allowing us to derive bounds on the running time.

The first obvious limitation of LSH is that it does not guarantee to find the exact neighborhood - accuracy is after all what is sacrificed to gain scalability. Another limitation of LSH is the fact that new parameters are introduced. In the previous simple example, the parameters k and L would need to be determined. The performance of an LSH-based ANN algorithm is very dependent on these parameters, so they must be optimized to gain the highest effectiveness.

1.1.2 Hierarchical Navigable Small Worlds Graphs

To understand HNSW graphs we can go back to two techniques that heavily inspired them: probability skip lists and navigable small worlds graphs.

Probability skip lists (see [Pug90]) were introduced as a fast method for searching through an array while also allowing quick insertion. They have a layered structure of linked lists with higher layers having large jumps by links '*skipping*' a number of elements. Going down the layers, these jumps become smaller and smaller by decreasing the number of elements skipped by each link. A query is carried out by moving the linked lists in the layers, and if the value searched for is smaller than the current node, we have overshot and move to the previous nodes position in the next layer below.

Navigable small world (NSW) graphs take the concept of a proximity graph, but are build in a way so the nodes have both long-range and short range edges. This makes NSW graphs very traversable, reducing search times to (poly)logarithmic complexity [MPLK14].

HNSW combines the ideas of probability skip lists and NSW graphs and builds a hierarchy of NSW graphs with higher layers being more sparse and with longer edges [MY20]. Important parameters for construction of the graph include:

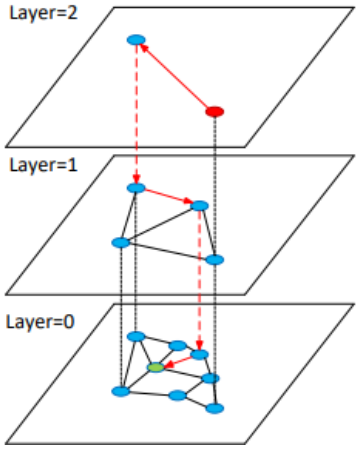


Figure 1.2: A visualization of the HSNW graph. Illustration from paper by Malkov et al. [MY20].

- $p[level]$: This is the probability that a point will be inserted at a given level. Will be low for the top layers and increase as you move down the hierarchy.
- M: This parameter controls the number of edges for a given node, i.e. the degree of the nodes in the HNSW graph.
- efC: This is the beam width during the construction of the HNSW graph. It controls the number of candidate nearest neighbors considered when inserting points in the graph.

For search queries, another parameter is relevant:

- efS: The maximum number of candidate neighbors considered at a given time when traversing through the graph during search.

HNSW allows for very fast search, and it can quickly create an index of the k-nearest neighbors of each point. However, at least the way it looks in the current landscape, HNSW is not supported by the same theoretical guarantees as for example LSH. The performance of ANN algorithms using HNSW graphs for fast approximate similarity search is primarily backed up by experiments.

1.2 Contributions and thesis outline

In the previous sections we have discussed some methods for approximate nearest neighbor search and some of their strengths and weaknesses. In

this thesis we will introduce implementations of algorithms for outlier detection, DBSCAN clustering and Single-Linkage hierarchical clustering. We will outline how these algorithms take advantage of ANN search and showcase how ANN search can provide more scalable solutions in different settings while still retaining accuracy. We will compare the performance in running time and accuracy of the approximate algorithms against the existing baselines. More specifically, the contributions of the thesis are the following results:

- Chapter 2: Camilla Birch Okkels, Martin Aumüller, and Arthur Zimek. On the design of scalable outlier detection methods using approximate nearest neighbor graphs [OAZ24]. In SISAP 2024, pages 170–184.
- Chapter 3: Camilla Birch Okkels, Martin Aumüller, Viktor Bello Thomsen, and Arthur Zimek. High-dimensional density-based clustering using locality-sensitive hashing [OATZ25]. EDBT 2025, pages 694–706.
- Chapter 4: Camilla Birch Okkels, Erik Thordsen, Martin Aumüller, Arthur Zimek, and Erich Schubert. Approximate single-linkage clustering using graph-based indexes: MST-based approaches and incremental searchers [OTA+26]. In SISAP 2025, pages 233–247 (best paper award nominee).

These papers tackle different tasks, but common for them all is they rely on some variant of similarity search and encounter a scalability problem if the exact neighborhood is required. Our proposals explore the benefits specific ANN approaches can give in these various settings and their potential limitations.

Chapter 2 - On the design of scalable outlier detection methods using approximate nearest neighbor graphs

Outlier detection is an essential tool for data analysis, but similarity search poses a scalability bottleneck for methods using exact nearest neighbors. In this chapter we introduce methods for outlier detection using HNSW graphs for fast similarity search and use kNN and Local Outlier Factor (LOF) as baseline methods. We evaluate methods that compute the kNN and LOF scores directly from the approximate neighborhood - referred

to as black-box methods. We also design white-box methods that compute outlier scores from the HNSW graph itself. AUCROC[2] scores were computed as quality measures on a wide range of datasets. The quality and running times of the approximate methods were then compared to the baselines to evaluate the performance. A range of values for the *efC* and *M* parameters of the HNSW graphs were tested in experiments. This allowed us to detect the impact the individual parameters had on the quality and to compute correlations between the recall and quality. With these results we open new ground for designing more task-aware algorithms taking advantage of techniques in ANN search.

Chapter 3 - High-dimensional density-based clustering using locality-sensitive hashing

In this chapter we explore the use of a multi-level LSH data structure to create a scalable implementation of an approximate DBSCAN algorithm. There are examples of different variants of fast algorithms for DBSCAN clustering. Examples include the implementation by Gan and Tao [GT15] which gives good theoretical bounds, but scales *exponentially* with d, and ultimately did not result in improvements in practice (see the paper by Schubert et al. [SSE+17]). Another example is the LSH-based approach by Keramatian et al. [KGPT22], which lacks the theoretical bounds on running time and accuracy and therefore does not escape the quadratic running time for some parameter choices of DBSCAN. The core contribution of the paper was to fill this gap and introduce an algorithm that has both the theoretical bounds on accuracy and good performance supported empirically.

Chapter 4 - Approximate single-linkage clustering using graph-based indexes: MST-based approaches and incremental searchers

Moving from flat clustering in chapter 3, this chapter introduces three different algorithms for the hierarchical Single-Linkage clustering. All were implemented with HNSW graphs, but to varying degree of approximation. Single-Linkage clustering is equivalent to finding the edges in the minimum spanning tree (MST) of the complete graph on the dataset with edge weights w representing the distance between pairs of points. Considering this, two of the algorithms are MST-based, the MST being imposed

[2]Area Under the ROC (Receiver Operating Characteristic) Curve.

on the HNSW graph. The third one uses a heap-of-searchers to incrementally search through the graph and merge the global smallest edge in the searchers at a given step. The paper explores the performance of these three graph-based algorithms and compares running time and accuracy to exact baselines; the industry-standard implementation in the SciPy library and the Vantage-Point tree based implementation by Schubert [Sch24].

Chapter 2

On the Design of Scalable Outlier Detection Methods using Approximate Nearest Neighbor Graphs

First published in: Similarity Search and Applications.
SISAP 2024. Lecture Notes in Computer Science, vol 15268, pp 170-184 by Springer Nature [OAZ24]
Reproduced with permission from Springer Nature
Joint work with: Martin Aumüller and Arthur Zimek

Abstract

Efficient and reliable methods for distinguishing outliers in data remain crucial for data analysis. Although supervised methods based on neural networks have gained recent traction, unsupervised methods such as the kNN outlier method and local outlier factor (LOF) remain state-of-the-art solutions according to different standardized benchmarks. Unfortunately, exact outlier detection through nearest neighbor search queries provides a scalability bottleneck for the high-dimensional, big datasets that are routinely analyzed in data science applications. This paper explores benefits and limitations of using approximate nearest neighbor search via Hierarchical Navigable Small World graphs (HNSW) to overcome this scalability barrier. We evaluate direct implementations that compute the kNN and LOF score from approximate neighborhoods and show the robustness of the outlier detection even in settings where the approximation is far away from the exact neighborhoods. Furthermore, we design white-box methods that compute the outlier scores directly from the underlying graph. These methods show much more variability in the quality of the outlier scores and open new ground for the development of task-aware tools based on approximate nearest neighbor search techniques.

2.1 Introduction

At the core of many outlier detection methods is some notion of density and the intuition that outliers are observations appearing in areas of relatively low density. Most observations are clustered, i.e., in areas of relatively high density around their location. In this fundamental intuition, several aspects are defined in different ways when implemented in different methods. Firstly there would be some particular way of estimating the density (rarely as a proper kernel density estimate, such as in `KDEOS` [SZK14a], but usually using some assessment of ε-range queries or of k-nearest neighbors and their distances, which can be seen as proxies for a proper density estimation [ZF18]). Secondly some particular way of deciding if the density is low or high, and finally some particular way to compare local density estimates, i.e., the aspect of the density being *relatively* low or high. The last aspect is reflected in the notion of "locality" in outlier detection methods [SZK14b]. Since the notion of outlierness itself is (perhaps necessarily) vague and inexact [ZF18], and commonly used techniques for density estimation and their comparison are inexact by nature, it has been argued that the exact neighborhood might also be dispensable, if we only have a good approximation [KSZ17a, ZF18]. We follow this line of thought here and consider a more recent alternative [MY20] to the approximate methods explored in related work. The benefit of using approximate nearest neighbor search is obvious in terms of efficiency, addressing a bottleneck in many data mining algorithms, and particularly outlier detection algorithms.

Given the importance of outlier detection in large data collections, the guiding research question is: *How can recent advances in approximate nearest neighbor search improve the scalability of classical outlier detection methods without sacrificing the quality of the result?* In this paper, we focus on designing scalable variants for the well-known unsupervised outlier detectors kNN and LOF (local outlier factors) [BKNS00]. The basis of this investigation will be HNSW (Hierarchical Navigable Small World Graph) by Malkov and Yashunin [MY20], which stood the test of time, for example testified by an invited talk at SISAP 2023. Despite its weak worst-case guarantees [IX23], it is the basis of most implementations in the ann-benchmarks [ABF20] project, and remains one of the best performing implementations even for approximate nearest neighbor search on billion-scale datasets [MSB$^+$24]. Since both the kNN and LOF method rely on information about the k nearest neighbors of each data point, it is no surprise that a method with close to exact results will provide reliable outlier

scores. However, while fast search time is paramount for approximate nearest neighbor search, the time to build the index is part of the running time considerations for downstream tasks such as outlier detection. Thus, optimal hyper-parameters for fast approximate searches might render the approach noncompetitive due to the index building time, and a closer investigation of the quality-performance trade-off is warranted.

While ANN methods can often be used as a "black-box" by replacing the exact nearest neighbor search routines in data mining applications, not much focus has been put on investigating if one can directly use the indexing data structure. Since popular outlier detection methods such as `ODIN` work directly on the kNN graph by using the in-degree as outlier score, a direct use of the HNSW graph could potentially further improve the scalability. However, the nature of approximate nearest neighbor graphs poses many challenges: first, these approaches use undirected edges and impose a strict degree bound on each node; second, a key insight of approximate nearest neighbor graphs is a pruning step which makes sure that direct edges between nodes are avoided if they are deemed unnecessary, for example because a node is still reachable through another neighbor. Both techniques make traditional approaches seem difficult to apply: There is no diversity in in-degree, rendering approaches like `ODIN` meaningless, and connections to near neighors are explicitly pruned away, making the computation of kNN and LOF scores unreliable.

After discussing related work in Section 2.2 and providing a summary of the necessary preliminaries in Section 2.3, we propose the "black-box" and "white-box" outlier detection methods in Section 2.4. The white-box method will use the sum of distances to neighbors as outlier score (imitating the kNN method) and contrast these scores to the score of their neighbors (imitating LOF). The evaluation of these methods is presented in Section 2.5. Our main findings are that HNSW-based kNN and LOF score computation provides a reliable and scalable alternative to exact alternatives. Despite the remarks made above, the white-box method provides competitive alternatives which further improve scalability. Our results indicate that, for some datasets, the inexactness of the approximation *helps* in achieving better results, adding further evidence that inexactness can improve downstream applications [KSZ17a].

2.2 Related work

Various methods for outlier detection are based on assessing the k nearest neighbors and their distances, resulting in some (proxy of a) local density estimate (or local model), which can be compared to other local models [ZF18]. Classic examples are the kNN outlier score (simply the distance to the kth nearest neighbor) [RRS00] or the aggregation (sum, average) of the distances to the k nearest neighbors [AP05]. The concept of locality (i.e., the local model of density is not compared to all other models, but to models in the vicinity) has been introduced with LOF [BKNS00], which inspired numerous variations [SZK14b]. Despite these many variants, the classic methods (kNN and LOF) can still be considered state-of-the-art [CZS+16].

The information of k nearest neighbors for each point can be interpreted as a directed graph (with the distances as edge weight), and has been used as such for outlier detection explicitly [HKF04] (`ODIN`) or implicitly [RNI15], yet not for approximate neighbor search. Approximate methods for outlier detection typically are filter-refinement methods targeting the top-n outliers only, or methods based on approximate nearest neighbor search (these two categories are not strictly mutually exclusive). Targeting the top-n outliers only allows to use approximate solutions in a first step (filter), and keeping only those candidates that have a chance to become one of the top-n outliers, when their (approximate) score is refined to the exact score [OTW+10, JTH01, BS03, KGKB03, NG09, AF09].

While an advantage of using approximation techniques for filtering is the efficiency w.r.t. the number of instances in a dataset, the use of approximate neighborhood search additionally considers challenges of high dimensionality [ZSK12]. Several methods based on approximate neighborhood search are employing techniques based on random projections, for example, using LSH [WPT11], LSH in combination with isolation forests [ZDH+17], or projection-indexed nearest neighbors [dCH12, BVBB21]. Other studies explored the use of space-filling curves [SZK15, KSZ17a] as well as LSH and NN-descent in combination with an ensemble approach over the approximate results [KSZ17a].

The works by Campos et al. [CZS+16] and Han et al. [HHH+22] provide the basis for evaluating outlier detection methods. The former provides a large set of well-curated datasets (with and without duplicates and with different outlier ratios) and detailed benchmarking results, while the later includes state-of-the-art deep learning-based methods and has an easy to integrate pipeline.

2.3 Preliminaries

2.3.1 Outlier Detection

Let $\mathcal{X}$ be a metric space equipped with distance function d. Given a dataset $S \subseteq \mathcal{X}$, a data point $x \in S$, and an integer $k \geq 1$, we let $k\text{NN}(x)$ be the set of the k closest points to x in S.

The computational problem *Outlier Detection* is to give each point $x \in S$ a score $score(x) \in \mathbb{R}$ that reflects its "outlierness." Given a threshold $\tau \in \mathbb{R}$, all points with a score of at least τ are considered the outliers, all other points are considered inliers.

The two most classic and popular outlier detection methods are the kNN outlier detection method (or kNN for short) and the local outlier factor (LOF). kNN [RRS00] uses the distance to the k-th nearest neighbor as outlier score (or other variants, such as the mean of distances). This can be seen as a local density estimate, and the most prominent outliers are those with smallest local density. The Local Outlier Factor (LOF) [BKNS00] uses the kNN sets to calculate a "local reachability density" (lrd) for a data point $x \in S$, a particular form of a local density estimate. The local outlier factor for $x \in S$ is determined by comparing the lrd of a point x to the lrds of its k nearest neighbors, thus considering local differences whereas kNN gives a global perspective [SZK14b]. If $\text{LOF}_k(x)$ produces a high value (compared to the other data points in S), then x is an area of lower density compared to its neighbors and is therefore considered to be an outlier [BKNS00]. The time complexity of LOF is $O(k)$ operations per data point for a total of $O(kN)$. It requires that for each point $x \in S$, the set $k\text{NN}(x)$ is known, which requires $O(N^2)$ distances computations in the worst-case in high-dimensional spaces.

2.3.2 Graph-based approximate nearest neighbor search

Given a dataset $S \subseteq \mathbb{R}^d$ and parameters M,ef, the goal is to build a graph $G = (V,E)$, where each point is represented by a vertex and edges exist between a point and a "diverse" set of at most M close points. Let us assume that such a graph G is given. To find the nearest neighbors of a query point q, HNSW [MY20] uses a hierarchy of graphs to find a good entry point into the bottom-layer graph that indexes all points. Given such a start point, carry out a greedy hill climbing. In each round, consider the currently closest point to the query not considered before. Inspect the neighborhood and compute the distances to the query point. After each round, trim the

Algorithm 1: BLACKBOX-HNSW(S,k,M,efC,efS)

```
1  kNN ← {}, lrd_k ← {}, lof_k ← {} kNN_score_k ← {}
       // empty dictionary to keep track of kNN score
2  I ← HNSW-BUILD(S,M,efC)
       // build an HNSW-based index I
3  foreach p ∈ S do
4  |    kNN[p] ← I.search(p,k,efS)
5  foreach p ∈ S do
6  |    Compute lrd_k[p]
7  foreach p ∈ S do
8  |    Compute lof_k[p]
9  |    kNN_score_k[p] ← distance to k-th NN in kNN[p]
10 return (kNN_score_k,lof_k)
```

list of current closest points (inspected and non-inspected) to ef, which is usually called the beam width. Terminate if all points in the list have been considered. (Note that this is not a bound on the number of distance computations, since considered points might be trimmed.) To build the graph, order all the points and insert them one-by-one using the search algorithm¸ often with a smaller beam width than used for the queries. From the points inspected in this search, a pruned set of M points is chosen as neighbors of the inserted point (pruning might be necessary for its neighbors if the degree bound M is not met). There exist many other graph-based indexes that change details of this construction [IM18, SDS+19].

2.4 k-NN and LOF computation using HNSW

2.4.1 Computing kNN and LOF using HNSW as blackbox

Algorithm 1 presents pseudocode for our approach to compute kNN and LOF using HNSW as a black-box. In this construction, the user has to pick the k parameter for kNN and LOF. To build the HNSW index, two additional parameters M (the degree bound) and efC (the beam width during the construction search) are required. To find the (approximate) nearest neighbors, the user has to specify the beam width efS used during search. The running time of Algorithm 1 consists of building the index, search for each point for its k nearest neighbors, and—in the case of LOF—to compute the LOF scores from the kNN sets.

Algorithm 2: WHITEBOX-HNSW(S,M,*efC*)

score $\leftarrow \{\}$ `// empty dictionary to keep track of outlier score.`
contrast_score $\leftarrow \{\}$ `// empty dictionary to keep track of outlier score contrasted with neighborhood.`
$(V,E) \leftarrow$ HNSW-BUILD(S,M,*efC*) `// store groundlayer graph constructed by HNSW. Associate node` $v \in V$ `with its data point` $p \in S$.
foreach $v \in V$ **do**
    score$[v] \leftarrow \frac{1}{\deg(v)} \sum_{(v,w)\in E} \text{dist}(v,w)$.
foreach $v \in V$ **do**
    contrast_score$[v] \leftarrow \frac{1}{\text{score[v]}\deg(v)} \sum_{(v,w)\in E} \text{score}[w]$.
return (score,contrast_score)

2.4.2 Computing *k*NN and LOF using HNSW as whitebox

Algorithm 2 presents our approach to directly compute *k*NN and LOF-like scores from the node neighborhoods in the approximate nearest neighbor graph. `GANNOD` (Graph ANN-based outlier detection) first builds a standard HNSW index, but only requires access to the ground layer graph. From the groundlayer graph (V,E) of the HNSW index, we compute an outlier score as follows: For each node $v \in V$, first the average distance to its neighbors is computed. This is the first outlier score that we assign to each point, similar to *k*NN. The *contrasted outlier score* of v is the ratio of the average score of its neighbors to its own score, similar to LOF.

As in Algorithm 1, the first step is to build an HNSW index for parameter choices M and *efC*. From this graph, we compute outlier scores directly and look at at most $O(M|V|)$ edges, which renders this approach much more efficient than Algorithm 1.

As motivated in the introduction, the design goals of approximate nearest neighbor graphs for high-dimensional data are typically not aligned with keeping the exact neighborhood information that the exact computation of *k*NN and LOF scores requires.

2.5 Evaluation

Baselines. We compare the HNSW-based outlier detection methods (Algorithm 1, "blackbox (BB)", and Algorithm 2, "whitebox (WB)") to

the PyOD baseline [ZNL19] for computing kNN and LOF. As another baseline, we implemented `ODIN` [HKF04] which builds the (exact) k-NN graph and uses the in-degree as outlier score. All methods use the popular implementation of `NearestNeighbor` provided with sklearn to speed up the neighborhood computation, in particular for low-dimensional datasets using tree-based indexes.[1]

Implementation notes. Our implementation of the white-box method builds upon `hnswlib`[2] and is implemented in C++ with a Python wrapper to run through the experimental pipeline. We use the same Python-based numpy code as sklearn's LOF implementation[3] to compute LOF scores for the blackbox. The implementation is available at: `https://github.com/CamillaOkkels/ANN-outlier-detection.git`.

Evaluation setup. All experiments are carried out on an Intel Xeon E5-2690v4 (2.60 GHz) with 512GB RAM using Ubuntu 20.04.6 LTS. The experimental pipeline is written in Python and uses some functionality provided by the ADBench outlier detection framework [HHH+22].

In the construction of the HNSW for the blackbox methods, *efC* was fixed at 100. We let *efS* take values [8,16,32,48,64,128], and the value of M was set to [8,16,32,48]. All combinations of values of M and *efS* were inspected in the evaluation. For the whitebox method we let *efC* take values [10,50,100,200] and M the same values as in the blackbox evaluation. Again all combinations of *efC* and M were inspected. Each experiment was run 10 times, and we report on the mean over these runs because some parameter choices on some datasets had a high variance in the AUCROC scores produced.

Datasets. Table 2.1 details the datasets used for the experimental evaluation. Several of the datasets (ALOI through SPEECH) are from the ADBench repository [HHH+22], which we refer to as small- to medium-sized datasets. The datasets selected had the highest value of $n \cdot d$ with n being the number of points and d being the dimension. Additionally we have some million-scale datasets from the Kitsune project [MDES18, BVBB21]. Each dataset comes with ground truth labels of inliers/outliers.

Performance and Quality Metrics. As quality metric, we use the Area Under the ROC Curve (AUCROC) against the ground truth labels. The ROC curve

[1] `https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestNeighbors.html`

[2] `https://github.com/nmslib/hnswlib`

[3] `https://github.com/scikit-learn/scikit-learn/blob/1.5.1/sklearn/neighbors/_lof.py`

Table 2.1: Overview of the datasets.

Dataset	#points (n)	dim. (d)	#Outliers
ALOI	49 534	27	1 508
BACKDOOR	95 329	196	2 329
CAMPAIGN	41 188	62	4 640
CELEBA	202 599	39	4 547
CENSUS	299 285	500	18 568
COVER	286 048	10	2 747
FRAUD	284 807	29	492
HTTP	567 498	3	2 211
INTERNETADS	1 966	1 555	368
MNIST	7 603	100	700
MUSK	3 062	166	97
SPEECH	3 686	400	61
Kitsune SSDP flood	4 077 263	115	1 439 603
Kitsune Mirai	764 136	116	642 516
Kitsune Active Wiretap	2 278 688	115	923 215

Table 2.2: Quality of Approximate Neighborhood of HNSW graph.

Dataset	M	recall@10 whitebox	recall@100 whitebox	recall100@100 blackbox
ALOI	4	0.38	0.86	0.99
	8	0.34	0.80	1.00
	20	0.34	0.79	1.00
	40	0.36	0.78	1.00
BACKDOOR	4	0.40	0.71	0.96
	8	0.37	0.68	0.99
	20	0.37	0.67	0.99
	40	0.39	0.72	0.99
MUSK	4	0.47	0.84	0.97
	8	0.42	0.81	1.00
	20	0.42	0.78	1.00
	40	0.34	0.66	1.00

plots the performance of the outlier detector at all thresholds with True Positive Rate on the y-axis and False Positive Rate on the x-axis. The AUCROC being the area under the ROC curve will range from 0 (for a model ranking outliers consistently after inliers) to 1 (for a model correctly ranking outliers before inliers), with 0.5 being the expected value for a random ranking. As performance metric, we record the time it takes to compute the outlier score for a given dataset and parameter configuration. For the HNSW-based approaches, this includes building the index and either carrying out batched searches (Algorithm 1) or traversing the graph (Algorithm 2).

2.5.1 Results on small to medium-sized datasets

Quality of approximate neighborhood. Table 2.2 reports on the quality of the approximate neighborhood. For each dataset and M value, we report

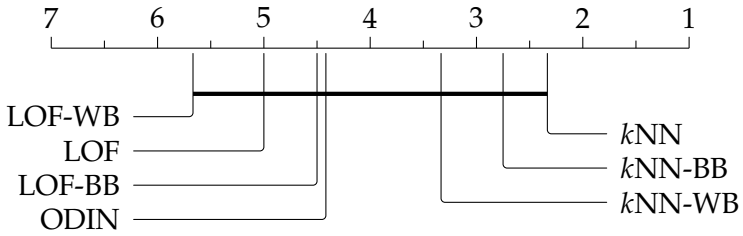


Figure 2.1: Critical difference plot: the methods are shown with their average rank over all datasets. Methods within the critical difference (i.e., connected by a line) do not show quality differences that are statistically significant.

on the highest recall achieved by the neighborhoods that form the basis of Algorithm 1 and Algorithm 2. For the whitebox method, we measure the fraction of the 10-NN and 100-NN which have a direct connection to the point in the graph (are connected by an edge), normalized by the degree of the node. For the blackbox method, we report the fraction of true nearest neighbors for $k = 100$. We can observe from the results that the blackbox variant achieves close to perfect recall no matter which M parameter is used. The results are very different for the whitebox method. Over the datasets, we notice that each node is only connected to very few of its 10 nearest neighbors. Most of the connections are to nodes that are among the 100 nearest neighbors, but there is a large diversity between the different datasets. Thus, the approximate neighborhood of a neighbor includes a diverse set of points, showing the effectiveness of the pruning rules.

Global comparison between methods. Figure 2.1 presents a ranking of the proposed methods across all datasets. We rank the achieved quality of each method averaged over all datasets mentioned in Table 2.1. For each method we first took the average AUCROC score over the 10 runs for each parameter setting. We then took the average AUCROC score achieved out of all parameter choices on a dataset and compared it to the score of the other methods to get the rankings. The plot indicates that global outlier detection methods rank best on average, indicating that the datasets mostly contain global outliers. However, we observe that the ranking differences are statistically insignificant, supporting the claim that the quality of the proposed methods is comparable to their baseline counterparts, kNN and LOF.

Quality evaluation of the individual methods. Figure 2.2 illustrates the difference in AUCROC scores of the kNN-BB, LOF-BB, kNN-WB

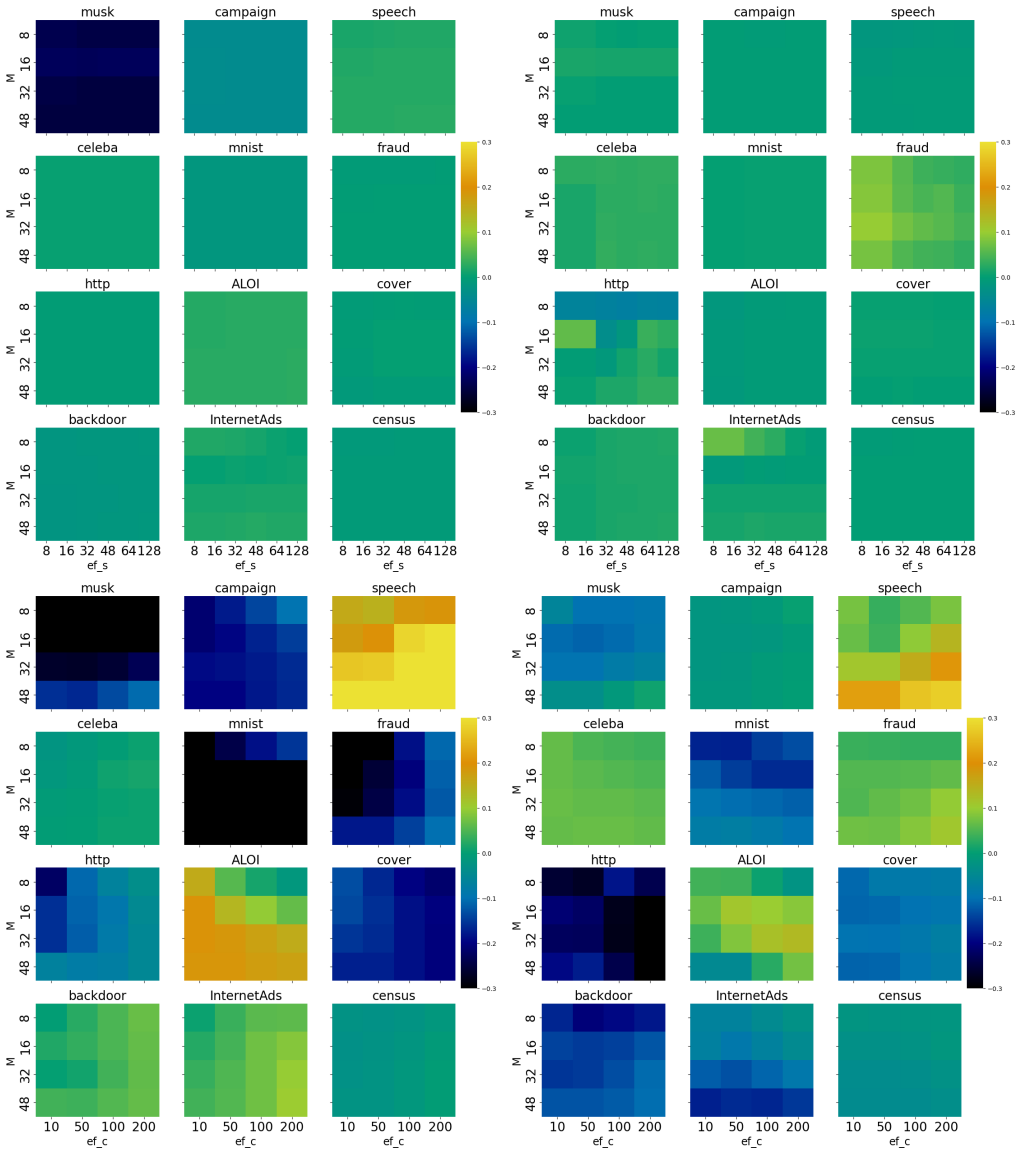


Figure 2.2: Heatmap of difference in AUCROC scores compared to baseline of the blackbox kNN (top left), blackbox LOF (top right), whitebox kNN (bottom left), and whitebox LOF (bottom right) method as a function of parameters efS (blackbox)/ef (whitebox) and M (y-axis).

and LOF-WB implementations respectively. The methods are evaluated against comparable methods, i.e. kNN-BB, kNN-WB against kNN and LOF-BB, LOF-WB against LOF. A score above 0 indicates that the proposed method provides better AUCROC than its baseline. The results shown

Table 2.3: Speedup over PyOD's *k*NN method on small to medium-sized datasets. Values above 1 indicate a speedup, values below 1 a slowdown.

Dataset	Time	kNN	LOF	ODIN	kNN-WB	LOF-WB	kNN-BB	LOF-BB
ALOI	3.4s	1.0	0.9	0.7	13.2	13.2	7.0	7.0
BACKDOOR	7.7s	1.0	1.2	0.8	9.4	9.5	6.6	6.6
CAMPAIGN	2.9s	1.0	1.0	0.7	9.9	9.9	5.6	5.6
CELEBA	3.7s	1.0	1.1	0.5	4.6	4.5	2.5	2.5
CENSUS	107.1s	1.0	1.1	1.1	16.2	16.1	10.0	10.0
COVER	18.1s	1.0	1.0	0.6	14.6	14.5	8.0	8.0
FRAUD	16.6s	1.0	1.1	0.7	9.6	9.5	5.8	5.8
HTTP	7.7s	1.0	0.9	0.5	9.3	9.2	4.6	4.6
INTERNETADS	0.3s	1.0	2.3	1.1	1.1	1.1	1.1	1.1
MNIST	0.3s	1.0	0.8	0.5	4.8	4.8	2.8	2.8
MUSK	0.1s	1.0	1.1	0.3	3.8	3.8	2.3	2.3
SPEECH	0.3s	1.0	2.4	0.6	1.4	1.4	1.3	1.3

are averages over the 10 runs.[4]

We observe that the blackbox methods (top left and right in Figure 2.2) maintain a comparable AUCROC score to the baselines. For *k*NN-BB only Musk seemed to stand out. Even then, the lowest value achieved with musk was about -0.25 whereas the highest value achieved for any other dataset was around 0.03 - a maximum difference of only 0.28. For LOF there is slightly more variability, only Http had too far spread of individual AUCROC values. This indicates that for the black-box methods, we can use small values of *M* and *efS* while still obtaining accurate outlier detection results.

The picture looks very different for the whitebox method (bottom row). For the *k*NN method (bottom left), we see improved results over the whole parameter range for datasets such as Donors, ALOI, and Speech. However, we only see worse scores for datasets such as Mnist, Fraud, and Musk. For these, we usually require larger *M* and *ef* values to obtain comparable results. The LOF variant (bottom right) has again a larger result diversity. Except for Http, it often resembles or improves on the LOF baseline. A larger value of *M* is not as important as for the *k*NN variant, but *ef* should usually be set to a large value. This results indicate an interesting tradeoff: While the whitebox method has faster running time, it seems that the blackbox method can work with much smaller parameters.

[4]Building the HNSW index using `hnswlib` in a multicore environment lead to slightly different graphs due to different pruning orders. This non-determinism resulted in different quality scores for the same hyperparameter settings.

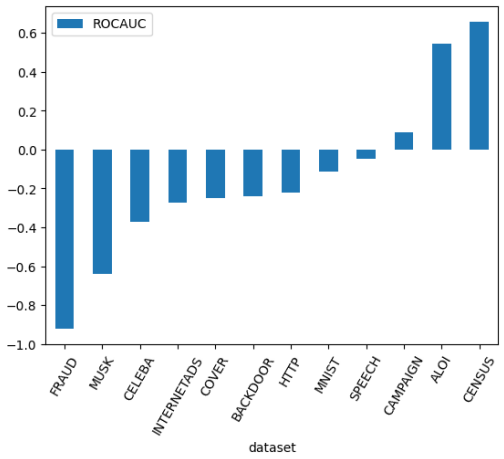


Figure 2.3: The correlation coefficient between recall and AUCROC for kNN-BB.

Scalability considerations. Table 2.3 summarizes the running time of the proposed methods on the small- to medium-sized datasets. We provide the time of the kNN method ($k = 100$), and the speedup over the kNN method to compare performance. Except for InternetAds and Speech, both the black-box and the white-box methods speed up the outlier detection significantly. In particular, the whitebox method shows the promised improvements over the blackbox method, usually being around twice as fast as the blackbox method.

Is better recall of the nearest neighbours always better for outlier detection? Figure 2.3 shows an evaluation of how the individual recall value of the approximate neighborhood ($k = 100$) influences the outlier detection quality. The figure shows a plot of the Pearson correlation coefficient between average recall and average AUCROC. The behavior is very dataset depend: In particular for Fraud and Musk, a good result can only be achieved with approximate neighborhoods. For other datasets, such as ALOI, Census, and Donors, the correlation is positive and points towards accurate neighborhood representations positively influencing the quality.

2.5.2 Results on the large Kitsune datasets

Now consider the bottom three datasets in Table 2.1. On these datasets the PyOD implementation of kNN and LOF could not finish the task in several hours.

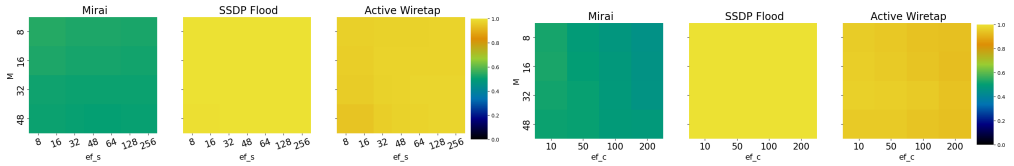


Figure 2.4: Heatmap of AUCROC scores of the blackbox (left)/whitebox (right) kNN method for the Kitsune datasets as a function of parameters efS and efC (x-axis) and M (y-axis). Note that the scale is absolute due to the unavailability of baseline AUCROC values. For the blackbox method $k = 15$ and $efC = 200$ were chosen.

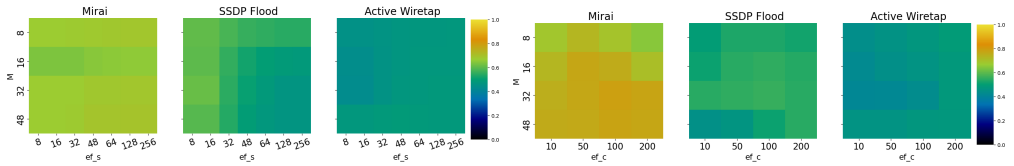


Figure 2.5: Heatmap of AUCROC scores of the blackbox (left)/whitebox (right) LOF method for the Kitsune datasets as a function of parameters efS/efC (x-axis) and M (y-axis). For the blackbox method $k = 15$ and $efC = 200$ were chosen.

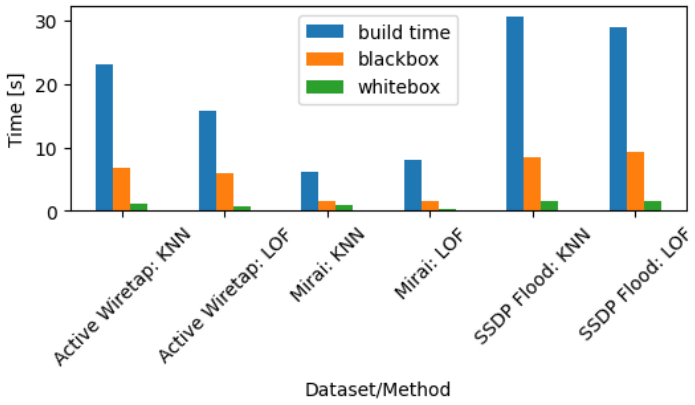


Figure 2.6: Running time whitebox/blackbox on Kitsune datasets.

Quality. Figure 2.4 presents an overview of the detection quality of the kNN-based methods on the three Kitsune datasets. Outliers in the SSDP Flood and Active Wiretap datasets are well identified using the kNN method, with both the whitebox and black-box method achieving similar quality. In contrast the outliers in the Mirai dataset are not identified through global methods as the AUCROC score achieved is comparable to what would be expected from random predictions. Figure 2.5 shows that the Mirai dataset is indeed identified by local methods, such as LOF-BB and LOF-WB. For the largest values of M and efC, the local methods

outperform the *k*NN methods (0.69 vs. 0.5 AUCROC for blackbox and 0.77 vs. 0.48 for whitebox). In contrast, the outliers of the SSDP Flood and Active Wiretap datasets do not seem to be identified by the local methods.

Scalability analysis. Figure 2.6 presents running time results on the three largest datasets for the whitebox and the blackbox method. The plots present the *minimum speed-up* of the white-box method vs. the black-box method, i.e., it is the smallest difference in performance we have observed in the tested parameter range. The plot shows that the majority of the running time is spent in building the index. The black-box method is roughly a factor of 4 faster than the time it took to build the index; the white-box method can carry out the detection step 10—30x faster. As seen in the previous paragraph, this comes at *no cost in quality*. We remark once more that while the detection time is below one minute for the proposed methods, the PyOD-based baseline cannot carry out the task within several hours.

2.6 Discussion

In this paper, we considered the scalable computation of *k*NN and LOF-based outlier scores. We suggested two variants, one based on using HNSW as a blackbox to compute approximate neighborhoods through *k*-NN searches, the other based on traversing the HNSW graph directly. Our experiments suggest that both variants provide a scalable alternative to the methods available in standard libraries such as `sklearn` and `PyOD`. In some cases, we noticed that the approximation helps the outlier detection result and provides better quality than an exact neighborhood computation.

In our evaluation we fixed certain values – namely *k* and *efC* for the blackbox. While the results presented here do hold true more generally, it would be interesting for future work to study more closely the impact of these parameters. Especially *k* could be of interest as the optimal *k* for a NN-based outlier detection method can vary greatly from dataset to dataset. The parameter *efC* impacts the build time of the HNSW data structure. Seeing as build time had the greatest contribution to the running time it would be of interest to see if this could be improved while still maintaining a certain level of quality. In a more general sense it would be interesting to do a more in depth study of how to choose good parameters given certain dataset attributes.

In our study, we did not make use of the hierarchy that is provided with HNSW: Can this hierarchy be used to improve the whitebox approach? Moreover, `ODIN` used the kNN graph in a novel way by checking the in-degree, can the outlier detection goal be integrated into the graph building process? This could both allow better scalability, but could also give rise to novel estimators.

With this study, we hope to initiate a line of research that treats approximate methods as tools for algorithm design that go beyond replacing exact neighborhoods with their approximate counterparts. This idea can readily be applied in other applications: For example, in future work we would like to study how to use the HNSW graph to build a (hierarchical) density-based clustering. Many other data mining tasks can be studied under this research lens.

Chapter 3

High-Dimensional Density-Based Clustering Using Locality-Sensitive Hashing

First published in: Conference on Extending Database Technology (EDBT 2025) [OATZ25]
Joint work with: Martin Aumüller, Viktor Bello Thomsen and Arthur Zimek

Abstract

The DBSCAN algorithm is a popular density-based clustering method to find clusters of arbitrary shapes without requiring an initial guess on the number of clusters. While there are methods to run DBSCAN efficiently in low-dimensional data in near-linear time, there remains a need for an efficient DBSCAN algorithm that scales to high-dimensional data. The bottleneck in high-dimensional data is that the range queries necessary in carrying out the algorithm suffer from the curse of dimensionality. In this paper we present the SRRDBSCAN algorithm. This algorithm is an implementation of approximate DBSCAN using locality-sensitive hashing. We prove subquadratic running time bounds under reasonable assumptions about the data. An important ingredient in the design of the data structure is the use of a multi-level LSH data structure, which automatically adapts to the density of data points. An extensive empirical analysis shows that the approximation does not significantly impact the quality of the clustering found by the algorithm as compared to the exact DBSCAN clustering. Moreover, our algorithm is competitive with other approaches even in low-dimensional settings, and thus provides a general-purpose DBSCAN implementation for arbitrary data.

3.1 Introduction

In recent years the amount of data available has increased dramatically. It is no longer uncommon for data collections to consist of millions if not billions of data points. Furthermore, and particularly driven by deep-learning-based embeddings such as CLIP [RKH+21], each data point can have many features (dimensions); for an algorithm to be viable for data analysis tasks, it needs to be able to keep up with the trend of increasing size and dimensions in data.

Clustering data is one of the basic primitives of data mining. There exist many different approaches to clustering. One important approach is k-means and its variants, in which the analyst has a good guess on the number k of clusters (or tries out different values of k). If the number of clusters is not known, or the clusters have a non-convex shape, *density-based approaches* are often the preferred approach. Examples of such methods are DBSCAN [EKSX96], HDBSCAN [CMS13, CMZS15], hierarchical clustering [ABKS99], and others [CKSZ20]. In this paper, we focus on DBSCAN.

Informally, the DBSCAN problem asks to group the data in such a way that the clusters themselves should correspond to dense regions of points and should be separated by sparse regions. These regions are created by identifying points with more than a certain threshold of close points at a certain distance threshold. Both the number of points required (`minPts`) and the distance threshold (ε) are parameters set by the analyst. Points satisfying the properties are so called "core points"; they are used to initiate the formation of the dense regions, and based on the notion of ε-reachability, clusters are formed. A formal definition of the problem is given in Section 3.3.

The simplest way to compute the clustering of n data points requires the calculation of the distances between all pairs of points. In this case the number of required distance computations is $O(n^2)$. For many distance measures, the distance comparison of two points in $\mathbb{R}^d$ can be carried out in time $O(d)$, so the total running time is $O(n^2 d)$. Due to the *curse of dimensionality* [BGRS99], approximate techniques are necessary to break this quadratic-time barrier for an arbitrary number of dimensions. Gan and Tao [GT15] were the first to formally define the c-approximate DBSCAN problem, which relaxes the reachability criterion for clusters: if a pair of core points is $c \cdot \varepsilon$ away, they may belong to the same cluster. The purpose of this paper is to propose efficient implementations of approximate DBSCAN for high-dimensional data based on hashing techniques.

As pointed out by Gan and Tao [GT15], the running time of the DBSCAN algorithm for arbitrary but constant d was up for dispute for a long time. In their paper, they proved both a lower bound of $\Omega(n^{4/3})$ for exact DBSCAN, and provided a linear time algorithm for an approximate version. While this sounds as good as one can hope for (each point has to be inspected at least once to make a clustering decision for it), the dependency of their algorithm is *exponential* in the dimension of the dataset. As pointed out by Schubert et al. [SSE+17], even for small d the suggested implementation did in fact not translate into practical improvements.

The present paper studies the design space of locality-sensitive hashing (LSH [IM98]) based algorithms to solve the approximate variant of DBSCAN with strong probabilistic guarantees. Our algorithm, to be described in Section 3.4, provides the following guarantees:

Theorem 3.1.1 (Informal version of Theorem 3.4.3)**.** *Let $S \subseteq \mathbb{R}^d$ be the dataset, minPts and ε be the DBSCAN-specific parameters, and $c \geq 1$ be an approximation factor. There exists an algorithm that solves the c-approximate DBSCAN problem under Euclidean distance with constant probability. Its expected running time*[1] *is*

$$\tilde{O}(minPts^{1-1/c^2} n^{1+1/c^2} d),$$

if the neighborhood of core points is "not too dense."

For example, consider the case that $c = 2$, which means that points at distance ε to 2ε may count for a point being a core point in approximate DBSCAN. If minPts is set to some constant, the expected running time of the algorithm is $O(dn^{1.25})$ *for arbitrary dimensionality*. Even for extreme settings, such as minPts = n, or $c = 1$ (exact DBSCAN), the running time smoothly degrades to guarantee quadratic time in the worst-case.

The backbone of the algorithm is the Multi-level LSH data structure first described by Ahle et al. [AAP17]. On a high level, our algorithm uses this data structure for DBSCAN to reduce the number of distance computations necessary by only considering points hashed together in the data structure. We will both analyze theoretically and observe empirically that different density regions require access to finer granularity of the hash function. In previous hashing-based work such as [KGPT22], only a single level is used for all points; this will necessarily translate into quadratic running time for some parameter choices of DBSCAN.

[1] With the $\tilde{O}(.)$ notation we suppress polylogarithmic factors in n and *minPts*.

Contributions. The contribution of this paper is the design, implementation, and evaluation of an efficient multi-level LSH data structure that provides a solution to the approximate DBSCAN problem. The proposed algorithm is supported by theoretical guarantees on the time complexity as well as the correctness of the output. The general description of the algorithm of Section 3.4 uses LSH as a black-box which means that our algorithm can be applied in all distance spaces that are "LSH-able" [CK15]. Only in Section 3.5 we describe particular details of the LSH function and other implementation choices that influence empirical performance. The experimental results in Section 3.6 show competitive performance in running time of the algorithm compared to existing DBSCAN implementations in the literature, even for low-dimensional data. As we will show in the experimental section, none of the competitors that provide theoretical guarantees on the output quality are able to solve both low- and high-dimensional clustering problems on datasets containing millions of data points within reasonable time limits, often failing due to the dimensionality of the datasets, or certain clustering characteristica rendering their approach inefficient—both in terms of running time and memory consumption. Both our implementation, the experimental framework, and additional experimental results are available at `https://github.com/CamillaOkkels/srrdbscan` and `https://github.com/CamillaOkkels/dbscan-benchmark`.

We stress that, while LSH is a well-known technique for enabling scalable data analysis, no prior work on (approximate) DBSCAN has both strong theoretical guarantees on the result quality *and* good empirical performance.

3.2 Related work

There exist many proposed variations on the original DBSCAN algorithm that try to mitigate its drawbacks. In the 2-dimensional case efficient algorithms exist that have improved the running time from $O(n^2)$ to $O(n \cdot \log(n))$ [dBGR19]. Efficient algorithms for the general d-dimensional case as proposed by Gan and Tao [GT15] as well as Schubert et al. [SSE+17]—while achieving sub-quadratic running times—rely on constant dimensionality. Indeed, when the dimensionality becomes high, these algorithms do not gain significant improvement compared to the naïve implementation of DBSCAN. While we focus our efforts on efficient DBSCAN implementations, the same research questions are actively investigated for hierarchical clusterings as well [GT18, dBGR19, ACH19, WYGS21, HYS23].

Subspace clustering. Much research has been spent on clustering in high-dimensional data [KKZ09, HKZ23], in particular including some adaptations of DBSCAN such as SUBCLU [KKK04], PreDeCon [BKKK04], 4C [BKKZ04], or COPAC [ABK+07]. These and many other methods in this area broadly described as subspace clustering do not, however, focus on efficiency and robustness against the curse of dimensionality when solving the DBSCAN problem as such, but on re-defining the problem as to identify (possibly multiple) clustering solutions in (different) subspaces. These and related methods, although explicitly addressing some issues that come with high dimensional data, namely the different attributions of relevant vs. irrelevant attributes for different clusters, do not address the scalability and robustness issues for the underlying basic DBSCAN problem when it comes to very large and high-dimensional data.

Efficient implementations. Most approaches to speeding up DBSCAN rely on using an indexing structure to allow for faster (range) search operations. These structures usually split up the high-dimensional space into a grid; since the number of grid cells increases exponentially in the dimensionality d, such approaches will often have depend exponentially on the dimensionality. An efficient implementation of DBSCAN was proposed by Gan and Tao in [GT15, GT17] for 2-dimensional as well as d-dimensional DBSCAN. The 2D algorithm solves DBSCAN in time $O(n \cdot \log(n))$ by imposing a grid and using the cells to reduce the number of distance computations for core point identification. They presented an approximate and exact algorithm for solving DBSCAN for $d \geq 3$. The data structure for the approximate algorithm is a quadtree-like hierarchical grid structure. They achieve a running time of $O(n)$ for constant d, where the running time depends exponentially on d. The state-of-the-art grid-based approach is TPEDBSCAN by Wang et al. [WGS20] which focused on an efficient parallel implementation of the grid-based approach; it has the same exponential dependency on d.

Another line of work based the computation of DBSCAN clusterings on sampling approaches. SNGDBSCAN by Jiang et al. [JJL20] is the state-of-the-art approach in this regime. For constant dimensionality d (again hiding an exponential dependency), they prove that sampling $O(n \log n)$ random pairs of points is sufficient to identify the DBSCAN clusters if "clusters are sufficiently dense."

Hashing-based clustering. Koga et al. [KIW07] and Keramatian et al. [KGPT22] both rely on LSH to design clustering algorithms that work for high-dimensional data. In contrast to grid-based approaches that

impose a grid structure, the main idea of these hashing-based algorithms is to hash data points into (multiple) hash tables. Traditionally, this is achieved using LSH. The promise of using LSH is that the probability of hashing to the same hash value is a function increasing in the similarity of the data points. The main bottleneck of all hashing-based algorithms is that in each bucket of a hash table, an all-to-all comparison has to be carried out to find close neighbors for all data points in the bucket, yielding *quadratic time in the bucket size*. Thus, care has to be taken in choosing parameters such that (i) buckets are not too large and (ii) enough repetitions are carried out to guarantee core point identification with high enough probability. Koga et al. [KIW07] introduce an algorithm called *LSH-LINK* to compute a hierarchical clustering. However, their running time guarantees might exceed $O(n^2)$, and—more importantly—there is no correctness guarantee for their algorithm. Keramatian et al. [KGPT22] recently provided an efficient clustering algorithm called *IP.LSH.DBSCAN* (IP.LSH) that computes a single reference point per bucket. This heuristic alleviates the quadratic-time barrier in each bucket, but does not provide strong formal guarantees [KGPT22, Lemma 2], which we will also observe in the experimental evaluation (Section 3.6.2). They also require additional parameters to set the number of hash functions and hash tables to which the algorithm is very sensitive, also to be explored in the experimental section. Other LSH-based approaches with no theoretical guarantees on the result quality include [SQ19, WGZ07].

Other approximate methods. While not being hashing-based, random projection-based ideas have been explored to cluster data. This idea was pioneered by Schneider et al. [SV17]. In very recent work, Xu and Pham [XP24] designed an efficient algorithm for DBSCAN using random projections. While they give theoretical guarantees on certain subroutines, there is no general running time statement and the correctness of the merging step requires the same strong assumptions [XP24, Assumption 1] as sampling-based approaches [JJL20].

3.3 Preliminaries

3.3.1 Definitions

In the following, assume a data set $S \subseteq \mathbb{R}^d$ consisting of n points is given. Let $d\colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ be some distance measure, for example the Euclidean distance $d(p,q) = \|p-q\|_2$. For a point $q \in \mathbb{R}^d$ and a distance threshold

$\varepsilon \geq 0$, define $N_\varepsilon(q) = \{p \in S \mid d(p,q) \leq \varepsilon\}$ as the ε-neighborhood of q in S. While certain core concepts of DBSCAN [EKSX96] are standard, in particular the role of border points in clusters is often vaguely discussed. We consider the DBSCAN* problem [CMZS15] in which border points are not part of a clustering and define an approximate version as follows:

Definition 3.3.1 (c-approximate core point sets)**.** *Let $q \in S$ be a data point, let ε and minPts be the DBSCAN parameters, and let $c \geq 1$ be an approximation factor. If $|N_\varepsilon(q)| \geq$ minPts, q is a* core point*. If $|N_\varepsilon(q)| <$ minPts and $|N_{c\varepsilon}(q)| \geq$ minPts, q is a c-*approximate core point*. Let CP^* be the set of all core points in S. A set $CP \subseteq S$ is a c-*approximate core point set *if $CP^* \subseteq CP$ and all points in $CP \setminus CP^*$ are c-approximate core points.*

We remark that for $c = 1$, the core point set is the unique set of core points in S in the classical DBSCAN notion. For $c > 1$, we allow that the core point set moreover includes points that are only core points at distance threshold $c\varepsilon$, but we do not require that all of these are included in the c-approximate core point set.

Definition 3.3.2 (c-approximate density-reachability)**.** *Let CP be an c-approximate core point set. A point p is said to be density reachable in CP from another point q if there exists a sequence of points $(p := p_1, p_2, \ldots, p_\tau =: q)$ from CP (with $\tau \geq 2$) that satisfies $d(p_i, p_{i+1}) \leq \varepsilon$ for each $i \in [1, \tau - 1]$.*

Definition 3.3.3 (c-approximate DBSCAN* clustering)**.** *Given c, ε, and minPts, let $CP \subseteq S$ be an c-approximate core point set. Let $\sim_{CP}$ be the relation such that $p \sim_{CP} q$ if and only if p and q are c-approximately density-reachable in CP. The $(c, \varepsilon,$minPts$)$-approximate DBSCAN* clustering $\mathscr{C}$ of CP are the equivalence classes of $\sim_{CP}$. The remaining points $\{p \in S \setminus CP\}$ are noise.*

Alternatively, we can see the clusters as the connected components induced by the c-approximate core point set CP of the graph $G = (S, E)$, where an edge exists if two points are at distance at most ε [dBGR19]. For $c = 1$, we recover the exact DBSCAN* clustering.

Figure 3.1 shows a DBSCAN* clustering according to Definition 3.3.3. The clusters returned by DBSCAN* are $\{p1, p2, \ldots, p_8\}$ and $\{p10, \ldots, p16\}$. Both $p9$ and $p17$ are ignored because they are not a core point.

Both [GT17] and [dBGR19] use a different notion of approximation. Their core point classification is exact, but Definition 3.3.2 (iii) uses $c \cdot \varepsilon$ as a distance value. While the individual approximate clustering might differ, it is straight-forward to see that the quality of the resulting approximate

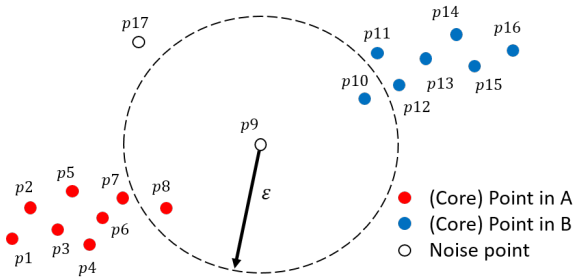


Figure 3.1: An example of a DBSCAN* clustering with the depicted ε and $minPts = 4$. The clusters only consist of core points.

clustering can be "sandwiched" in between two exact solutions. Thus, these different notions are similar in their quality.

Theorem 3.3.4 (Sandwich theorem [GT17]). *Given $S \subseteq \mathbb{R}^d$,ε,$minPts$, and $c \geq 1$.*

- *Let $\mathscr{C}_\varepsilon$ be the clusters for $(\varepsilon,minPts)$-DBSCAN*.*
- *Let $\mathscr{C}_c$ be a $(c,\varepsilon,minPts)$-approximate DBSCAN* clustering.*
- *Let $\mathscr{C}_{c\cdot\varepsilon}$ be the clusters for $(c\varepsilon,minPts)$-DBSCAN*.*

Then we have: (1) For each $C_\varepsilon \in \mathscr{C}_\varepsilon$ there exists $C_c \in \mathscr{C}_c$ such that $C_\varepsilon \subseteq C_c$. (2) For each $C_c \in \mathscr{C}_c$ there exists $C_{c\cdot\varepsilon} \in \mathscr{C}_{c\cdot\varepsilon}$ such that $C_c \subseteq C_{c\cdot\varepsilon}$.

Theorem 3.3.4 is a natural consequence of the well documented fact (see for example [ABKS99, GT18]) that exact DBSCAN* clusters for small epsilon are subsets of DBSCAN* clusters for larger epsilon. In particular, all core points of $(\varepsilon,minPts)$-DBSCAN* are also c-approximate core points, and all non-core points in $(c\varepsilon,minPts)$-DBSCAN* are also non-core points in the approximate setting. The strength of this theorem lies in the fact that an approximate DBSCAN* definition can be sandwiched between two exact solutions. Consider the instances where the exact DBSCAN* clusters are robust up to a small variation in ε, i.e., $\mathscr{C}_\varepsilon = \mathscr{C}_{c\cdot\varepsilon}$. Theorem 3.3.4 states that the solution to approximate DBSCAN* with parameters $(c,\varepsilon,minPts)$ would also be the solution to exact DBSCAN*. In other words the similarity between the solution to exact DBSCAN* and approximate DBSCAN is entirely dependent on the robustness of the clusters.

3.3.2 Objective of DBSCAN* algorithms

The objective of DBSCAN* is to find a clustering $\mathscr{C}$ given parameters ε and $minPts$ that satisfy Definition 3.3.3 with $c = 1$. Any algorithm for

solving the DBSCAN* problem should have the following properties: (i) It needs to identify core points given the parameters ε and *minPts*. (ii) Given the list of core points CP and ε, clusters have to be identified based on density-reachability.

3.3.3 Locality-Sensitive Hashing

To reduce distance computations we use Locality-Sensitive Hashing [IM98]. Let $\Pr(\chi)$ denote the probability of an event χ.

Definition 3.3.5 (Locality-Sensitive Hash Family [IM98])**.** *Given a distance function d, a family of hash functions $\mathcal{H} = \{h : \mathbb{R}^d \to R\}$ is said to be (d_1,d_2,p_1,p_2)-sensitive if for any pair of points $p,q \in \mathbb{R}^d$ the following statements are true:*

(i) *For $d(q,p) \leq d_1$, $\Pr_{h\sim\mathcal{H}}(h(q) = h(p)) \geq p_1$.*
(ii) *For $d(q,p) \geq d_2$, $\Pr_{h\sim\mathcal{H}}(h(q) = h(p)) \leq p_2$.*
(iii) *$p_1 \geq p_2$ and $d_1 \leq d_2$.*

Many LSH families have been described in the past. Of particular importance are BitSampling [IM98] for binary strings under Hamming distance, MinHash [Bro97] for sets, SimHash [Cha02] and Crosspolytope LSH [AIL+15] for angular distance on the unit sphere, and EuclideanLSH [DIIM04] for Euclidean space. As it is standard in the literature [Ahl20], we assume that p_1 is constant. The quality of an LSH is measured as $\rho = \log(1/p_1)/\log(1/p_2) \leq 1$.

3.4 Proposed algorithm

In this section, we first describe our algorithm for solving the approximate DBSCAN* problem. Algorithm 3 outlines the algorithm SRRDBSCAN (Spherical Range Reporting DBSCAN). In addition to the DBSCAN* parameters S, ε, and *minPts*, and the approximation factor c, our algorithms takes two user parameters $\delta \in (0,1)$ (which controls the success probability) and $L \geq 1$ (which governs the space consumption of the index data structure). In Sections 3.4.1—3.4.4 we discuss the proposed algorithm in detail, including the index building phase, the core point identification, and merging core points into the final clustering. In Section 3.4.5 we argue for the algorithm's correctness and then analyze its running time.

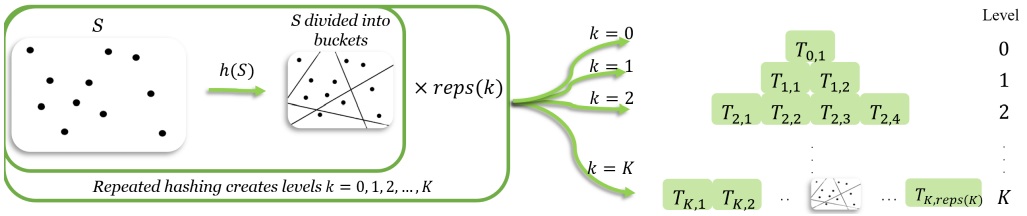


Figure 3.2: An illustration of the multi-level data structure. Given dataset S, starting from $k=0$, the data is hashed $\text{reps}(k)$ times for each level k to be constructed. In this example, $\text{reps}(k)=2^k$. This is repeated until the final level K for which $\text{reps}(K)\leq L$.

Algorithm 3: SRRDBSCAN(S,ε,$minPts$,L,δ,c)

Input: $S\subseteq\mathbb{R}^d$, parameters ε,$minPts$,L,δ,c
Output: A list of cluster labels.

1 $\mathcal{I}\leftarrow$ build multi-level LSH for S,L,δ // Sect. 3.4.1
2 $k_{\text{best}}\leftarrow$ findBestLevel(S,$\mathcal{I}$,L) // Sect. 3.4.3
3 $CP\leftarrow$ classifyCP(S,$\mathcal{I}$,k_{best},ε,$minPts$,c) // Sect. 3.4.3
4 clusters $\leftarrow$ clusterCore(CP, ε,L,δ) // Sect. 3.4.4
5 **return** *clusters*

3.4.1 Index building

The proposed algorithm makes use of the Multi-Level LSH data structure first described by Ahle et al. [AAP17]. Given a dataset $S\subseteq\mathbb{R}^d$, a parameter L, and access to a (p_1,p_2,d_1,d_2)-sensitive hash family $\mathcal{H}$ consisting of function that map from $\mathbb{R}^d$ to some range R, it builds a multi-level data structure as follows. For a level k,$0\leq k\leq K$, in the data structure, choose k hash functions $\{g_1,g_2,\ldots,g_k\}$ uniformly at random and independently from $\mathcal{H}$ to obtain the hash function $h_k=(g_1,g_2,\ldots,g_k)$. The hash value of a point $x\in\mathbb{R}^d$ is $h_k(x)=(g_1(x),\ldots,g_k(x))\in R^k$. For each $x\in S$, compute $h_k(x)$ and, using a hash table, associate with each hash value the list of points that hash to it. Repeat this process $\text{reps}(k)$ many times independently, where $\text{reps}(k)$ is a parameter to be set later. The data structure for level k consists of these $\text{reps}(k)$ hash tables $T_{k,1},\ldots,T_{k,\text{reps}(k)}$. We denote by $T_{k,i}(p)$ the bucket that point p was hashed to on level k and repetition i. Repeat this process independently for each $k\in\{0,\ldots,K\}$, where K is defined to be the largest integer satisfying $\text{reps}(K)\leq L$. Figure 3.2 provides an example. For reasons that will be addressed in the analysis, L has to be large enough such that $K\geq\left\lceil\frac{\log(n/minPts)}{\log(1/p_2)}\right\rceil$.

3.4.2 Motivation for the data structure

Assume some index $T_{k,j}$ for $k \in \{0,\dots,K\}$ and $1 \leq j \leq \text{reps}(k)$ built for a dataset S. Let ε,minPts be the DBSCAN parameters, and let c be the approximation factor. For each $x \in S$ we have to identify if it is a core point and (potentially) merge cluster labels. Let us first consider the influence of the different levels $k \in \{0,\dots,K\}$. For a given level k, concatenating k hash functions chosen independently at random from $\mathcal{H}$ results in higher selectivity of the hash function: if two points are at distance at most $d_1 := \varepsilon$, their collision probability will be *at least* p_1^k, if they are at least $d_2 := c\varepsilon$ far apart, it is *at most* p_2^k (Def. 3.3.5). Let $p \in S$ be an arbitrary point. Given the properties of the LSH family, at most $n \cdot p_2^k$ points that are not within distance $c\varepsilon$ are going to collide with p under the hash function. However, a point that counts towards p being a core point has probability at least p_1^k of colliding with p. Repeating the construction $\Theta(1/p_1^k)$ times guarantees that it collides with p with constant probability. Thus, we can *upper bound* the work for deciding whether point p is a core point on level k by $O(p_1^{-k} \cdot (minPts + np_2^k))$. Yet, the actual work depends heavily on the data: If the dataset consists of clusters, each of size $minPts+1$ with very small intra-cluster distances, but large inter-cluster distances, then a small level k suffices to split up the clusters and the upper bound from before would clearly misguide us to use a larger level with higher cost. In the most extreme case where we have n copies of the same data point, we would spend time $O(minPts \cdot n/p_1^k)$, so the best level is the first. But if the dataset contains a long "chain" of core points at distance ε to each other, and the remaining dataset is noise, slightly further away than distance $c \cdot \varepsilon$, then we would indeed need to inspect a deeper level in which these noise points are likely to be split up from core points. Hence we propose to first compute the work that needs to be carried out on level k (which is easier since we only have to sum up the number of colliding points, i.e., the bucket sizes in the hash table), choose the level with minimal cost to carry out the core point identification, and later merge clusters. The multi-level data structure allows for *data dependent* choices.

3.4.3 Core point identification

To identify c-approximate core points, i.e., the points that have at least *minPts* many neighbors at distance ε (where we might count points at distance at most $c\varepsilon$), we proceed in two steps. First, using Algorithm 4, we determine a best level to carry out the core point identification in the

Algorithm 4: findBestLevel(S,$\mathcal{I}$,L)

Input: Points S, data structure $\mathcal{I}$, and a parameter L
Output: Value k_{best} of "best level"

1 $n \leftarrow |S|, \quad k \leftarrow 0, \quad k_{\text{best}} \leftarrow 0, \quad \omega_{k_{\text{best}}} \leftarrow n^2$
2 **while** $n \cdot reps(k) \leq \min(n \cdot L, \omega_{k_{best}})$ **do**
3 $\quad \omega_k \leftarrow \sum_{q \in S} \sum_{i=1}^{\text{reps}(k)} \frac{|T_{k,i}(q)|-1}{2} + 1$
4 $\quad$ **if** $\omega_k \leq \omega_{k_{best}}$ **then**
5 $\quad\quad k_{\text{best}} \leftarrow k, \quad \omega_{k_{\text{best}}} \leftarrow \omega_k$
6 $\quad k \leftarrow k+1$
7 **return** k_{best}

multi-level data structure. To do so, we compute in Line 3 the amount of work that will be necessary to inspect all buckets on level k and to carry out all-to-all comparisons in each of them (without actually carrying them out). We proceed in the order $k = 0,1,2,...$ and stop as soon as the time necessary for inspecting the buckets is already more than carrying out these all-to-all comparisons in the best level found so far. Let k_{best} be the level returned by Algorithm 4. Algorithm 5 describes the core point identification given such a best level. For a point $p \in S$, it counts the number of distinct points in the buckets $T_{k_{\text{best}},1}(p),...,T_{k_{\text{best}},\text{reps}(k_{\text{best}})}(p)$ that are at distance at most $c \cdot \varepsilon$. It stops as soon as the number is *minPts*, and classifies the point as a core point.

3.4.4 Merging core points

Algorithm 6 shows how to identify clusters given the list CP of core points identified in the previous step. First, initialize a union-find data structure. Given CP, rebuild the multi-level data structure on CP and identify the best level k_{best} using Algorithm 4. Next, identify *all* the points at distance at most ε from a core point p by going through buckets $T_{k_{\text{best}},1}(p),...,T_{k_{\text{best}},\text{reps}(k_{\text{best}})}(p)$. For each point q found in the buckets, check if the distance is at most ε and merge the cluster using a union operation if that is the case. The clustering $\mathscr{C}$ is the sets of the union-find data structure.

Algorithm 5: classifyCP(S,$\mathcal{I}$,k_{best},ε,$minPts$,c)

Input: Points S, data structure $\mathcal{I}$, a best level k_{best}, and parameters ε,$minPts$,c
Output: List CP of core points

```
1  CP ← { }
2  foreach p ∈ S do
3  |  N(p) = { }
4  |  for i ← 1 to reps(k_best) do
5  |  |  foreach q ∈ T_{k_best,i}(p) do
6  |  |  |  if d(p,q) ≤ c·ε and q ∉ N(p) then
7  |  |  |  |  N(p) ← N(p) ∪ {q}
8  |  |  |  if |N(p)| = minPts then
9  |  |  |  |  CP ← CP ∪ {p}
10 |  |  |  |  break (out of line 4 loop)
11 return CP
```

Algorithm 6: clusterCore(CP, ε,L,δ)

Input: A list CP of core points and parameters ε,L,δ
Output: A list of cluster labels/identifiers for each point.

```
1  uf ← UnionFind(|CP|) ;                    // Initialize
   Union-Find data structure to track clusters
2  I ← build multi-level LSH for CP,L,δ;
3  k_best ← findBestLevel(CP,I,L)
4  foreach p ∈ CP do
5  |  for i ← 1 to reps(k_best) do
6  |  |  foreach q ∈ T_{k_best,i}(p) do
7  |  |  |  if uf.find(p) ≠ uf.find(q) and d(p,q) ≤ ε then
8  |  |  |  |  uf.union(p,q);
9  return {uf.find(p) | p ∈ CP};
```

3.4.5 Analysis of the algorithm

Correctness guarantee

The following lemma says that the algorithm works correctly with probability at least $1-\delta$ no matter which level is chosen for the individual points in Algorithm 5.

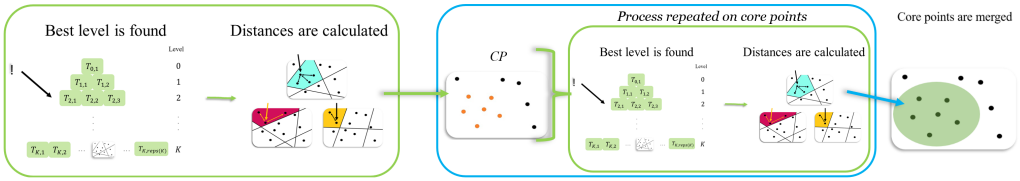


Figure 3.3: Illustration of the SRRDBSCAN algorithm. Part (a) illustrates the algorithm up to and including core point identification outlined in lines 1-3 in Algorithm 3. Part (b) illustrates the merging step outlined in Algorithm 6. The process of building the data structure and searching for close points is repeated, but only on core points, and the output is clusters.

Lemma 3.4.1 (Correctness guarantee). *Let $S \subseteq \mathbb{R}^d$ contain n data points. Let CP^* be the set of core points with respect to $(\varepsilon, minPts)$-DBSCAN* on S. Set*

$$reps(k) = p_1^{-k} \cdot \ln\left(n \cdot minPts \cdot (k+1)^2 / \delta\right)$$

and let CP be the output of Algorithm 5. With probability at least $1-\delta$, $CP^* \subseteq CP$ and all points in CP are c-approximate core points.

For the proof, we will make use of the following lemma:

Lemma 3.4.2. *Let $p \in S$ be a core point with respect to $(\varepsilon, minPts)$-DBSCAN* and given some $\delta \in (0,1)$. Then for*

$$reps(k) \geq p_1^{-k} \cdot \ln(minPts/\delta),$$

p is identified as a core point by Alg. 5 with probability at least $1-\delta$.

Proof. Fix a level $k \in \{0,\ldots,K\}$. Let p be a core point in S. Given a point q with $d(q,p) \leq \varepsilon$, let A_q be the event that p and q do not get hashed together in any hash table on level k. Since all repetitions are independent and by the definition of LSH, $\Pr(A_i) \leq (1-p_1^k)^{\text{reps}(k)}$. Now, w.l.o.g., suppose $|N_\varepsilon(p)| = minPts$.[2] In order to identify p as a core point, it should be hashed with every point in $N_\varepsilon(p)$ in at least one hash table on the level. Using a union bound we can find an upper bound on the probability that there exists a hash table where p is not hashed with some $q \in N_\varepsilon(p)$:

$$\Pr\Big(\bigcup_{q \in N_\varepsilon(p)} A_q\Big) \leq \sum_{j=1}^{minPts} \left(1-p_1^k\right)^{\text{reps}(k)} = minPts \cdot \left(1-p_1^k\right)^{\text{reps}(k)}$$

[2] If $|N_\varepsilon(p)| \geq minPts$, the probability that p gets hashed with $minPts$ close points, thereby identifying it as a core point, is only higher.

Using the upper bound $\left(1-p_1^k\right)^{\text{reps}(k)} \leq e^{-p_1^k \cdot \text{reps}(k)}$ we get:

$$\Pr\Big(\bigcup_{i=1}^{minPts} A_i\Big) \leq minPts \cdot e^{-p_1^k \cdot \text{reps}(k)} \tag{3.4.1}$$

Solving $minPts \cdot e^{-p_1^k \cdot \text{reps}(k)} \leq \delta$ for $\text{reps}(k)$ ends the proof. □

Proof of Lemma 3.4.1. Let B_i be the event for any given core point i that it is not identified at a level k. From (3.4.1) we have that $\Pr(B_i) \leq minPts \cdot e^{-p_1^k \cdot \text{reps}(k)}$. Using a union bound over all n potential core points, we can upper bound the probability that at least one core point is not identified on at least one level as

$$\Pr\Big(\bigcup_{i=1}^{n\cdot(K+1)} B_i\Big) \leq \sum_{k=0}^{K} n \cdot minPts \cdot e^{-p_1^k \cdot \text{reps}(k)}$$

By setting $\text{reps}(k) \geq p_1^{-k} \ln\left(2 \cdot n \cdot minPts \cdot (k+1)^2/\delta\right)$, we may bound this probability by:

$$\Pr\Big(\bigcup_{i=1}^{n\cdot(K+1)} B_i\Big) \leq \sum_{k=0}^{K} \frac{\delta}{2\cdot(k+1)^2} \leq \delta \cdot \frac{\pi^2}{12} < \delta \ ,$$

where the last inequality uses the fact that $\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6}$. □

Running time analysis

We will prove the following theorem summarizing the expected running time of the proposed algorithm:

Theorem 3.4.3 (Running time)**.** *Let $S \subseteq \mathbb{R}^d$ with n points be given. Let $minPts, \varepsilon$ be the DBSCAN parameters, and let $L, c \geq 1, \delta > 0$ be user parameters. Define $\mathcal{N} = \sum_{q \in S} |N_{c\cdot\varepsilon}(q)|^{\rho(c)}$. With probability at least $1-\delta$, the output of Algorithm 3 is a $(c, minPts, \varepsilon)$-approximate DBSCAN* clustering. Assuming that hash function evaluations and distance comparisons take time $O(d)$, the expected running time is*

$$\tilde{O}\Big(d \cdot \max\left(nKL, n^{1+\rho(c)} minPts^{1-\rho(c)}, n^{\rho(c)} minPts^{-\rho(c)} \mathcal{N}\right)\Big).$$

The value $\rho = \log(1/p_1)/\log(1/p_2) \leq 1$ is governed by the LSH family that is used for a given distance measure. For example, EuclideanLSH [DIIM04] has $\rho(c) = 1/c$, and there exist families that achieve

$1/c^2$ [AI08]. The three terms in Theorem 3.4.3 represent the time to build the index, the expected time to identify all core points, and the expected time necessary to merge core points into clusters. If $|N_{c\varepsilon}(q)| = O(minPts)$, i.e., if the number of points at distance at most $c\varepsilon$ is not much larger than *minPts*, then the term $n^{\rho(c)} minPts^{-\rho(c)} \mathcal{N}$ is upper bounded by $O\big(n^{1+\rho(c)} minPts^{1-\rho(c)}\big)$. This means that the running time of the merging step is no higher than the time spent in the core point identification step.

In the following, we analyse each of the three steps separately, only counting the number of points inspected by the algorithm (i.e., disregarding the $O(d)$ term present in Theorem 3.4.3).

Index building From Lemma 3.4.2, we build the data structure for all levels $k \in \{0, \dots, K\}$ such that $\text{reps}(K) \leq L$, using that $\text{reps}(k) = p_1^{-k} \cdot \ln\big(n \cdot minPts \cdot k^2/\delta\big)$. There are $O(L)$ hash tables in the multi-level data structure, as $\sum_{0 \leq k \leq K} \text{reps}(k) = O\big(\text{reps}(K)\big)$. For each hash table, we have to evaluate not more than nK hash values and insert n points into the hash table. Thus, the overall time necessary to build the index is $O(nKL)$.

Core point identification We proceed in two steps: First, we show that a good level exists in the data structure; next, we rely on the properties of the Multi-Level LSH data structure proven in [AAP17] to argue that the adaptive level choice does not lead to a worse asymptotical running time.

Lemma 3.4.4. *Set* $k = \left\lceil \frac{\log(n/minPts)}{\log(1/p2)} \right\rceil$. *Carrying out Algorithm 5 on level k, the expected number of distance comparisons (Line 6) is* $\tilde{O}\big(n^{1+\rho} minPts^{1-\rho}\big)$.

Proof. By linearity of expectation, the expected number of distance comparisons over all n data points is just n times the expected number of distance comparisons for a single point.

Fix a point $q \in S$. There are three types of points it can get hashed with into the same bucket in $T_{k,i}$:

1. Close points (CL): All the points p with $d(q,p) \leq \varepsilon$. These count towards q being a core point. This happens with probability at most 1.
2. Middle points (MP): All the points p with $\varepsilon \leq d(q,p) \leq c \cdot \varepsilon$. These count towards q being a core point. This happens with probability at most p_1^k.
3. Far points (FP): All the points p with $d(q,p) \geq c \cdot \varepsilon$. This happens with probability at most p_2^k.

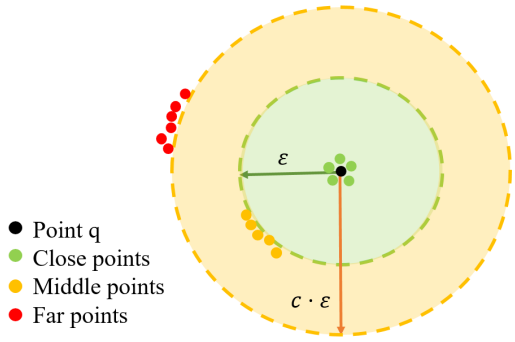


Figure 3.4: An illustration of the three different categories of points as seen from the perspective of a point q.

Figure 3.4 visualizes a setting in which these probabilities are attained, i.e., a worst-case setting for the analysis. The expected size of the bucket $T_{k,i}(q)$ is given by $\sum_{p \in S} \Pr(h(q) = h(p))$ and can be upper bounded by $E[|T_{k,i}(q)|] \leq \#CL + \#MP \cdot p_1^k + \#FP \cdot p_2^k$.

The first two summands of the summation of the right-hand side are at most *minPts*, since Algorithm 5 immediately returns as soon as *minPts* neighbors at distance at most $c\varepsilon$ are identified. By the choice of k, we may bound the final summand by $\#FP \cdot p_2^k \leq n \cdot p_2^k \leq minPts$. So, $E[|T_{k,i}(q)|] \leq minPts$ for a single repetition i. The excepted number of distance computations for q is then $O(\text{reps}(k) minPts)$ with $\text{reps}(k) = \tilde{O}(p_1^{-k})$ (cf. Lemma 3.4.1), which is

$$\tilde{O}\big((n/minPts)^{\rho} minPts\big) = \tilde{O}\big(n^{\rho} minPts^{1-\rho}\big)$$

. The proof follows from linearity of expectation over the n data points. □

While Lemma 3.4.4 proves that there exists a good level to check, the proposed algorithm first investigates the bucket sizes to make an adaptive choice for which level is the best to use. A core contribution of Ahle et al. [AAP17] states that this adaptive choice has asymptotically (almost) the same running time as if the best level had been known. To extend their Theorem 3 to our setting, first define

$$E(W_k) = \sum_{q \in S} \sum_{1 \leq i \leq \text{reps}(k)} \Big(O(1) + \sum_{x \in T_{k,i}(q)} \Pr\big(h(q) = h(x)\big) \Big) \tag{3.4.2}$$

$$W^* = \min_{0 \leq k \leq K} E(W_k)$$

This—purely abstractly—represents the expected work on the optimal level if the whole data distribution were known.

Lemma 3.4.5. *Let $S, \varepsilon, minPts, \delta$ be given. Let W^* be defined as above. The expected running time of Algorithm 4 and Algorithm 5 is $O(W^*)$.*

Proof. The proof will follow closely that of Ahle et al. [AAP17, Theorem 3]. First, consider Algorithm 4.

Given constant access to the size of the buckets, Line 3 takes time $O(n \cdot \text{reps}(k))$. Suppose the last value of k before the loop starting in Line 2 terminates is k^*. Using the choice of $\text{reps}(k)$ from Lemma 3.4.1, all iterations of the loop until that point in time took expected time

$$\begin{aligned}\sum_{k=1}^{k^*} O\big(n \cdot \ln\big(n \cdot minPts \cdot k^2/\delta\big) \cdot p_1^{-k}\big) \\ &\leq n \cdot \ln\big(n \cdot minPts \cdot (k^*)^2/\delta\big) \cdot p_1^{-k^*} \sum_{k=0}^{\infty} O\big(p_1^k\big) \\ &\leq O(\omega_{k_{\text{best}}}).\end{aligned}$$

The last step follows from the while condition which ensures that $n \cdot \text{reps}(k^*) \leq \omega_{k_{\text{best}}}$.

Algorithm 5 looks at the buckets of k_{best} and computes exactly $\omega_{k_{\text{best}}}$ distances. By Jensen's inequality,

$$E[\omega_{k_{\text{best}}}] = E[\min_{0 \leq k \leq K}(\omega_k)] \leq \min_{0 \leq k \leq K} E[\omega_k]$$

which equals $\min_{0 \leq k \leq K} E[W_k] = W^*$. □

We note that this result is slightly different from the result of Ahle et al. [AAP17, Theorem 1], which observed a slight asymptotic difference between the "clairvoyant" algorithm that knows the best level and the adaptive variant. This is because our definition of $E[W_k]$ in (3.4.2) uses the same repetition count as the adaptive algorithm. In the setting of [AAP17], they have an asymptotic blow-up because they carry out a different number of repetitions. Lemmas 3.4.4 and 3.4.5 together show that the expected running time of Algorithms 4 and 5 is $\tilde{O}(n^{1+\rho} minPts^{1-\rho})$, and can potentially be much better.

Merging clusters We analyze the running time of Algorithm 6.

Lemma 3.4.6 (Running time of merging step)**.** *Given a set CP of c-approximate core points, let $\mathcal{N} = \sum_{q \in CP} |N_{c \cdot \varepsilon}(q)|$. Algorithm 6 labels the core points in expected time*

$$\tilde{O}(|CP|^{\rho(c)} \cdot minPts^{-\rho(c)} \cdot \mathcal{N}).$$

The proof follows the same steps as seen for the proofs of Lemma 3.4.4 with the main difference being that Algorithm 6 cannot return after finding just *minPts* close points as we observe the merging of two sub-clusters can depend on a single core point. As a result the expected bucket sizes—and consequently the running time—will depend on the neighborhood sizes.

Proof. Let CP be the set of core points. We observe that for any core point $|N_{c\cdot\varepsilon}(q)| \geq minPts$. This means $|CP| \cdot p_2^k \leq |N_{c\cdot\varepsilon}(q)|$. As in the proof of Lemma 3.4.4, the expected contribution from a bucket is given by:

$$E[|T_{k,i}(q)|] \leq \#CL + \#MP \cdot p_1^k + \#FP \cdot p_2^k \leq 2 \cdot |N_{c\cdot\varepsilon}(q)|. \tag{3.4.3}$$

Let k_{best} be the best level found by the algorithm. The expected time cost T to carry out the for loop (Line 4 in Algorithm 6) is

$$T \leq 2 \cdot \text{reps}(k_{\text{best}}) \cdot \sum_{q \in CP} |N_{c\cdot\varepsilon}(q)|. \tag{3.4.4}$$

Substituting *minPts* in Lemma 3.4.2 by $|N_{c\cdot\varepsilon}(q)|$ in the proof of Lemma 3.4.2, and observing that for any core point $|N_{c\cdot\varepsilon}(q)| \leq |CP|$, we conclude that setting $\text{reps}(k) = p_1^{-k} \cdot \ln\left(\frac{|CP|^2 \cdot k^2}{\delta}\right)$ ensures all pairs of core points q,p for which $d(q,p) \leq \varepsilon$ are hashed together at least once. Similar to the core points identification step, to ensure $|CP| \cdot p_2^k \leq minPts$ we have $\tilde{O}(p_1^{-k})$ is $\tilde{O}\left(\left(\frac{CP}{minPts}\right)^{\rho}\right)$. □

3.5 Implementation Choices

In this section we highlight implementation-specific choices that we did to speed up the empirical performance of our algorithm.

Details of the hash function. We carry out the empirical analysis under Euclidean distance and use E2LSH [DIIM04] as our LSH function. A single hash function $h : \mathbb{R}^d \to \mathbb{Z}$ takes three parameters $a \in \mathbb{R}^d, r \in \mathbb{R}, b \in \mathbb{R}$. For one hash function, $a \sim N(0,1)^d$ is a d-dimensional vector of standard normal random variables, and b is uniformly chosen in the interval $[0,r)$. We use $r = 4\varepsilon$ as recommended [DIIM04]. The hash value of a point $p \in \mathbb{R}^d$ is $h_{a,b,r}(p) = \lfloor \frac{a \cdot p + b}{r} \rfloor$, where $a \cdot p$ is the inner product of the vectors a and p. Applying this hash function k times yields a tuple of integers $(H_1,\ldots,H_k) \in \mathbb{Z}^k$. We map these hash values to an integer using universal hashing [CW79]: Let $P = 2^{61} - 1$ and let random integers $u_1,\ldots,u_k$ be chosen

uniformly at random from $\{1,...,P-1\}$. Given $(H_1,...,H_k)$, use $\sum_{1\leq i\leq k} H_i u_i \mod P$ as key in the hash table.

Details of the merging step. Algorithm 6 details the merging step. For each core point, we conceptually retrieve all the points in all the buckets on the point's best level and check if these two points should belong to the same cluster. Analogously, the work carried out in a bucket is quadratic in its size. Since we are using a union-find data structure to keep track of the *current clustering*, we carry out the work in a single bucket as follows: Let `uf` be the state of the union-find data structure before inspecting the points in bucket B. First, using the `find` operation on `uf`, split up the points in B into their current clusters. Put all points into an initially empty priority queue PQ and associate with each point the size of its current cluster. While PQ is not empty and there is more than one cluster left, remove the point p with smallest priority and compare it to all points in clusters different than its own. As soon as a point q is found that is at distance at most ε, call `union`(p,q) to merge the two clusterings and stop comparing to other points in q's cluster. Organizing data in this way helps as soon as clusterings evolved while processing different repetitions. Instead of quadratic work in each bucket, the work is at most the product of the individual cluster sizes in a bucket, which can potentially be much smaller.

3.6 Experimental Evaluation

This section reports on the results of our experiments. Our code is written in C++ and compiled using the flags `-std=c++17 -O3 -mavx2`. We base our implementation on the code provided by [KGPT22] and use the same parallelism framework. Our implementation is available at `https://github.com/CamillaOkkels/srrdbscan`.

Experimental setup. Experiments were run on a machine with 2x14 core Intel Xeon E5-2690v4 (2.60 GHz) with 512GB RAM using Ubuntu 20.04.6 LTS. All parts of the code are parallelized by using a concurrent hash table (Intel TBB), and splitting up points into batches. We restrict the amount of RAM that can be used to 100 GB and put a time limit of 10 hours on a run for a single parameter.

Hyperparameter choices. Our implementation requires the user to set a bound on the memory consumption (L) and the failure probability (δ). We restrict the memory usage to 5 GB for all datasets except Gist for which we chose to restrict the memory usage to 15 GB. To compare with other baselines, we use $c=1$ (no approximation) during core point classification,

Table 3.1: Overview of the datasets.

Dataset	#points (n)	dim. (d)	*minPts*	Baseline
Mnist	60 000	784	100	FAISS
Gist	1 000 000	960	20	—
Glove	1 183 514	100	100	FAISS
Aloi	49 534	27	100	FAISS
Census	299 285	500	650	FAISS
Celeba	202 599	39	200	FAISS
Pamap2	2 872 533	4	100	TPE
Household	2 049 280	7	100	TPE

which only puts us at a disadvantage. We choose $\delta \in \{0.01,0.1,0.5,0.9\}$ to control the probability that a core point is missed, cf. Lemma 3.4.2.

Datasets. Table 3.1 summarizes the datasets used for the evaluation. We use two popular low-dimensional datasets (Pamap2 and Household) for clustering [SSE+17] to verify the performance of our algorithm on low-dimensional datasets. Aloi [GBS05, KKSZ11], Census, and Celeba are standard benchmarks in outlier detection [HHH+22]; Aloi has been used in clustering analysis [HKK+10, KSZ11] and Mnist is a standard dataset used for clustering high-dimensional data [KGPT22, XP24]. The two high-dimensional datasets Gist and Glove are standard datasets for high-dimensional nearest neighbor search [ABF20] and are the most challenging in terms of their size.

Reproducibility. We developed benchmarking infrastructure to (i) carry out dataset preparation for common datasets used in cluster analysis, (ii) install and run all implementations in Docker containers, and (iii) evaluate the results and produce the plots. This benchmarking and evaluation environment is available at `https://github.com/CamillaOkkels/dbscan-benchmark`.

Competitors. We compare the proposed implementation (referred to as SRR in the following) to the following baselines: (i) sklearn's DBSCAN[3], (ii) TPEDBSCAN [WGS20], and (iii) SNGDBSCAN [JJL20]. We do not include the projection-based method from [SV17] because [XP24] ruled out its scalability even on small high-dimensional datasets. The former is one of the most widely used implementations which speeds up range queries using a tree index data structure and has been shown [KSZ17b] to be one of the fastest implementations of DBSCAN. The latter two have

[3] `https://scikit-learn.org/stable/modules/clustering.html`

been described in Section 3.2 and present the state-of-the-art for grid- and sampling-based approaches. There are no parameters to set for (i) and (ii), and both are exact. For (iii), the user has to manually set the sampling probability and we used the values $\{0.01,0.05,0.1,0.2\}$. We will refer to the competitors as follows: (i) SKL, (ii) TPE, and (iii) SNG. To cope with high-dimensional datasets, we developed our own exact baseline based on the popular FAISS library [DGD+24], which provides efficient exact nearest neighbor search with high level of parallelism.[4]

We include IP.LSH [KGPT22] in the evaluation, but for the lack of theoretical guarantees omit its results in the head-to-head comparison. A detailed overview of the clustering quality and its running time is provided in Section 3.6.2.

We do not compare directly to the recent paper by Xu and Pham [XP24] since they work on inner product spaces and solve other metric spaces by first embedding—distorting the distance—while we are interested in the comparison to DBSCAN in Euclidean space. We remark that they achieve faster clustering times but lack worst-case guarantees, which is a focus of this paper.

Methodology. We follow the methodology of Schubert et al. [SSE+17]. For each dataset, a single *minPts* value is fixed. Each implementation is run on a collection of ε values that cover the range from "all points are noise" to "all points belong to a single cluster." As noted in [SSE+17] different implementations have different trade-offs regarding the density of the datasets and only testing a wide range of ε values gives an overview of the overall performance of an implementation.

Quality and performance metrics. As quality metric, we measure the *adjusted Rand index* (ARI) [HA85] of the clustering returned by our algorithm given an exact DBSCAN ground-truth clustering. Values close to 0 (1) indicate a poor (strong) agreement between both solutions. As performance metric, we record the total time needed to cluster the data given some $(\varepsilon,\mathit{minPts})$ combination.

Objectives of the experiment. Our experiments are tailored to answer the following questions.

[4]We used their exact index with a bruteforce search as the basic search data structure. In the first step, we search for the *minPts*-nearest neighbors to decide whether a point is a core point. Next, we carry out the brute-force search only on core points to find a value K such that the K-th nearest neighbor of each core point is at most at distance ε, starting from the guess $K = \mathit{minPts}$ and doubling in each step. Based on these neighborhoods, we merge labels and assign border points in a final step.

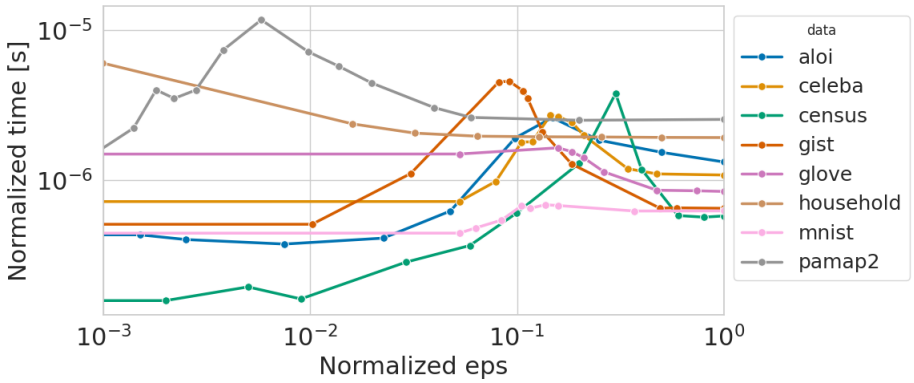
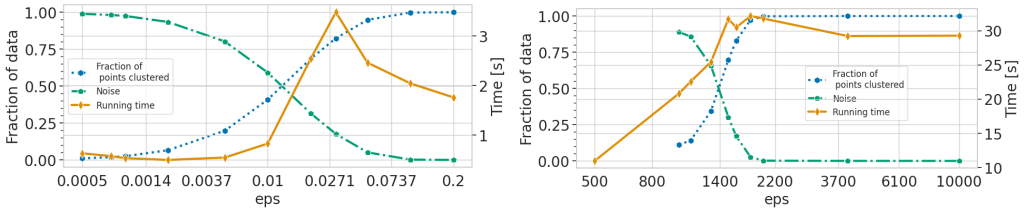


Figure 3.5: Normalized running times with $\delta = 0.1$; x-axis: normalized ε value using $(\varepsilon - \varepsilon_{\text{min}}) / (\varepsilon_{\text{max}} - \varepsilon_{\text{min}})$; y-axis: Running time in seconds normalized by $n \cdot d$.



(a) ALOI (b) MNIST

Figure 3.6: Clustering structure and running time for $\delta = 0.1$.

(Q1) How is the overall running time of the proposed algorithm influenced by the clustering structure of the dataset?
(Q2) How does the overall running time perform in comparison to other existing baselines?
(Q3) What is the clustering quality w.r.t. exact DBSCAN results in practice?
(Q4) How does the proposed multi-level data structure adapt to different choices of ε?
(Q5) How does the approximation factor c influence the running time and result quality?

3.6.1 Experimental Results

Running time results (Q1 & Q2). Figure 3.5 summarizes the running time results for running our implementation for $\delta = 0.1$ across all different choices of datasets and ε-values. Because of the dataset diversity regarding their dimensionality d and their size n, the running times are scaled by $d \cdot n$. This normalizes running times to reflect the amount of distance

Table 3.2: Results of running time experiments. If method is not exact, fastest configuration with ARI at least .7 is displayed. "$-_{HD}$": dimensionality too high to run implementation. This error is due to a safeguard in the implementation of TPE that restricts the dimension of the input; $-_T$ / $-_M$: 10 hours or 100 GB RAM exceeded respectively. (*) means no baseline ($\delta = 0.1$ for SRR is reported). $|\mathscr{C}|$: number of clusters.

	ε	$\lvert\mathscr{C}\rvert$	Time [s]				
			SRR	SNG	TPE	FAISS	SKL
Aloi	0.0008	2	**0.37**	1.95	$-_{HD}$	21.75	7.69
	0.01	243	0.47	**0.46**	$-_{HD}$	300.02	12.84
	0.1	2	1.44	**1.4**	$-_{HD}$	173.44	49.88
Celeba	1.0	2291	**4.35**	15.91	$-_{HD}$	323.04	4369.7
	1.6	51	**7.71**	18.37	$-_{HD}$	1078.9	4727.9
	4.0	1	**5.85**	43.83	$-_{HD}$	1616.0	$-_M$
Census	0.03	37	**21.12**	23.4	$-_{HD}$	3768.1	$-_T$
	1.0	1816	**30.96**	93.78	$-_{HD}$	6357.3	$-_T$
	8.0	2	**62.36**	772.88	$-_{HD}$	11823	$-_M$
Gist	0.3*	794	**486.6**	1205.0	$-_{HD}$	$-_T$	$-_T$
	1.3*	27	**3367**	8224.89	$-_{HD}$	$-_T$	$-_T$
	6.0*	1	**625.7**	15627	$-_{HD}$	$-_T$	$-_M$
Glove	2.0	8	**176.14**	1064.84	$-_{HD}$	7433.47	$-_T$
	5.0*	11	**165.9**	*2036.14*	$-_{HD}$	$-_T$	$-_T$
	20.0	1	**99.22**	4282.37	$-_{HD}$	18487	$-_M$
Mnist	1000	2	14.41	**13.64**	$-_{HD}$	197.51	3201.3
	1600	7	30.49	**20.05**	$-_{HD}$	683.1	3225.7
	4000	1	**25.2**	42.69	$-_{HD}$	492.7	2946.9
Household	0.5	69	56.19	$-_M$	**1.13**	$-_T$	$-_M$
	1.0	63	49.43	$-_M$	**1.43**	$-_T$	$-_M$
	500	1	24.24	$-_M$	**0.64**	15609	$-_M$
Pamap2	0.5	276	15.37	$-_M$	**1.6**	13095	183.86
	3.0	275	80.16	1657.3	**1.49**	$-_T$	319.85
	30.0	3	29.96	3572.3	**1.11**	$-_T$	$-_M$

comparisons that have been carried out by the implementation to find the clustering. To account for different distance distributions in the dataset, ε values are scaled between the minimum and maximum value. From the plot we see that higher dimensionality of the dataset does not increase the difficulty of finding the clustering, with no more than a factor 10 difference in normalized running time despite the diversity of tested datasets. Across all datasets, the low-dimensional datasets Pamap2 and Household take the longest normalized time to obtain a clustering. Each dataset comes

Table 3.3: ARI for different epsilon and failure probabilities.

δ	$\varepsilon = 0.0008$		$\varepsilon = 0.01$		$\varepsilon = 0.1$	
	T [s]	ARI	T [s]	ARI	T [s]	ARI
0.01	0.89	0.9423	1.8	0.9569	2.88	1.0
0.1	0.58	0.9444	0.83	0.9553	2.04	1.0
0.5	0.48	0.9374	0.47	0.9185	1.44	0.9124
0.9	0.37	0.9243	0.44	0.6455	1.04	0.0084

(a) Aloi

δ	$\varepsilon = 1000$		$\varepsilon = 1600$		$\varepsilon = 4000$	
	T [s]	ARI	T [s]	ARI	T [s]	ARI
0.01	36.11	0.9938	43.4	0.9749	25.2	1.0
0.1	20.82	0.9944	30.49	0.8866	29.22	1.0
0.5	14.41	0.9819	17.62	0.5472	14.77	0.0
0.9	10.34	0.6297	13.9	0.0372	13.49	0.0

(b) Mnist

with its own characteristic peak, which poses the question for the relation between the clustering structure and the measured running time (Q1).

Figure 3.6 relates the empirical running time to the clustering structure for two datasets Aloi (medium-dimensional) and Mnist (high-dimensional). We observe distinct phases in the running time for increasing ε-values: When all points are noise (small relative ε), the algorithm has to check each bucket until it confirms that the point is not a core point. In this case, buckets on deeper levels are small since the area is not dense. No work has to be carried out in the merging step. In the other extreme when all points are core points (large relative ε), it is likely that all points hash into the same bucket, i.e., points will immediately be confirmed as core points; the merging step can merge all points into one cluster based on a single data point as center. The task is most difficult to solve in the setting where clusters have emerged, but they do not all belong to the same clustering yet. The trends observed on these datasets translate to the trends observed in all datasets.

Table 3.2 compares the running time achieved by our solution to the other baselines, answering (Q2). We make the following observations. As noticed by other authors as well [XP24], the popular sklearn DBSCAN implementation fails to find a clustering within reasonable memory and time limits. For all datasets except Aloi and Mnist, it fails for some or all ε values, both in terms of running time (e.g., Census) or memory (e.g.,

Pamap2). For the low-dimensional datasets Household and Pamap2, TPEDBSCAN provides the most efficient solution. It finds the clustering in a few seconds for all parameter choices; our implementation takes a second place and finds all clusterings within a minute. All other competitors are slower and fail for some ε values because of exceeding the time restriction (FAISS) or using too much memory (SNGDBSCAN and SKLEARN). TPEDBSCAN is only useful for low-dimensional datasets.

For medium-sized datasets such as Aloi, Celeba, and MNIST, our implementation and SNGDBSCAN perform best. For high-dimensional and million-sized datasets, only our implementation and SNGDBSCAN finds a clustering within the time and memory restrictions, and our implementation is between 2.4x and 40x faster at finding the clusters. Over all datasets and ε values, for all but GIST the clustering is found in less than 4 minutes. All other baselines cannot reliably find a clustering within 150x this time limit (exceeding 10 hours) or 10x the allowed space (exceeding 100GB RAM).

Influence of δ on clustering quality (Q3). Table 3.3 showcases the influence of changing δ in our implementation for the datasets Aloi and Mnist, again representing the spectrum of different dataset dimensionality. Recall from Lemma 3.4.2 that smaller δ increases the repetition count on each level. Fixing an ε value, increasing δ—as expected—improves the running time at cost of lower cluster quality. A δ-value of .5 (50% chance of misclassifying a core point on worst-case input) does not affect the ARI score by much on Aloi, but greatly decreases the clustering quality for Mnist. Since there is no big loss in performance in choosing a small δ-value (running times increased by not more than a factor of 4), these values provide much better result quality at little cost.

Influence of density parameter ε on multi-level data structure (Q4). Figure 3.7 visualizes the contribution of the different algorithmic subroutines (index building, core point identification, cluster merging) to the total running time on Pamap2 for increasing ε values.[5] When clusters emerge (ε increasing from 0.3 to 7), the time to classify core points is first increasing until $\varepsilon = 3$. The running time of finding the clustering is maximal for this value, and this can be contributed largely to the core point identification step. For larger ε, the contribution of this step decreases since it is easier to verify that

[5]The same trends hold for other datasets, but they are most pronounced for Pamap2.

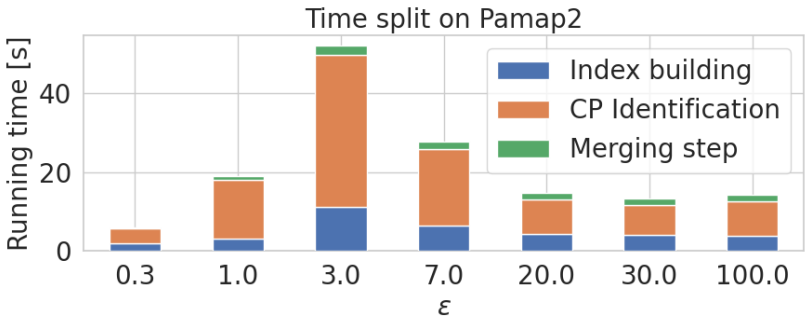


Figure 3.7: Influence of ε on algorithm steps for failure probability $\delta = 0.1$.

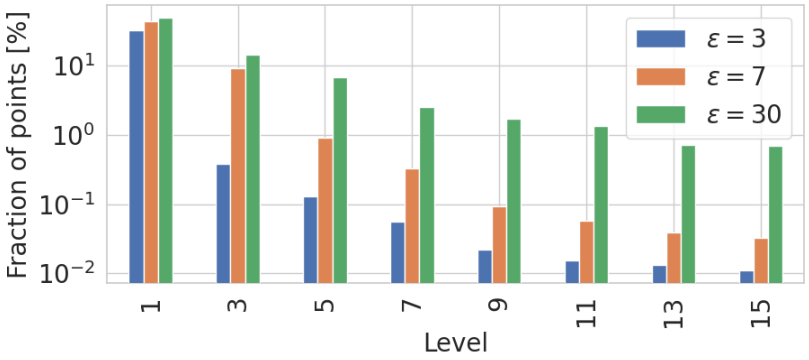


Figure 3.8: Work per level on Pamap2 for $\varepsilon \in \{3,7,30\}$ and $\delta = 0.1$. The y-axis represents the fraction of points that collide with a point, averaged over all dataset points. Note that the y-axis is log-scaled.

a point has at least *minPts* neighbors. In contrast to core point identification, the merging step does not take a significant share of the total running time.

We now focus on the level that is chosen by the proposed data structure. Figure 3.8 shows the work per level for three different ε values from the discussion above, i.e., the value ω_k in Algorithm 4. Again, the distinct density regions of the clustering have a large influence on the multi-level data structure. For $\varepsilon = 3$, clusters have emerged but points can be well-separated on deeper levels in the data structures such as $K = 15$, where only 0.01% of data points are going to be inspected on average to classify a point. For larger values of ε, the hash function puts more points into the same bucket which allows for quick validation of core points. A lower value of K is chosen in such a case. For example, for $\varepsilon = 7$ and $K = 9$, each point collides with roughly 1% of the data points. Setting these numbers in relation to Figure 3.7, the core point classification takes less time in that case as for $K = 15$ for the $\varepsilon = 3$.

Influence of approximation factor (Q5). For the following experiment we vary the approximation factor c of SRR (cf. Line 6 in Algorithm 5). For space reasons, we focus on the ALOI and MNIST dataset. As parameters, we choose $c \in \{1.0, 1.1, 1.5, 2.0\}$ and $\delta = 0.1$. From the definition of an approximate DBSCAN* clustering in Section 3.3, we expect more points to be identified as core points. The impact on running time is less obvious. In the theoretical derivation of the upper bound, cf. Theorem 3.4.3, a larger c value decreases the running time. Of course, this upper bound is on worst-case input, so a higher approximation may very well lead to a higher running time on real-world datasets. In particular, the processing of the greater number of core points may be more time consuming. An unlucky distribution of core points in the buckets and a large number of emerging clusters could make the merging step more inefficient with more points.

Figure 3.9 shows the running time as a function of ε for different approximation factors c. For ALOI, there is little to be gained by changing the approximation factor. For MNIST, the running time improved with the approximation factor for almost all values of ε.

Figure 3.10 illustrates the ARI score of SRR as a function of ε for different approximation factors c. As expected, the overall quality decreases as the approximation increases. An exception is for MNIST, for which a value of $c = 1.1$ is actually better for $\varepsilon \geq 1500$.

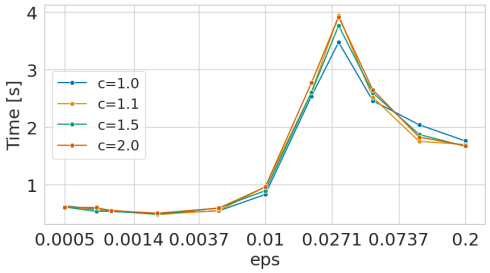
To summarize, we do not observe a big improvement in running time that would compensate for the loss in quality. We thus recommend to use SRR with $c = 1$.

3.6.2 Comparison to IP.LSH.DBSCAN

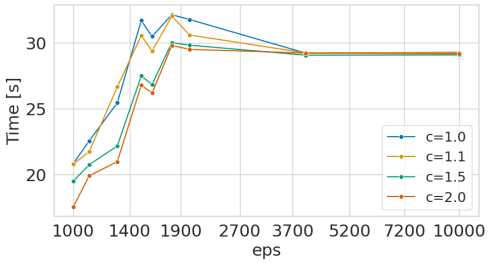
In this section we compare our work empirically to the recently suggested IP.LSH method by Keramatian et al. [KGPT22].

Overview. Given parameters M and L provided by the user (in addition to the DBSCAN parameters), IP.LSH builds an index for the dataset S that consists of L repetitions. In each repetition, M hash functions are concatenated and each point in S is stored in the bucket identified by the M-tuple of individual hash values. This is comparable to building a single level in our data structure with a fixed repetition count that is *not* adjusted automatically to the DBSCAN parameters in combination with the used LSH family.

To identify *core points*, each bucket that contains at least *minPts* many points is checked. The medoid x^* of the points is calculated (the data point

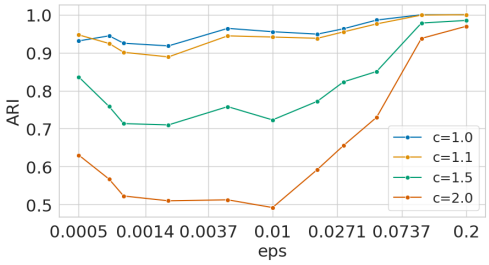


(a) ALOI

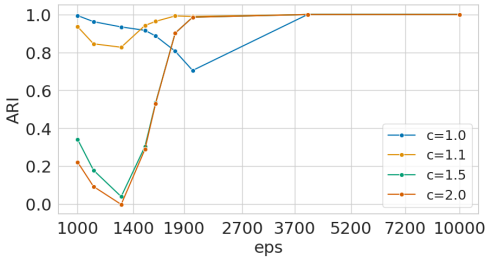


(b) MNIST

Figure 3.9: Running time as a function of ε for the ALOI and PAMAP2 datasets for different values of c.

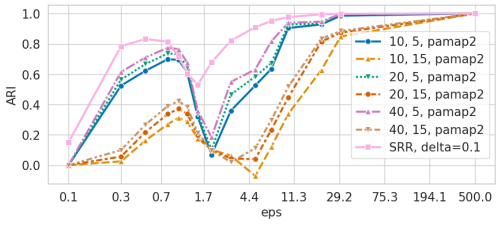


(a) ALOI

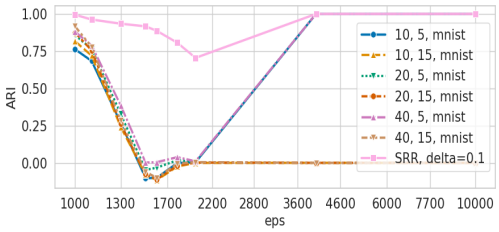


(b) MNIST

Figure 3.10: ARI score as a function of ε for the ALOI and PAMAP2 datasets for different approximation factors c.



(a) PAMAP2



(b) MNIST

Figure 3.11: ARI score for different parameter choices (L,M) for IP.LSH.DBSCAN vs SRRDBSCAN.

closes to the average point of all points that collide in the bucket). This point is marked as a core point if at least *minPts* points are at distance ε in the bucket. All these points at distance at most ε are marked as belonging to the same cluster as x^*. In a second phase, all buckets whose medoid is a core point are inspected again, and distance calculations to other points in the bucket marked as core points, i.e., core points identified in *other buckets*, are carried out. If two points are at distance at most ε, they share a cluster label. As in our implementation, a union-find data structure is employed to keep track of the cluster labeling.

Differences. From a running time perspective, the IP.LSH heuristic alleviates the quadratic time barrier in each bucket; if our implementation and the implementation of IP.LSH would end up using the same level and repetition count, IP.LSH would carry out much less work. However, this improvement comes with great uncertainty about the produced clustering: As stated in [KGPT22, Lemma 2], the probability of correctly identifying a core point is extremely low for IP.LSH. This is because it only identifies a single potential point as core point in each bucket. Moreover, for chains of density-reachable core points to get merged into the same cluster, these core points have to be chosen in *other buckets* to be taken into consideration.

Experimental comparison. To confirm whether the uncertainty about the clustering quality can also be observed in practice, we carry out an experimental evaluation using the code provided with [KGPT22]. We note that our quality results cannot be compared to [KGPT22]: the evaluation provided there focused on the quality of the clustering in terms of predicting known class labels, while we compare to the exact (ε,*minPts*)-DBSCAN clustering for a diverse range of density parameters ε.

For the experiment, we vary $L \in \{10,20,40\}$ and $M \in \{5,10,15\}$. We report on the results for Mnist and Pamap2. As we can see from Figure 3.11, the quality of the resulting clustering varies widely across datasets and ε values. For M equal to 5 and the Mnist dataset, the ARI score is high only for very small density values (almost all points are noise) or very large values (all points belong to the same cluster). In between these values, the quality drops to almost 0. Varying the repetition count L does not alleviate this problem. For $M = 15$ (Figure 3.11(b)), not enough points collide so core points are never identified and the resulting quality is low.

The trend on the Pamap2 dataset is slightly different. It starts at 0 for low values of ε (note that at $\varepsilon = 0.1$, the Pamap2 dataset actually does have clusters) after which it increases to a local maximum, drops again and then finally increases to 1. Pamap2 has many points, so the local maximum can be explained by the fact that clusters are dense, so there are more chances to find the necessary points to form them. When increasing ε, at some point the clusters that emerged are merged together. This too can be dependent on a single core point being identified. This might be an explanation for the drop we observe around $\varepsilon = 1.5$ to $\varepsilon = 5$.

In terms of the measured running time, the *slowest* clustering time achieved by IP.LSH on MNIST was 1.5s. In comparison, the *fastest* time for SRRDBSCAN of all values of ε with $\delta = 0.1$ was 20.8s. Similarly, while

SRRDBSCAN was faster than certain parameter settings of IP.LSH on PAMAP2 for some values of epsilon, the fastest average running time of IP.LSH over all values of epsilon of all parameter settings on PAMAP2 was 2.59s compared an average running time of 48.3s over all ε for SRRDBSCAN with $\delta = 0.1$. IP.LSH finds a clustering much faster, but this has to be seen under the light that the quality with respect to the baseline DBSCAN is unreliable. The comparison is made between the $\delta = 0.1$ run of SRRDBSCAN and the fastest run across the different parameter settings of IP.LSH. As we can see from Tables 3.3a and 3.3b, SRRDBSCAN achieves decent quality for all values of epsilon in this setting, whereas the quality of the IP.LSH results is much more inconsistent.

3.7 Conclusion

In this paper we presented SRRDBSCAN, an LSH-based DBSCAN implementation with strong theoretical guarantees on running time and result quality. Through an extensive empirical evaluation, we showed that our implementation provides robust clusterings where other baselines fail because of problems that stem from the curse of dimensionality (exceeding generous running time bounds), too high memory footprint, or too bad clustering quality.

As future work, it would be interesting to apply our methods for finding hierarchical clusterings such as HDBSCAN [CMS13]. Another interesting research agenda would explore the combination of a sampling-based approach such as [JJL20] with LSH.

Chapter 4

Approximate Single-Linkage Clustering Using Graph-Based Indexes: MST-Based Approaches and Incremental Searchers

First published in: Similarity Search and Applications.
SISAP 2025. Lecture Notes in Computer Science, vol 16134, pp 233–247 by Springer Nature [OTA+26]
- Best paper award nominee, invited to submit an extended version to a special issue of Information Systems
Reproduced with permission from Springer Nature
Joint work with: Erik Thordsen, Martin Aumüller, Arthur Zimek and Erich Schubert

Abstract

Current exact single-linkage clustering algorithms have asymptotically quadratic complexity. We present algorithms for *approximate* single-linkage clustering with empirically near-linear scalability. We explore both graph index-based incremental nearest neighbor search and an iterative exploration scheme on the graph index approximating the MST of the reachability graph similar to Kruskal. As graph index, we use both the bottom layer and a combination of all layers of an HNSW as a stand-in for connected search graphs. We provide experiments comparing the clusterings to baselines such as exact single linkage implementation and an algorithm using metric tree-based searchers. We explore the impact of the HNSW hyperparameters on the performance in terms of running time and clustering quality and evaluate the empirical asymptotic complexity.

4.1 Introduction

Clustering is one of the fundamental data mining tasks aiming at identifying groups in data. What makes good groups depends on the adopted clustering paradigm. Some methods aim at finding compact clusters (typically assuming a given number of clusters), others aim at finding groups of connected points, where the connectivity is a transitive property, allowing for non-compact, arbitrarily shaped clusters. The first group is often named "partitioning", the second "density-based". Particular examples are k-means for a partitioning-based method, and DBSCAN [EKSX96] for a density-based method. While these two categories both result in a set of "flat" clusters, a third category finds hierarchies of clusters, where the clusters could fall in the "partitioning" or in the "density-based" category, and larger clusters can contain smaller clusters (and those in turn even smaller clusters), allowing to mine complex concept hierarchies or studying a dataset at different levels of granularity.

Traditionally, clustering algorithms need to determine distances between points in a metric space, often in the form of neighborhood searches from a given point. In big datasets containing high-dimensional vectors, these operations are often the limiting factor in terms of efficiency and, hence, applicability. Indexing methods, which organize the data points in a way that support neighboring searches, can be employed to facilitate efficient search where needed in the clustering routine [KSZ17b, SSE+17]. However, for challenging datasets, in particular high-dimensional ones, traditional indexing methods would often degenerate and the runtime of the algorithm would become more expensive than a linear scan [WSB98]. While there is little hope [AW15] for exact similarity search with "better-than-linear" worst case performance, researchers have made significant progress on using approximate, "inexact" techniques to speed up neighborhood searches. In particular, the graph-based approach HNSW [MY20] led to a paradigm shift from tree- or hashing-based approaches towards graph-based approaches, see for example the recent survey by Azizi et al. [AEP25].

In the context of clustering, some work has been done to adapt or enhance "flat" clustering methods for handling large datasets efficiently using approximate neighbors [GT17, OATZ25, SV17, XP24]. On the other hand, hierarchical clustering has been more often sped up with other techniques (e.g., parallelization) [WYGS21, dSIN+21, HYS23]. Notable exceptions for a concrete linkage criterion are the hashing-based methods

that were described by Koga et al. [KIW07] (for a single-linkage heuristic) and Abboud et al. [ACH19] for Ward's linkage. The main conceptual challenge of efficient, hierarchical clustering is that all points have to be merged into a single cluster eventually. However, this means that in the beginning close neighbors have to be merged together, while at the end almost unrelated points are the ones that lead to a cluster being merged. Approximate neighbor search methods—if used as a surrogate for the exact indexing methods—can only help with the first step.

In this paper we explore the use of HNSW [MY20] in a whitebox manner (cf. the discussion of blackbox vs. whitebox use of approximate nearest neighbor methods albeit for outlier detection by Okkels et al. [OAZ24]). This means that we operate directly on the approximate neighborhood graph derived by HNSW to derive an approximate hierarchical clustering structure, which is a new direction in approximate hierarchical clustering. The main challenge remains how to use the information present in the graph. We consider three different approaches: one uses the HNSW graph to generate candidate pairs for an incremental searcher as described by Schubert [Sch24], while the other two build a hierarchical clustering directly on top of the graph.

Our contributions can be summarized as follows:

- We initiate the study of graph-based hierarchical clustering and explore the design space of such algorithms. In particular, we propose three hierarchical clustering algorithms based on single-linkage clustering. As a byproduct we propose a novel method for incremental searching a graph-based index.
- We carry out an empirical evaluation of the proposed methods, identifying suitable hyperparameter settings and showing their competitiveness against state-of-the-art baselines.
- We identify advantages and disadvantages of using approximate, graph-based hierarchical clustering methods.

In the following, we discuss technical background and closely related methods (Section 4.2), propose our methods of hierarchical clustering with HNSW (Section 4.3), evaluate the method (Section 4.4), and conclude our findings (Section 4.5).

4.2 Background and Related Work

Let $(\mathcal{X}, d)$ be a point space equipped with a distance measure $d\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+$, for example d-dimensional Euclidean space with $\mathcal{X} := \mathbb{R}^d$ with $d(a,b) := \|a-b\|_2$. Given a dataset $S \subseteq \mathcal{X}$ of N points, the goal of hierarchical clustering is to publish a rooted binary tree that has n leaves which represent the n data points. The children in the subtree rooted at an inner node belong to the same clustering, thus introducing a hierarchy of clusterings. The selection of which nodes to merge is known as the *linkage criterion*. Traditionally, the tree representation is called a *dendrogram*.

4.2.1 Hierarchical Single Link

Hierarchical Agglomerative Clustering is probably the oldest clustering method, whose origins can be traced back to numerical taxonomy in the 1950s [SM+58]. Several variants and algorithms of this method exist, and the basic textbook version can be explained as treating each point as a separate cluster initially, then always merging the two closest clusters. Depending on how we measure the distance of clusters, we can obtain very different behavior. In this work, we will focus on *single-linkage clustering*, where the distance of two clusters C_A and C_B is defined as $\min_{a \in C_A, b \in C_B} d(a,b)$: intuitively, the distance is defined by the single shortest edge between the two clusters.

Single-linkage clustering is effectively equivalent to finding the minimum spanning tree (MST) in the complete graph (S,E,w) induced by the dataset S, where w represents the distance between a pair of points. The early method SLINK [Sib73] runs in quadratic time and linear memory, and closely resembles Prim's algorithm for the minimum spanning tree. Another classic algorithm for MST is Kruskal's algorithm [Kru56]. In classic Kruskal, all edges are inserted into a heap, ordered by increasing edge length; then we repeatedly poll the next shortest edge and merge these clusters with a union-find data structure. The complexity of the algorithm is $O(E \log E)$ where E is the number of edges. For a fully connected graph, this hence is $O(N^2 \log N)$.

4.2.2 Hierarchical Clustering using Incremental Neighbor Search

For hierarchical clustering with incremental similarity search, Schubert [Sch24] introduced multiple algorithm variants. For this work, we

focus solely on single-linkage clustering, and specifically on the Heap-of-Searchers Single-Link (HSSL) algorithm. While Schubert [Sch24] used a vantage point tree (VP-tree) for exact search, we will investigate approximate clustering with the HNSW graph. For the algorithms, we build on the HNSW graph framework created and explored by Thordsen et al. [TS25].

Similar to Kruskal's algorithm discussed above, the idea of this algorithm is to try to enumerate all edges in increasing length. Instead of using one large heap, HSSL uses a heap of *incremental searchers*. For each point we create a searcher to enumerate its neighbors. After finding the first nearest neighbor, the searcher is inserted into the heap with this distance as key. We then pull from the heap the current nearest neighbor, and insert the edge into a union-find data structure, advance the searcher to the next nearest neighbor and insert it back into the heap. The efficiency of this approach depends very much on how much work we can avoid in incremental nearest neighbor search. A searcher could be implemented by computing all distances to the query point and putting them into a heap, but this clearly would yield an $O(N^2 \log N)$ run-time. Specifically, it is desirable to skip computing distances where (A) the points are already in the same cluster due to earlier cluster merges, or (B) the distances are larger than the longest edge of the minimum spanning tree. A tree-based similarity search structure such as the VP-tree used by Schubert [Sch24] can help here, as lower bounds to subsets of the data help in postponing distance computations to these sets until we find no shorter edges. Furthermore, we may be able to skip computing distances of points that are already in the same cluster due to earlier merges.

4.2.3 Hierarchical Navigable Small World Graphs

Hierarchical Navigable Small World (HNSW) [MY20] is a graph-based indexing structure for approximate nearest neighbor search. Given a dataset $S \subseteq \mathcal{X}$ and parameters M,*efC*, a hierarchical graph $G = (V,E)$ is built. On each layer of the graph, a point is represented by a vertex with edges to a diverse set of nearest neighbors. The parameter M determines the maximum number of neighbors per node, and *efC* limits the width of the beam search during construction, see *ef* below. The graph structure is constructed hierarchically by adding points probabilistically to the layers with higher layers having lower probability of points appearing.

A search in this graph is carried out in the following way: starting at a node in the top layer, the graph is traversed in a greedy fashion, moving

to neighbors that are closer to the query point. When no neighbors are found to be closer to the query, the search moves to the next layer down. This is repeated until the search terminates on the bottom layer and the nearest points found can be returned. Instead of making progress only for a single point, the queue of potential points is restricted to *ef* points, known as the *width of the beam*. HNSW builds the graph incrementally by inserting each point at a time. Inserting a point into the graph works by searching for the point to be inserted using the standard search algorithm, and choosing a diverse set of neighbors based on the nodes explored by the search, which is known as the *neighborhood selection rule*.

4.3 Single-Link Clustering with HNSW

In the following we discuss three approaches to perform single-linkage clustering using the HNSW graph. All approaches will start by constructing an HNSW-based index for a given set of hyperparameters. The running time of finding the hierarchical clustering is thus the sum of index building time and running the clustering approach on the given index. In the following, we will assume that the graph is built. Note that the HNSW graph is not necessarily connected, although the "long edges" in the upper layers help. These exist because the set of near points on higher levels represent a small sample of the dataset; close neighbors are very likely to be actually far away. If the graph is not connected, we can enforce connectivity by sampling one point from each component and computing their distances. From a clustering perspective, we do want these components to be separated, and hence finding a suboptimal long edge does not negatively affect the clustering result.

4.3.1 The baseline: Minimum Spanning Tree on HNSW

As baseline approach, we simply compute the minimum spanning tree (MST) of all the edges in the HNSW graph. Given that we limit the number of neighbors of each point in this graph to be at most a small constant $M = O(1)$, it is easy to see that the we have $O(N)$ edges. Given the HNSW index, finding the MST takes time $O(N \log N)$ and represents a single-linkage-based, approximate clustering. We stress that the MST of the HNSW graph is not the MST of the complete graph induced by the dataset as discussed in Section 4.2.2.

Algorithm 7: HNSW-Kruskal(S, $\mathcal{G}$, *minPts*, *uf*)

Input: S – data; $\mathcal{G}$ – HNSW graph;
Output: Dendrogram data

```
1  H ← Min heap containing all edges from G with distance as weight
2  G_0 ← bottom layer of G
   /* Iterate over MST
      edges of reachability graph in ascending order */
3  while not all points are in the same cluster do
4  |  d_ij, (i,j) ← H.pop()
5  |  if i and j are in the same cluster then continue
6  |  Add edge i, j, d_ij to the Dendrogram.
7  |  foreach neighbor k of j in G_0 do
8  |  |  if i and k are in different clusters then H.push(d(s_i,s_k),(i,k))
9  |  foreach neighbor k of i in G_0 do
10 |  |  if j and k are in different clusters then H.push(d(s_j,s_k),(j,k))
11 return Dendrogram of merge information
```

4.3.2 Kruskal-style Hierarchical Clustering

To improve over the baseline, Algorithm 7 describes how to heuristically simulate running Kruskal's algorithm on the complete graph induced by the dataset. To run Kruskal's algorithm, we would need access to all $O(N^2)$ edges sorted by their distance but we want to avoid materializing all edges. We approach it like this: Initially, we begin with only the edges of the bottom layer of the HNSW graph, which only stores $O(N)$ edges, as in the baseline approach. To improve the clustering quality, when we poll an edge (a,b) from the graph, we insert new edges with one additional step in the HNSW graph: let N_a and N_b be the neighbors of a and b, then we add edges $\forall_{n_b \in N_b}(a,n_b)$ and $\forall_{n_a \in N_a}(n_a,b)$ to the graph if (1) these points are not yet connected, (2) the edge has not previously been added to the heap.

The running time of the algorithm is clearly not as straight-forward to discuss as the baseline algorithm. While initially there are only $O(N)$ edges, we might carry out the while loop without making any progress for pairs of points which are close to each other but have already been merged. Thus it will be interesting to evaluate the difference in running time and quality between the baseline and this Kruskal-style variant. The implementation of the algorithm comes with subtle challenges. Because of cliques in the graph, adding edges to all neighbors of the other point can easily cause a lot of duplicate edges to be added. Hence we need to keep track of

Algorithm 8: HNSW-HSSL(S, $\mathcal{G}$, efS, uf)

Input: S – data; $\mathcal{G}$ – HNSW graph; ef – number of candidates to keep; uf – UnionFind-data structure
Output: Dendrogram data

```
1  S ← {}, H ← {}                    // searchers and primary heap
2  foreach i ∈ S do
3  |   searcher
   |     ← HNSWSearcher(i, G, efS, uf)      // See Algorithm 9
4  |   S[i] ← searcher
5  |   H.push(searcher.peek(), p)
6  while not everything merged do
7  |   d_ij, i ← H.pop()
8  |   d_ij, j ← S[i].advance()
9  |   if not uf.connected(q,j) then
10 |   |   uf.merge(i,j)
11 |   |   Dendrogram ← (i, j, d_ij)
12 |   H.push(S[i].peek(), i)
13 return Dendrogram
```

which edges we have already explored. But in the worst case, we may have $O(N^2)$ edges and a simple hash set of all edges may already need a lot of memory. We can reduce this memory overhead by almost a factor of two if we use an array of hash sets where the smaller node id is the array index, and the larger node id is stored in the set. When running out of memory, it may be possible to add a cleanup step where all edges are removed from these maps that are connected in the union-find data structure.

4.3.3 Incremental Heap of Searchers using HNSW

Our final proposal is to implement the incremental searcher architecture HSSL that was considered by Schubert [Sch24] with HNSW. Algorithm 8 presents the overview of the incremental search approach, which can be viewed as a vertex-centered version of Kruskal's algorithm. At the start, we initialize an incremental searcher for each point in the dataset. In rounds, we take the smallest edge that is present in any of the N searchers and add it to the clustering (if the edge does not connect two points which are already clustered together). Our novel contribution is the design of an approximate incremental searcher data structure for HNSW, which might

Algorithm 9: HNSWSearcher(q, $\mathcal{G}$, ef, uf)

Input: q – data point; $\mathcal{G}$ – HNSW graph; ef – number of candidates to keep; uf – UnionFind-data structure
Output: Generator of candidate neighbors

```
1  E ← MinPrioQueue()                                  // expand queue
2  C ← MinPrioQueue()                                  // candidate queue
3  V ← {}                                              // visited set
4  E.push(0, q)                                        // starting vertex
   /* generator function
      to find neighbors in approx.  ascending order  */
5  while E is not empty do
6  |  (d,i) ← E.pop()
7  |  N ← G.get_neighbors(i)
8  |  foreach j ∈ N do
9  |  |  if not uf.connected(q,j) and j ∉ V then
10 |  |  |  d_ij ← d(i,j)     // compute distance to candidate
11 |  |  |  C.push(d_ij,j)          // add to candidate queue
12 |  |  |  E.push(d_ij,j)          // add to expansion queue
13 |  |  |  V.add(j)          // prevent duplicates in queue
14 |  while |C| > ef do
15 |  |  yield C.pop()      // return candidate and suspend
16 while C not empty do
17 |  yield C.pop()          // return remaining candidates
```

be of independent interest. Our pseudocode of the incremental searcher uses the Python *generator pattern* and is presented as Algorithm 9. When `next()` is called on this generator, it runs until the next call to `yield`, then execution is suspended. The main idea is to keep a buffer C of at least ef candidate neighbors, similar to a beam search for kNN in graph indexes, and a priority queue E of vertices to expand next. In contrast to a kNN search, we cannot truncate these sets. While we have less than ef candidates, we pop the next vertex from the expansion queue E, and for any neighbor we check that (1) we have not visited it already, (2) it is not yet connected in our union-find structure. We then compute the exact distance and enqueue it both in the candidate queue C and the expansion queue E. If we have at least ef candidates, the current best candidate is returned and removed from the heap.

From the three proposed methods, the HSSL variant uses HNSW's search method almost in a black-box method and should thus—given

Table 4.1: Overview over Datasets.

Dataset	Min. sample size	Max. sample size	dim. (d)
ALOI	50 000	110 000	63
MNIST	10 000	70 000	784

"good hyperparameter" values—accurately represent the neighborhood of a data point. However, it should also be the slowest method since it will by design enumerate close neighbors before far away neighbors, which might become a bottleneck in the final process of merging (far away) clusters.

4.4 Evaluation

Implementation Details and Experimental Setup. Experiments were run on a machine with 2x14 core Intel Xeon E5-2690v4 (2.60 GHz) with 512GB RAM using Ubuntu 20.04.6 LTS. The implementation was split into a backend HNSW implementation using the Rust programming language that exposes Python bindings. These Python bindings are used in the experimental pipeline. The incremental searcher (Algorithm 9) are implemented in Rust, but the merging logic in Algorithm 8 uses a pure Python solution. The part of the implementation in rust is from the framework by Thordsen et al. [TS25]. The core implementation and the benchmarking framework can be found at `https://github.com/CamillaOkkels/HSSL` and `https://github.com/CamillaOkkels/singleLinkage-benchmark` respectively.

Datasets. Table 4.1 summarizes the datasets that we have used to carry out the evaluation. We focus our study on real-world data using the ALOI and MNIST datasets, which have been used in cluster analysis before [OATZ25]. To measure scalability of the proposed implementations, we subsample the datasets, always ensuring that a smaller dataset is fully contained in a larger sample.

Evaluation criteria. As a performance measurement, we measure the overall time it took to compute the hierarchical clustering. To measure the quality of the clustering, we compare the produced approximate clustering to an exact baseline clustering. Given two dendrograms, we first measure the cophenetic distances [SR62] for each dendogram, which measure the minimum height of the dendrogram needed to connect two points. Our quality measure is the Pearson correlation between these distance matrices,

which we call *Cophenetic correlation*. Since the distance matrix contains $O(N^2)$ entries, we compute the correlation on a subsample of 10^7 pairs. Experiments using traditional metrics such as the adjusted Rand index (ARI) require an additional step to cut the dendrogram into partitions that introduces noticeable instability, while also offering only a limited capability to differentiate results as distances are not used by this measure.

Baselines. We compare our implementations against the industry-standard single-linkage implementation in the `SciPy` library [VGO+19]. As a direct competitor based on incremental searching, we use the VPTree-based implementation by Schubert [Sch24].

Hyperparameter settings. All three implementation used *max_build_heap_size* (*efC*) and *lowest_max_degree* (*M*) for the HNSW graph construction. The HNSW-HSSL method further took an additional parameter *ef* (*efS*). We let *efS* take values {5,11,22,47,100}, we let *efC* take values {25,42,71,119,200} and we let *M* take values: {14,26,51,100}. All the values were chosen to span a large interval, and to be evenly spaced in log-space. Testing of *M* for smaller values suggested that a minimum value of around 14 ensured the HNSW graph would be connected - with very few exceptions due to their random nature.

Key findings. We summarize our key findings as follows and will detail them in the following subsections.

1. There is a clear difference between the three proposed methods in terms of running time and achieved quality. The MST baseline is fastest but sensitive to hyperparameter choices, whereas the HSSL variant achieves the best quality with robust parameter selection. The difference in speed between MST and HSSL can be up to a factor 100x.

2. The additional work done in the Kruskal variant (Algorithm 7) does not yield big improvements in clustering quality over the MST baseline.

3. Both Kruskal and HSSL do not exhibit worst-case quadratic scaling on the tested datasets.

4. All three variants outperform the industry-standard `SciPy` baseline.

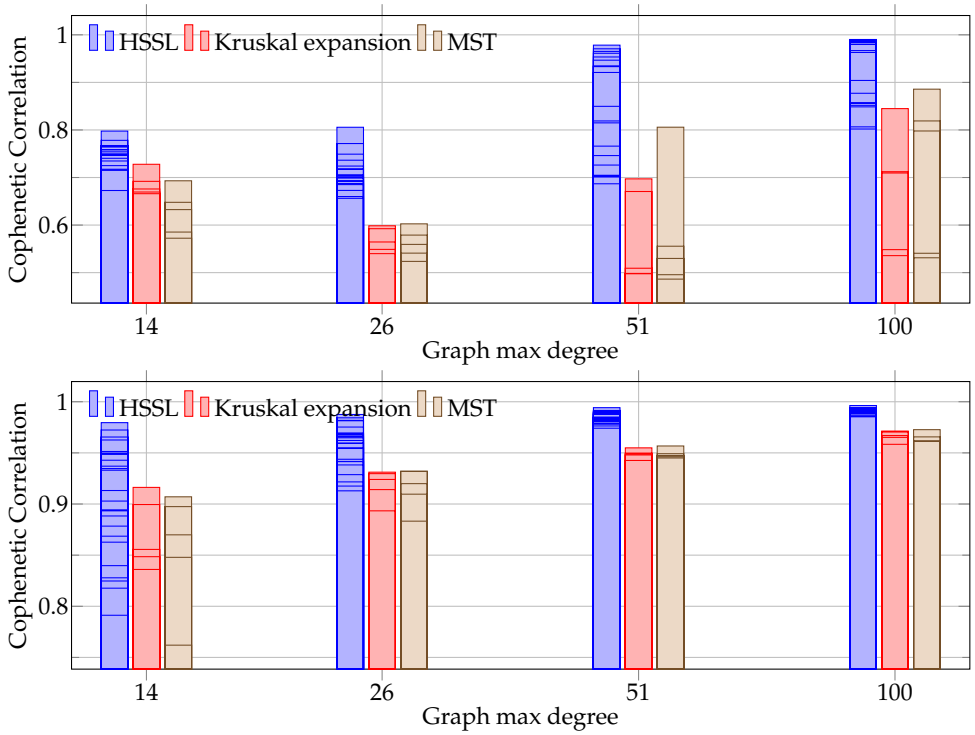


Figure 4.1: Cophenetic correlation on ALOI (top) and MNIST (bottom). Each horizontal line in the bars represents the quality achieved for a certain set of parameters (for Kruskal and MST, a single *efC*, for HSSL a pair (*efC*, *efS*)).

4.4.1 Internal Evaluation

Accuracy. Figure 4.1 shows a bar plot comparing the accuracy of the different methods for varying values of M. Each thick horizontal line in a bar represents the quality achieved for a certain set of parameters (for Kruskal and MST, a single *efC*, for HSSL a pair (*efC*, *efS*)). For each variant, a cophenetic correlation of at least .8 can be achieved by a careful selection of hyperparameters. In general, it is more difficult for our variants to achieve good quality on ALOI, compared to the MNIST dataset. We observe that the accuracy of HSSL increases with increasing M on both ALOI and MNIST. For both datasets and M choices, it outperforms the other two approaches regarding the best achievable clustering quality. The Kruskal and MST methods had increasing quality with increasing M for MNIST, but for ALOI we observed a slight decrease, until a slight increase again between $M = 51$ and $M = 100$. Interestingly, the baseline MST approach shows comparable performance, even outperfoming Kruskal

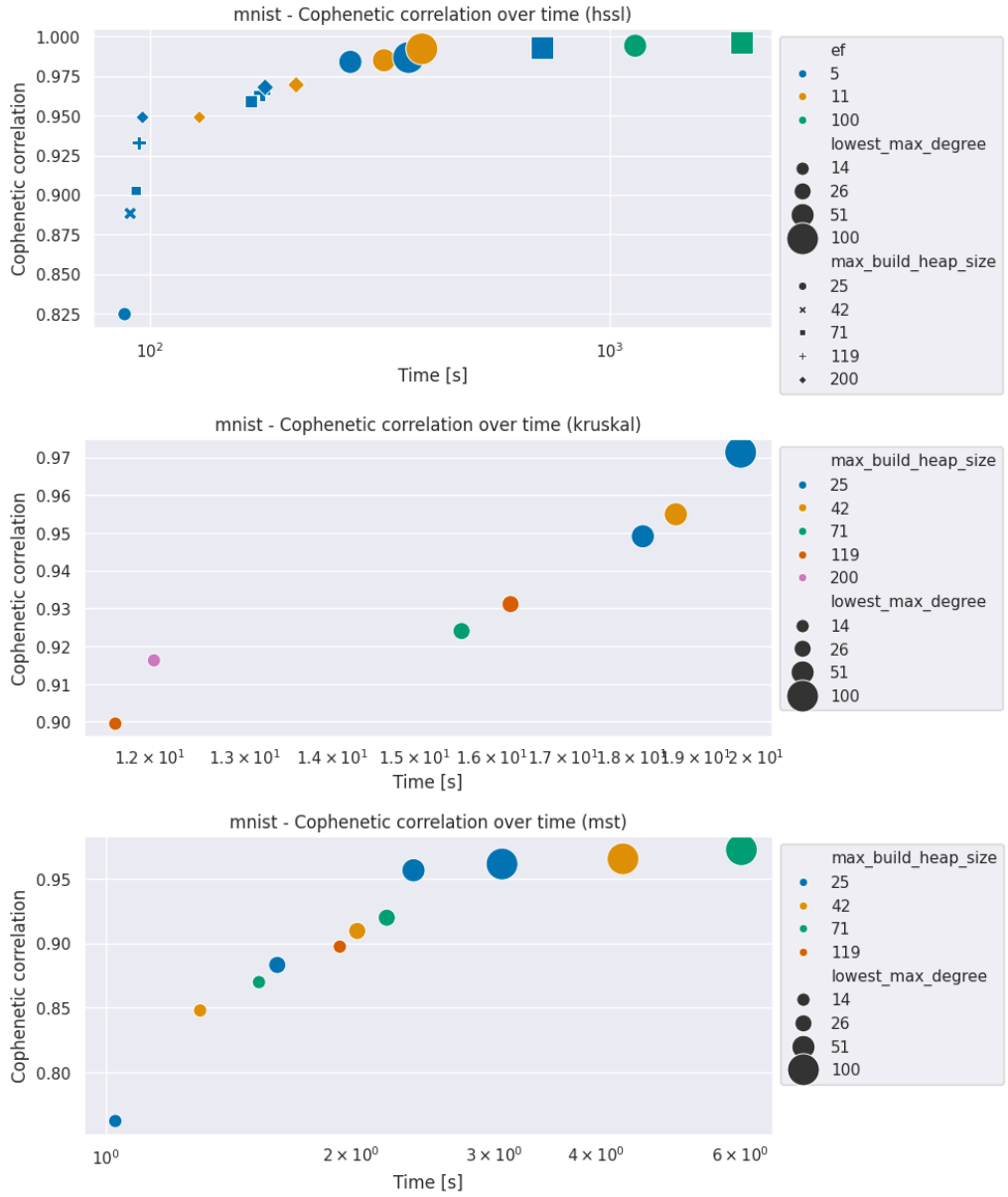


Figure 4.2: Pareto frontier of optimal cophenetic correlation scores on MNIST for HNSW-HSSL (top), HNSW-Kruskal (middle), HNSW-MST (bottom).

for some parameter choices. However, in particular on the ALOI dataset, the MST-based approach is more sensitive to hyperparameter choices.

Running time. Figure 4.2 plots the performance of the individual methods in relation to the achieved clustering quality. Due to space

constraints, we focus the discussion on the MNIST dataset and plot only the Pareto frontier of the measurements. The results that we measured on ALOI were similar.

We observe that for HNSW-HSSL, a larger M value allows for more accurate clustering quality. However, even the smallest value resulted in a clustering quality above .9 while leading to an efficient clustering algorithm. Getting from a clustering quality of .9 to close to 1 increases the running time by almost a factor of 10. The search parameter *efS* has little influence on the clustering quality and good results are achievable with the minimum setting of 5. This is helpful in terms of the robustness of the algorithm, since HNSW-HSSL comes with an additional search parameter that is missing in the other two variants.

For the other two variants, we again observe the trend that increasing M improved quality at the cost of slower run times on MNIST. The *max_build_heap_size* (*efC*) parameter lead to smaller variations in the clustering quality at the expense of running time.

Comparing all three plots, we notice major differences in running time. A clustering quality of at least .9 can be achieved in around 2 seconds (MST), 12 seconds (Kruskal), and 100 seconds (HSSL).

While Figure 4.2 visualizes the Pareto frontier and gives interesting observations over the hyperparameter choices, Figure 4.3 provides a scatter plot over all possible hyperparameter choices and relates measured running time to clustering quality. The plot reinforces the observations that each variant provides a very distinct performance tradeoff, i.e., the slowest parameter settings for MST are faster than the fastest for Kruskal, and similarily between Kruskal and HSSL. Quality-wise we observe that HSSL provides better quality for most of the hyperparameter choices. In particular the MST algorithm requires careful hyperparameter selection. For example, for ALOI there exist hyperparameters that require more time and provide poorer quality.

4.4.2 Comparison to other baselines

We now turn our focus to evaluate the three proposed methods against SciPy's single linkage clustering algorithm, and the VPTree incremental searcher by Schubert [Sch24]. For these experiments, we subsample the datasets to carry out a scalability analysis. For each sample size, we use the same parameter choices for the individual method. For all three methods the parameters are chosen as the fastest settings above a 0.8 cophenetic correlation.

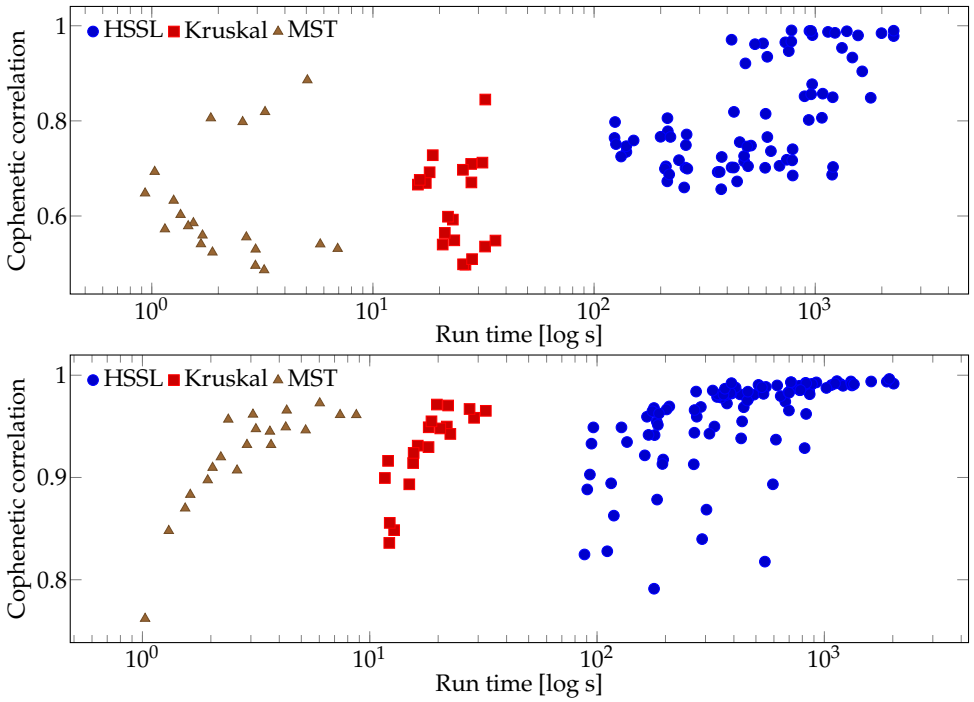


Figure 4.3: Run time vs. approximation quality on ALOI (top) and MNIST (bottom)

Figure 4.4 relates our three proposed methods to the competitors. To put the running times into context, we empirically fit regression lines to the observed running times. These running times are visible in the label of each method in the legend. Comparing to the industry-standard from `SciPy`, all of the proposed methods are faster. The difference is more pronounced for the MNIST dataset than for the ALOI dataset. Comparing the two implementations based on incremental searchers (`HNSWHSSL` and `VPTree`), we notice an interesting tradeoff in their performance: while the (exact) VPTree is faster on ALOI, it is slower on MNIST.

4.5 Conclusion

With this paper we initiated the study of efficient and accurate hierarchical clustering using HNSW as a baseline graph index. We proposed three methods: Two based on computing an MST in the HNSW graph, one based on the idea of incremental searchers. The evaluation highlighted advantages and limitations of the proposed methods. Interestingly, the most basic baseline of computing an MST of the HNSW graph can lead to a good

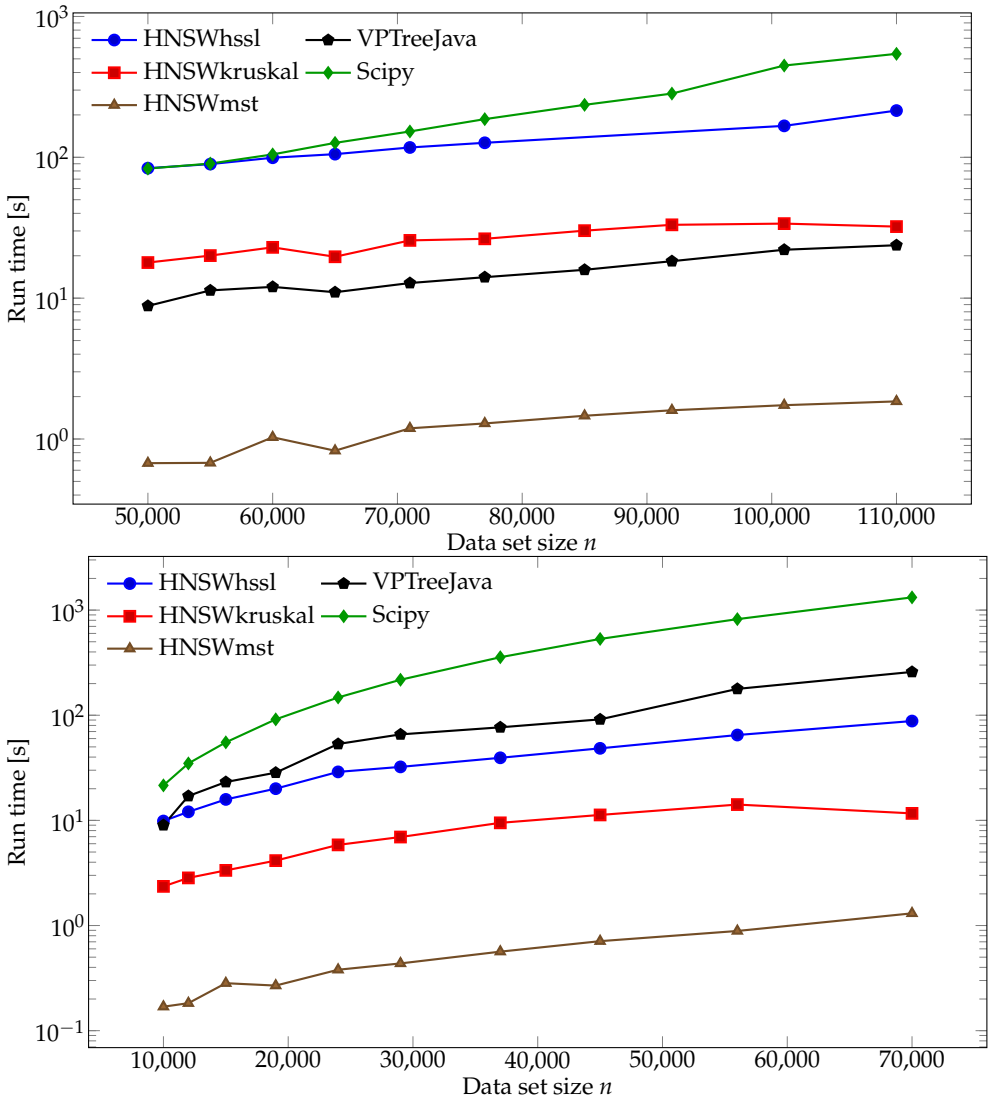


Figure 4.4: Running time vs. sample size on ALOI (top) and MNIST (bottom)

hierarchical clustering on the tested datasets. If good quality needs to be ensured, the incremental searcher-based approach showed its advantages.

While we believe that we cover much of the design space of hierarchical clustering using graph-based approaches, there is much room for improvements. Our most intriguing question is whether we can enhance the graph building progress to build an augmented graph that contains information necessary to speed up hierarchical clustering. From a theoretical point, it would be nice to understand the theoretical limitations of the proposed methods. While we did not observe quadratic scaling on real-world data,

constructing such input instances would be an important achievement for a theoretical understanding of the proposed methods.

4.6 Current work towards future extension

This section was not part of the original publication, but explores the work currently in progress and is part of the ongoing collaboration to extend the paper for the special issue of Information Systems. Section 4.6.1 and 4.6.2 outlines ideas we are currently working on developing and implementing to add to the experimental framework for evaluation.

4.6.1 HDBSCAN

HDBSCAN is another widely used hierarchical clustering method where merges happen based on a density parameter; *minPts*. Part of our current work revolves around extending our framework to include implementations for this additional hierarchical clustering. HDBSCAN is a quite natural extension from Single-Linkage, as it can be constructed very similarly. The difference being the MST edges relevant in HDBSCAN are from another underlying graph - the *mutual reachability graph*. This is the complete graph using *mutual reachability distance* as edge weights, let's denote it $\mathcal{G}_{mreach}$.

Define the *core distance* of a point x for a given parameter *minPts* to be the distance to the *minPts*'th nearest neighbor - denote it $core_{minPts}(x)$. The *mutual reachability distance* between two points x and y is then defined as:

$$d_{mreach} = max(core_{minPts}(x),\ core_{minPts}(y),\ d(x,y)) \tag{4.6.1}$$

One way to approximate the edges in the MST of $\mathcal{G}_{mreach}$ is the following: process the edges of HNSW smallest to largest, but count how many neighbors each point finds - i.e. how many times each point is part of an edge-processing - and don't merge immediately. Once a point finds *minPts* neighbors, mark it as a core point and merge it with all the neighboring points seen before that are *also* marked as core points. An essential parameter of this approximation is the degree of the nodes in the HNSW graph. A point needs to have a degree of *at least minPts* in order to be clustered and for the algorithm to produce a full dendrogram. A method could be to adapt the Kruskal algorithm and the mechanism with which it re-introduces edges to the HNSW graph.

4.6.2 Fix for disconnected graphs

The algorithms presented previously had the issue that when working on an HNSW graph with disconnected components, it would either give an error or produce an incomplete dendrogram. Current work is focused on introducing two types of fixes - one for MST and Kruskal, and another for HSSL.

MST and Kruskal process the edges of HNSW one-by-one, this means we can simply add random edges that connect the disconnected components. This can be done by running BFS (Breadth-First Search) on the HNSW graph and using a union-find to keep track of the elements of the components. Then edges can be sampled randomly between each pair of disconnected components. These edges can then be simply added to the priority queue.

HSSL works slightly differently. As opposed to the fix for Kruskal and MST where the new edges could be added *before* running the rest of the algorithm, a simple fix for HSSL would run after the search has terminated in the case where the dendrogram is not complete, but the searchers are all emptied. From there the existing union-find could be used to identify a pair of components from which random edges that connect them can be sampled. Choosing the smallest edge the components can be merged and the union-find updated. If there are still multiple components in the union-find, the process can be repeated.

Chapter 5

Conclusion and open problems

To round off the thesis we will summarize the key results from each chapter and the questions left open that can guide the direction for future work.

In Chapter 2 we introduced blackbox (kNN-BB, LOF-BB) and white-box (kNN-WB, LOF-WB) graph-based outlier detection methods. The white-box methods showed much more variability in AUCROC score than the black-box methods. We did not from these experiments observe a statistically significant difference in quality between the methods, but the black-box and white-box methods provided great speed-ups. We were able to run our proposed outlier methods in under a minute on the million-scale Kitsune datasets, which kNN and LOF could not finish in several hours. We also observed the impact of recall on the quality of the outlier detection method was dependent on the dataset. For some dataset, a higher recall produced higher accuracy, but for other datasets a lower recall actually seemed to result in lower accuracy. These results motivate the design of algorithms that use the structure of ANN techniques more directly instead of simply replacing the exact neighborhood with the approximate. For future work it would be of great interest to explore some of the following points also highlighted in section 2.6:

- The impact of the fixed parameters k and efC. Particularly the optimal k for an NN-based outlier detection approach can vary greatly between datasets. In general finding dataset attributes to guide towards good parameter choices would be valuable.

- Improving build-time of HNSW graphs (maybe by decreasing *efC*, see the first point) as we found this to be the greatest contributor to the running time.
- Our white-box approaches only used the ground-layer of the HNSW graph, so a natural extension would also be to design an implementation of an outlier detection method that actually utilizes the hierarchical structure.

Chapter 3 explored the use of a multi-level LSH data structure for DBSCAN clustering in an algorithm named SRRDBSCAN. We derived the following bound on the running time:

$$\tilde{O}(minPts^{1-1/c^2} n^{1+1/c^2} d)$$

(see theorem 3.4.3 for the full proof). Theorem 3.3.4 and 3.4.1 constitute the bound on accuracy, which states with probability $1-\delta$ all core points found by the algorithm are c-approximate core points, and all core points in $(\varepsilon, minPts)$-DBSCAN* are identified. We showed experimentally it had competitive performance compared to other state-of-the-art algorithms for DBSCAN clustering; SNGDBSCAN, TPEDBSCAN, FAISS and SKLEARN, and it was more stable in quality than the other LSH-based implementation we compared to - IP.LSH.DBSCAN.

The algorithms discussed in chapter 4 showed a clear indication of a trade-off between the amount of work and the quality achieved. The algorithm based on heap-of-searchers incremental search on the HNSW graph achieved good quality on more of the parameter settings, but was slower than both the MST-based approaches. All three outperformed the industry-standard SciPy implementation on the datasets tested in experiments. One limitation of the implementations as discussed in section 4.6.2 is they rely on the HNSW graph being fully connected to produce a complete dendrogram. The section outlines a potential fix to make the algorithms robust, and it would be valuable to test the performance of the algorithms with this addition. Since smaller values of *efC* and M give the HNSW graph a higher chance of being disconnected, making the algorithms robust to this would allow extending experiments to cover this part of the HNSW parameter space. This has the potential of increasing the speed-up of the algorithm, provided not too much quality is lost. Furthermore single-linkage can be naturally extended to HDBSCAN as outlined in section 4.6.1. It would

be very natural to extend the algorithms to HDBSCAN clustering. The section outlines an idea for an approximate HDBSCAN implementation that would be very interesting to explore experimentally.

With these results we have now showed the versatility and usefulness of ANN search techniques for designing scalable algorithms for both outlier detection and clustering.

Bibliography

[AAP17] Thomas D. Ahle, Martin Aumüller, and Rasmus Pagh. Parameter-free locality sensitive hashing for spherical range reporting. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, jan 2017.

[ABF20] Martin Aumüller, Erik Bernhardsson, and Alexander John Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Inf. Syst.*, 87, 2020.

[ABK+07] Elke Achtert, Christian Böhm, Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. Robust, complete, and efficient correlation clustering. In *Proceedings of the Seventh SIAM International Conference on Data Mining, April 26-28, 2007, Minneapolis, Minnesota, USA*, pages 413–418. SIAM, 2007.

[ABKS99] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, page 49–60, New York, NY, USA, 1999. Association for Computing Machinery.

[ACH19] Amir Abboud, Vincent Cohen-Addad, and Hussein Houdrouge. Subquadratic high-dimensional hierarchical clustering. In *NeurIPS*, pages 11576–11586, 2019.

[AEP25] Ilias Azizi, Karima Echihabi, and Themis Palpanas. Graph-based vector search: An experimental evaluation of the state-of-the-art. *Proc. ACM Manag. Data*, 3(1):43:1–43:31, 2025.

[AF09] F. Angiulli and F. Fassetti. DOLPHIN: an efficient algorithm for mining distance-based outliers in very large datasets. *ACM TKDD*, 3(1):4:1–57, 2009.

[Ahl20] Thomas Dybdahl Ahle. On the problem of p_1^{-1} in locality-sensitive hashing. In *SISAP*, volume 12440 of *Lecture Notes in Computer Science*, pages 85–93. Springer, 2020.

[AI08] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[AIL+15] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya P. Razenshteyn, and Ludwig Schmidt. Practical and optimal LSH for angular distance. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 1225–1233, 2015.

[AP05] F. Angiulli and C. Pizzuti. Outlier mining in large high-dimensional data sets. *IEEE TKDE*, 17(2):203–215, 2005.

[AW15] Josh Alman and Ryan Williams. Probabilistic polynomials and hamming nearest neighbors. In *FOCS*, pages 136–150. IEEE Computer Society, 2015.

[BGRS99] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *ICDT*, volume 1540 of *Lecture Notes in Computer Science*, pages 217–235. Springer, 1999.

[BKKK04] Christian Böhm, Karin Kailing, Hans-Peter Kriegel, and Peer Kröger. Density connected clustering with local subspace preferences. In *Proceedings of the 4th IEEE International Conference on Data Mining (ICDM 2004), 1-4 November 2004, Brighton, UK*, pages 27–34. IEEE Computer Society, 2004.

[BKKZ04] Christian Böhm, Karin Kailing, Peer Kröger, and Arthur Zimek. Computing clusters of correlation connected objects. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International*

Conference on Management of Data, Paris, France, June 13-18, 2004, pages 455–466. ACM, 2004.

[BKNS00] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. LOF: identifying density-based local outliers. In *SIGMOD Conference*, pages 93–104. ACM, 2000.

[Bro97] Andrei Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[BS03] S. D. Bay and M. Schwabacher. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. In *Proc. KDD*, pages 29–38, 2003.

[BVBB21] Arindam Bhattacharya, Sumanth Varambally, Amitabha Bagchi, and Srikanta Bedathur. Fast one-class classification using class boundary-preserving random projections. In *KDD*, 2021.

[Cha02] Moses S Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388. ACM, 2002.

[CK15] Flavio Chierichetti and Ravi Kumar. Lsh-preserving functions and their applications. *J. ACM*, 62(5):33:1–33:25, 2015.

[CKSZ20] Ricardo J. G. B. Campello, Peer Kröger, Jörg Sander, and Arthur Zimek. Density-based clustering. *WIREs Data Mining Knowl. Discov.*, 10(2), 2020.

[CMS13] Ricardo J. G. B. Campello, Davoud Moulavi, and Jörg Sander. Density-based clustering based on hierarchical density estimates. In *PAKDD (2)*, volume 7819 of *Lecture Notes in Computer Science*, pages 160–172. Springer, 2013.

[CMZS15] Ricardo J. G. B. Campello, Davoud Moulavi, Arthur Zimek, and Jörg Sander. Hierarchical density estimates for data clustering, visualization, and outlier detection. *ACM Trans. Knowl. Discov. Data*, 10(1):5:1–5:51, 2015.

[CW79] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

[CZS+16] Guilherme O Campos, Arthur Zimek, Jörg Sander, Ricardo JGB Campello, Barbora Micenková, Erich Schubert, Ira Assent, and Michael E Houle. On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study. *Data mining and knowledge discovery*, 30(4):891–927, 2016.

[dBGR19] Mark de Berg, Ade Gunawan, and Marcel Roeloffzen. Faster DBSCAN and HDBSCAN in low-dimensional euclidean spaces. *Int. J. Comput. Geom. Appl.*, 29(1):21–47, 2019.

[dCH12] T. de Vries, S. Chawla, and M. E. Houle. Density-preserving projections for large-scale local anomaly detection. *KAIS*, 32(1):25–52, 2012.

[DGD+24] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. 2024.

[DIIM04] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG*, pages 253–262. ACM, 2004.

[dSIN+21] Joelson Antônio dos Santos, Syed Talat Iqbal, Murilo Coelho Naldi, Ricardo J. G. B. Campello, and Joerg Sander. Hierarchical density-based clustering using mapreduce. *IEEE Trans. Big Data*, 7(1):102–114, 2021.

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231. AAAI Press, 1996.

[GBS05] Jan-Mark Geusebroek, Gertjan J. Burghouts, and Arnold W. M. Smeulders. The amsterdam library of object images. *Int. J. Comput. Vis.*, 61(1):103–112, 2005.

[GT15] Junhao Gan and Yufei Tao. Dbscan revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 519–530, New York, NY, USA, 2015. Association for Computing Machinery.

[GT17] Junhao Gan and Yufei Tao. On the hardness and approximation of euclidean DBSCAN. *ACM Trans. Database Syst.*, 42(3):14:1–14:45, 2017.

[GT18] Junhao Gan and Yufei Tao. Fast euclidean OPTICS with bounded precision in low dimensional space. In *SIGMOD Conference*, pages 1067–1082. ACM, 2018.

[HA85] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of Classification*, 2(1):193–218, 1985.

[HHH+22] Songqiao Han, Xiyang Hu, Hailiang Huang, Minqi Jiang, and Yue Zhao. Adbench: Anomaly detection benchmark. In *NeurIPS*, 2022.

[HKF04] Ville Hautamäki, Ismo Kärkkäinen, and Pasi Fränti. Outlier detection using k-nearest neighbour graph. In *ICPR (3)*, pages 430–433. IEEE Computer Society, 2004.

[HKK+10] Michael E. Houle, Hans-Peter Kriegel, Peer Kröger, Erich Schubert, and Arthur Zimek. Can shared-neighbor distances defeat the curse of dimensionality? In *SSDBM*, volume 6187 of *Lecture Notes in Computer Science*, pages 482–500. Springer, 2010.

[HKZ23] Michael E. Houle, Marie Kiermeier, and Arthur Zimek. Clustering high-dimensional data. In L. Rokach, O. Maimon, and E. Shmueli, editors, *Machine Learning for Data Science Handbook*. Springer, 2023.

[HYS23] Yihao Huang, Shangdi Yu, and Julian Shun. Faster parallel exact density peaks clustering. In *ACDA*, pages 49–62. SIAM, 2023.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC*, pages 604–613. ACM, 1998.

[IM18] M. Iwasaki and D. Miyazaki. Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data. *ArXiv e-prints*, October 2018.

[IX23] Piotr Indyk and Haike Xu. Worst-case performance of popular approximate nearest neighbor search implementations: Guarantees and limitations. In *NeurIPS*, 2023.

[JJL20] Heinrich Jiang, Jennifer Jang, and Jakub Lacki. Faster DBSCAN via subsampled similarity queries. In *NeurIPS*, 2020.

[JTH01] W. Jin, A. K. Tung, and J. Han. Mining top-n local outliers in large databases. In *Proc. KDD*, pages 293–298, 2001.

[KGKB03] G. Kollios, D. Gunopulos, N. Koudas, and S. Berchthold. Efficient biased sampling for approximate clustering and outlier detection in large datasets. *IEEE TKDE*, 15(5), 2003.

[KGPT22] Amir Keramatian, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas. Ip.lsh.dbscan: Integrated parallel density-based clustering through locality-sensitive hashing. In *Euro-Par 2022: Parallel Processing: 28th International Conference on Parallel and Distributed Computing, Glasgow, UK, August 22–26, 2022, Proceedings*, page 268–284, Berlin, Heidelberg, 2022. Springer-Verlag.

[KIW07] Hisashi Koga, Tetsuo Ishibashi, and Toshinori Watanabe. Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing. *Knowl. Inf. Syst.*, 12(1):25–53, 2007.

[KKK04] Karin Kailing, Hans-Peter Kriegel, and Peer Kröger. Density-connected subspace clustering for high-dimensional data. In Michael W. Berry, Umeshwar Dayal, Chandrika Kamath, and David B. Skillicorn, editors, *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, pages 246–256. SIAM, 2004.

[KKSZ11] Hans-Peter Kriegel, Peer Kröger, Erich Schubert, and Arthur Zimek. Interpreting and unifying outlier scores. In *SDM*, pages 13–24, 2011.

[KKZ09] Hans-Peter Kriegel, Peer Kröger, and Arthur Zimek. Clustering high-dimensional data: A survey on subspace clustering, pattern-based clustering, and correlation clustering. *ACM Trans. Knowl. Discov. Data*, 3(1):1:1–1:58, 2009.

[Kru56] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. American Mathematical Society*, 7(1):48–50, February 1956.

[KSZ11] Hans-Peter Kriegel, Erich Schubert, and Arthur Zimek. Evaluation of multiple clustering solutions. In *MultiClust@ECML/PKDD*, volume 772 of *CEUR Workshop Proceedings*, pages 55–66. CEUR-WS.org, 2011.

[KSZ17a] Evelyn Kirner, Erich Schubert, and Arthur Zimek. Good and bad neighborhood approximations for outlier detection ensembles. In *SISAP*, volume 10609 of *Lecture Notes in Computer Science*, pages 173–187. Springer, 2017.

[KSZ17b] Hans-Peter Kriegel, Erich Schubert, and Arthur Zimek. The (black) art of runtime evaluation: Are we comparing algorithms or implementations? *Knowl. Inf. Syst.*, 52(2):341–378, 2017.

[MDES18] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *The Network and Distributed System Security Symposium (NDSS) 2018*, 2018.

[MPLK14] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems*, 45:61–68, 2014.

[MSB+24] Magdalen Dobson Manohar, Zheqi Shen, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Harsha Vardhan Simhadri, and Yihan Sun. Parlayann: Scalable and deterministic parallel graph-based approximate nearest neighbor search algorithms. In *PPoPP*, pages 270–285. ACM, 2024.

[MY20] Yury A. Malkov and Dmitry A. Yashunin. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE TPAMI*, 42(4), 2020.

[NG09] H. V. Nguyen and V. Gopalkrishnan. Efficient pruning schemes for distance-based outlier detection. In *Proc. ECML PKDD*, pages 160–175, 2009.

[OATZ25] Camilla Birch Okkels, Martin Aumüller, Viktor Bello Thomsen, and Arthur Zimek. High-dimensional density-based clustering using locality-sensitive hashing. In *Proceedings*

28th International Conference on Extending Database Technology, EDBT 2025, Barcelona, Spain, March 25-28, 2025, pages 694–706. OpenProceedings.org, 2025.

[OAZ24] Camilla Birch Okkels, Martin Aumüller, and Arthur Zimek. On the design of scalable outlier detection methods using approximate nearest neighbor graphs. In *SISAP*, pages 170–184, 2024.

[OTA+26] Camilla Birch Okkels, Erik Thordsen, Martin Aumüller, Arthur Zimek, and Erich Schubert. Approximate single-linkage clustering using graph-based indexes: Mst-based approaches and incremental searchers. In Giuseppe Amato, Vladimir Mic, Agma Traina, Nicola Messina, Laurent Amsaleg, Gylfi Þór Guðmundsson, Björn Þór Jónsson, and Lucia Vadicamo, editors, *Similarity Search and Applications*, pages 233–247, Cham, 2026. Springer Nature Switzerland.

[OTW+10] G. H. Orair, C. Teixeira, Y. Wang, W. Meira Jr., and S. Parthasarathy. Distance-based outlier detection: Consolidation and renewed bearing. *PVLDB*, 3(2):1469–1480, 2010.

[Pug90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[RKH+21] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In *ICML*, volume 139 of *Proceedings of Machine Learning Research*, pages 8748–8763. PMLR, 2021.

[RNI15] Milos Radovanovic, Alexandros Nanopoulos, and Mirjana Ivanovic. Reverse nearest neighbors in unsupervised distance-based outlier detection. *IEEE Trans. Knowl. Data Eng.*, 27(5), 2015.

[RRS00] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. In *Proc. SIGMOD*, 2000.

[Sch24] Erich Schubert. Hierarchical clustering without pairwise distances by incremental similarity search. In *Proc. SISAP*, pages 238–252, 2024.

[SDS+19] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnaswamy, and Rohan Kadekodi. DiskANN: Fast Accurate Billion-point Nearest Neighbor Search on a Single Node. In *NeurIPS*, pages 13748–13758, 2019.

[Sib73] Robin Sibson. SLINK: an optimally efficient algorithm for the single-link cluster method. *Comput. J.*, 16(1):30–34, 1973.

[SM+58] Robert R Sokal, Charles D Michener, et al. A statistical method for evaluating systematic relationships. 1958.

[SQ19] Ye Shiqiu and Zhu Qingsheng. Dbscan clustering algorithm based on locality sensitive hashing. In *Journal of Physics: Conference Series*, volume 1314, page 012177. IOP Publishing, 2019.

[SR62] Robert R. Sokal and F. James Rohlf. The comparison of dendrograms by objective methods. *Taxon*, 11(2):33–40, 1962.

[SSE+17] Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. DBSCAN revisited, revisited: Why and how you should (still) use DBSCAN. *ACM Trans. Database Syst.*, 42(3):19:1–19:21, 2017.

[SV17] Johannes Schneider and Michail Vlachos. Scalable density-based clustering with quality guarantees using random projections. *Data Min. Knowl. Discov.*, 31(4):972–1005, 2017.

[SZK14a] Erich Schubert, Arthur Zimek, and Hans-Peter Kriegel. Generalized outlier detection with flexible kernel density estimates. In *SDM*, pages 542–550. SIAM, 2014.

[SZK14b] Erich Schubert, Arthur Zimek, and Hans-Peter Kriegel. Local outlier detection reconsidered: a generalized view on locality with applications to spatial, video, and network outlier detection. *Data mining and knowledge discovery*, 28:190–237, 2014.

[SZK15] E. Schubert, A. Zimek, and H.-P. Kriegel. Fast and scalable outlier detection with approximate nearest neighbor ensembles. In *Proc. DASFAA*, 2015.

[TS25] Erik Thordsen and Erich Schubert. Theoretical and practical insights into graph-based indexing. In *Proc. Int. Conf. on Similarity Search and Applications, SISAP*, 2025.

[VGO+19] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, Ilhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy. Scipy 1.0-fundamental algorithms for scientific computing in python. *CoRR*, abs/1907.10121, 2019.

[WGS20] Yiqiu Wang, Yan Gu, and Julian Shun. Theoretically-efficient and practical parallel DBSCAN. In *SIGMOD Conference*, pages 2555–2571. ACM, 2020.

[WGZ07] Yi-Pu Wu, Jin-Jiang Guo, and Xue-Jie Zhang. A linear dbscan algorithm based on lsh. In *2007 International Conference on Machine Learning and Cybernetics*, volume 5, pages 2608–2614. IEEE, 2007.

[WPT11] Y. Wang, S. Parthasarathy, and S. Tatikonda. Locality sensitive outlier detection: A ranking driven approach. In *Proc. ICDE*, 2011.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB*, pages 194–205, 1998.

[WYGS21] Yiqiu Wang, Shangdi Yu, Yan Gu, and Julian Shun. Fast parallel algorithms for euclidean minimum spanning tree and hierarchical spatial clustering. In *SIGMOD Conference*, pages 1982–1995. ACM, 2021.

[XP24] Haochuan Xu and Ninh Pham. Scalable density-based clustering with random projections. In *NeurIPS*, 2024.

[ZDH+17] X. Zhang, W. Dou, Q. He, R. Zhou, C. Leckie, R. Kotagiri, and Z. Salcic. LSHiForest: A generic framework for fast tree isolation based ensemble anomaly analysis. In *Proc. ICDE*, 2017.

[ZF18] Arthur Zimek and Peter Filzmoser. There and back again: Outlier detection between statistical reasoning and data mining algorithms. *Data Mining Knowl. Discov.*, 8(6), 2018.

[ZNL19] Yue Zhao, Zain Nasrullah, and Zheng Li. Pyod: A python toolbox for scalable outlier detection. *Journal of Machine Learning Research*, 20(96), 2019.

[ZSK12] Arthur Zimek, Erich Schubert, and Hans-Peter Kriegel. A survey on unsupervised outlier detection in high-dimensional numerical data. *Stat. Anal. Data Min.*, 5(5):363–387, 2012.