POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

A density-based method for scalable outlier detection in large datasets

Advisors

prof. Paolo Garza, Politecnico di Torino

prof. Abolfazl Asudeh, University of Illinois at Chicago

prof. Alessandro Campi, Politecnico di Milano

Candidate

Matteo Corain

matricola: 256654

Academic Year 2019-2020

Contents

List of Tables

List of Figures

List of Algorithms

Summary

DBSCAN is one of the most well-known algorithm in the field of density-based clustering, although its applicability to large datasets is generally disputed due to its high complexity. The aim of this work is to propose a new, parallel, Spark-based procedure for the sole purpose of anomaly detection, in a way which is coherent to the DBSCAN definition and suitable for the big data context. From a theoretical side, this algorithm is characterized by a worst-case performance boundary that depends linearly on the size of the dataset; in practical tests, it outperforms available solutions both in terms of result quality and overall scalability when the data grow large.

Chapter 1

Introduction

Outlier and noise detection are some of the most important tasks in a data mining pipeline. Specifically, outliers (correct observations which significantly differ from the others in terms of attribute values) may constitute a valuable information to collect *per se*, while noise points (produced due to a disturbed data generation process) need very often to be removed from a dataset before applying any kind of processing algorithm in order to improve the quality of the obtained results.

In a way, outlier detection is a complementary task with respect to clustering [23]: according to such definition, outliers may indeed be detected as the points such that they cannot be assigned to any of the identified clusters, or at least their assignment may be disputed. In this sense, this work builds upon the most recent developments in terms of parallelization of one of the most well-known clustering algorithms, DBSCAN, to define a novel procedure that detects outliers in a density-based fashion and runs in a worst-case linear time with respect to the size of the input dataset. In practical tests, such algorithm was able to outperform all the other available solutions, both in terms of the quality of the identified outliers and scalability in terms of running time.

This work is organized as follows. Chapter 2 (*Background and related work*) presents the theoretical notions and the main results found in literature regarding the two branches of density-based clustering and outlier detection algorithms. A brief overview of the main ideas behind the Apache Spark data processing library is also introduced. Chapter 3 (*The proposed algorithm*) describes the structure of the new algorithm, starting from some basic definitions and then continuing with the analysis of the three steps in which it can be conceptually subdivided and of some implementation tricks. Finally, Chapter 4 (*Experimental results*) presents the results of the execution of the algorithm on a range of different synthetic and real-world datasets, with the aim of characterizing the quality of such results and the overall scalability. Such results are put in comparison with those produced by some of the algorithms analyzed in Chapter 2, pointing out how the new algorithm outperforms those techniques in terms of both result quality and running time performance in most cases.

Chapter 2

Background and related work

In this chapter, an overview of the main results in the different branches of research the work targets is presented. In the first section, density-based clustering techniques are reviewed, focusing specifically on the DBSCAN algorithm and its parallel implementations. In the second section, some of the most common techniques for outlier detection are described. Finally, the last section introduces the main ideas of the Apache Spark data processing library, which has been used for the implementation of the algorithm.

2.1 Density-based clustering

2.1.1 Overview on clustering techniques

Clustering is a data mining task aiming to define groups of data, also called *clusters*, in a way that objects belonging to a cluster are more similar (in some sense) to one another than to objects outside the cluster [23]. Clustering represents a typical example of an *unsupervised* data mining task.

Due to the generality of the notion of cluster, many approaches and techniques have been proposed for the clustering task. Some of the most well-known techniques for clustering include the following [23]:

- *Prototype-based clustering*: in prototype-based clustering, each cluster is defined by a single representative (or prototype), usually the *centroid* or the *medoid* of the points in the cluster; data points are assigned to the cluster whose representative is closer (according to some distance measure) to the point itself. Examples of algorithms using this approach include the *K-means* algorithm and its variations (e.g. *K-medoids*, *bisecting K-means*); those are simple and efficient algorithms, but are bound to detect clusters with a specific shape (e.g. the globular shape, when the Euclidean distance measure is used) and do not perform well in presence of outliers.

- *Hierarchical clustering*: in hierarchical clustering, clusters are allowed to have nested subclusters; higher-level clusters are generated by joining the data points in all of their children. There are two basic approaches for generating a hierarchical clustering:

  - *Agglomerative*: agglomerative algorithms start from the lowest-level clusters (usually made up of a single point) and progressively merge them;
  - *Divisive*: divisive algorithms start from the highest-level clusters (usually a single cluster with all points) and progressively split them.

  The agglomerative approach is by far the most used; algorithms implementing such technique generally produce good-quality clusters, but they are computationally expensive and may not handle well noisy data.

- *Density-based clustering*: in density-based clustering, clusters are identified as regions characterized by a high density of points (according to some definition of density), separated from one another by regions with low density. One of the first and most well-known algorithm in this category is *DBSCAN*; such approach allows to define clusters with arbitrary shapes and is resistant to noise and outliers, but suffers when clusters have varying densities and does not perform well with high-dimensional data.

2.1.2 DBSCAN

DBSCAN [5] (Density-Based Spatial Clustering of Applications with Noise) is one of the first and most well-known algorithm for performing the clustering task in a density-based fashion. The traditional definition of the concept of density given by DBSCAN makes use of a *center-based approach*: the density is estimated, separately for each point in the dataset, by counting the number of samples whose distance from that point is less than a specified threshold, usually denoted as ϵ. This idea is formalized in the following definition.

Definition 2.1.1 (Neighborhood of a point)**.** Let ϵ be a positive real number and $\mathscr{D} \subseteq \mathbb{R}^d$ a reference dataset. The ϵ-neighborhood of a point $p \in \mathscr{D}$ is the set of points defined by:

$$\mathscr{N}_\epsilon(p) = \{q \in \mathscr{D} \ : \ dist(p,q) \leq \epsilon\} \tag{2.1}$$

Where $dist(\cdot,\cdot)$ is an arbitrary distance function in $\mathbb{R}^d$.

Note that an arbitrary point p is always included in its own ϵ-neighborhood, independently on the value of ϵ: indeed, $dist(p,p) = 0$, regardless of the chosen distance function.

In DBSCAN's terminology, the ϵ-neighborhood of a point is defined as "dense" whenever it contains a number of points that is greater than a specified threshold, usually denoted as *minPts*. With respect to the value of such parameter, each point in the dataset is classified according to the following definitions.

Definition 2.1.2 (Core point)**.** Let ϵ be a positive real number and *minPts* a positive integer number. An arbitrary point $p \in \mathscr{D}$ is a *core point* if and only if its ϵ-neighborhood contains at least *minPts* points:

$$|\mathscr{N}_\epsilon(p)| \geq minPts \tag{2.2}$$

Definition 2.1.3 (Border point)**.** An arbitrary point $p \in \mathscr{D}$ is a *border point* if it is not a core point, but it falls within the neighborhood of a core point.

Definition 2.1.4 (Outlier)**.** An arbitrary point $p \in \mathscr{D}$ is an *outlier* if it is neither a core point nor a border point.

In order to formalize the concept of cluster, the definitions of the property of density reachability and connection are needed.

Definition 2.1.5 (Density reachability)**.** Let $p, q \in \mathscr{D}$ be two arbitrary points in the dataset. We say that p is *directly density-reachable* from q with respect to parameters ϵ and *minPts* if the following conditions are verified:

- p is in the ϵ-neighborhood of q;
- q is a core point.

We say that p and q are *density-reachable* if there exists a chain of points $p_1, \dots, p_k$, where $p_1 = p$ and $p_k = q$, such that p_{i+1} is directly density-reachable from p_i for all $i = 1, \dots, k$.

Definition 2.1.6 (Density connection). Let $p, q \in \mathscr{D}$ be two arbitrary points in the dataset. We say that p is *density-connected* to q with respect to parameters ϵ and *minPts* if there exists a point $o \in \mathscr{D}$ such that both p and q are density-reachable from o.

Finally, the notion of cluster is given as follows.

Definition 2.1.7 (Cluster). A cluster C with respect to parameters ϵ and *minPts* is a non-empty subset of $\mathscr{D}$ satisfying the following conditions:

- For all $p, q \in \mathscr{D}$ such that $p \in C$ and q is density-reachable from p, then $q \in C$;
- For all $p, q \in C$, p is density-connected to q.

A principle implementation of DBSCAN is shown in Algorithm 2.1. The algorithm loops through all points in the dataset; for each unclassified point, it runs a range query in order to identify its neighbors. If the neighborhood contains less than *minPts* points, the vector is classified as noise; otherwise (i.e. p is core), it is assigned to a new cluster and a new loop is started to identify all the points that are density-reachable from the considered vector. For each of them, their label is set to the cluster identifier of the considered vector and a range query is performed; if they are core points as well, then all of their neighbors are assigned to the cluster (they are density-reachable from p, although not directly).

The time complexity of DBSCAN may simply be computed as:

$$O(n * \text{complexity of the range query computation}) \tag{2.3}$$

Naïve implementations of DBSCAN, using a linear scan of the dataset for performing the range query operation, have a $O(n^2)$ worst-case complexity; however, for low-dimensional data spaces, appropriate data structures may be used (such as R*-trees) to lower the average-case complexity of the algorithm.

Algorithm 2.1 The DBSCAN algorithm

```
function DBSCAN($data, eps, minPts$)
    $clusterId = 0$

    for $p$ in $data$ do
        ▷ Skip the point if already classified
        if GET-LABEL($p$) $\neq unclassified$ then
            continue
        end if

        ▷ Get the neighbors of the point
        $neighbors \leftarrow$ RANGE-QUERY($p, data, eps$)

        if SIZE($neighbors$) $< minPts$ then
            ▷ Point is noise
            SET-LABEL($p, noise$)
        else
            ▷ Point is core, new cluster identified
            $clusterId \leftarrow clusterId + 1$
            SET-LABEL($p, clusterId$)

            ▷ Expand cluster on nearby points
            $seedSet \leftarrow neighbors$
            while not EMPTY($seedSet$) do
                $q \leftarrow$ POP($seedSet$)

                ▷ Check that $q$ does not belong to other clusters
                if GET-LABEL($q$) $= unclassified \vee noise$ then
                    ▷ Assign neighbor to cluster
                    SET-LABEL($q, clusterId$)

                    ▷ Get neighbors
                    $neighbors \leftarrow$ RANGE-QUERY($q, data, eps$)

                    ▷ If $q$ is core, expand the cluster
                    if SIZE($neighbors$) $\geq minPts$ then
                        $seedSet \leftarrow$ APPEND($seedSet, neighbors$)
                    end if
                end if
            end while
        end if
    end for
end function
```

With respect to other clustering techniques, the advantages provided by the DBSCAN algorithm include:

- It does not require to specify the number of clusters *a priori*;
- It can find arbitrarily-shaped clusters;
- It allows to identify noise and outliers.

On the other hand, the main disadvantages of the DBSCAN algorithm can be summarized as follows:

- It does not behave well when clusters have varying densities;
- It performs poorly with high-dimensional data, since the concept of distance tends to lose significance when the number of dimensions grows;
- It has a high computational complexity, which makes it unsuitable in the big data context.

2.1.3 First approaches to grid-based DBSCAN

In order to cope with the high computational complexity of the base DBSCAN algorithm, some works have proposed the usage of a grid-based approach, which shows a greater scalability in practice with respect to the principle implementation of the algorithm.

One of the first attempts to make use of a grid structure to reduce the running time of DBSCAN is *GriDBSCAN* [16]. In their work, the authors propose the usage of a grid to evenly partition the input space into a user-specified number of cells (although they suggest that the cell width should be at least 2ϵ for performance reasons). Each partition is assigned all the points lying inside the cell, plus all the points within the *ϵ-enclosure* of such cell (i.e. the "border" of the cell having width ϵ). In this way, all the range queries may accurately be performed by considering only the points within each partition, although this leads to data duplication for points lying in the ϵ-enclosures. The algorithm computes clusters locally to each partition, then merges them according to the following rule: two clusters, identified within different partitions, need to be merged if there exists at least a core point which, separately in each partition, was assigned to one of the two clusters. Authors show that such algorithm is able to theoretically reduce the complexity of DBSCAN by a factor equal to the number of considered cells, when the merge cost is negligible; in their tests, GriDBSCAN outperforms a reference DBSCAN implementation by up to 9.4 times on real-world and synthetic datasets.

Another forerunner in the field of grid-based DBSCAN is *GF-DBSCAN* [24], which improves on the previous *FDBSCAN* [13]. Here, the authors propose the usage of a grid, in which each cell has side length of ϵ, to facilitate neighbor search: indeed, in order to retrieve the neighbors of a point within a cell, it is only necessary to look in the cell itself and in the adjacent ones. The algorithm scans linearly each data point; if the point does not belong to a cluster, a range query is performed: if the ϵ-neighborhood of the point contains at least *minPts* points, a new cluster is defined. Overlapping clusters are then merged into one to compute the final result. Such method is not able to achieve the same clustering as DBSCAN: Gunawan [8] effectively provides a counterexample for which both FDBSCAN and GF-DBSCAN would yield a different result with respect to the original procedure. In practical tests, however, this algorithm outperforms DBSCAN in terms of running time by up to three orders of magnitude while maintaining good levels of accuracy.

2.1.4 Gunawan's 2D algorithm

Gunawan [8] was the first to propose the usage of a grid with cells with diagonal length set to ϵ. In his work, the author proposes a two-dimensional algorithm that is shown to allow to run the exact DBSCAN in $O(n \log n)$ worst-case time, which inspired a number of successive studies on the parallel and approximated versions of DBSCAN.

The algorithm by Gunawan is subdivided in four consecutive phases:

- *Partitioning*: this step deals with the construction of a grid and with the assignment to each point to a cell in such grid; two partitioning methodologies are proposed, although only the fixed-size cells with diagonal ϵ approach is used in the following.
- *Core points identification*: this step deals with the identification of the core points within each cell; in particular, due to the size of the cell, the author shows how all points inside cells with cardinality at least *minPts* are core, while in the other case only a fixed number of cells (21, for the 2D case) needs to be checked for performing the range query.

- *Cluster merging*: this step deals with the definition of the final clusters, based on the fact that, whenever the distance between two core points in different cells is less than ϵ, the clusters originated by such core points need to be merged into one; in order to do so, the author proposes the construction of a graph-like structure, on which the connected components are computed.

- *Border and noise points identification*: finally, this step deals with the identification of border and noise points; the decision is based on the distance of each non-core point to a core point: if that is lower than ϵ, the point is on the border of a cluster, otherwise it is simply marked as noise.

Gunawan shows that Steps 1, 2 and 4 of the algorithm run in linear time with the size of the dataset; the linearithmic complexity is due to the cluster merging step, which has a $O(n \log n)$ worst-case boundary.

Subsequent works have further investigated the usage of such structure for accelerating DBSCAN performance. In particular, Gan and Tao [6] have shown that adapting Gunawan's algorithm for $d = 3$ achieves a $O\left((n \log n)^{4/3}\right)$ expected running time, while in higher dimensions the algorithm runs in $O\left(n^{2-\frac{2}{\lceil d/2 \rceil + 1}+\delta}\right)$ expected time, with $\delta > 0$ being an arbitrarily small constant. In addition to that, Gan and Tao [6] also have also proven that, for all dimensions $d \geq 3$, the DBSCAN problem can be solved only in at least $\Omega(n^{4/3})$ time, unless very significant advances in theoretical computer science are made. For this reason, exact DBSCAN algorithms are currently deemed to be intractable for processing very large amounts of data.

2.1.5 ρ-approximate DBSCAN

Due to the theoretical intractability of exact DBSCAN algorithms in the big data scenario, some work has recently been carried out with the aim of developing an approximate version of such algorithm, able to run within much stricter time constraints and characterized by solid theoretical guarantees.

The first and most famous of such approaches to approximated DBSCAN was proposed by Gan and Tao [6] and it is known as *ρ-approximate DBSCAN*. In ρ-approximate DBSCAN, an additional parameter, denoted as ρ, is introduced to control the degree of approximation of the algorithm. Based on the value of such parameter, the authors revisit the basic definitions of DBSCAN as follows.

Definition 2.1.8 (ρ-approximate density reachability)**.** Let $p, q \in \mathscr{D}$ be two arbitrary points in the dataset. We say that p is *ρ-approximate density reachable* from q with respect to parameters ϵ, *minPts* and ρ whenever there exists a chain of points, $p_1, \dots, p_k$, which satisfy the following statements:

- $p_1 = p$ and $p_k = q$;
- $p_1, \dots, p_{k-1}$ are core points;
- $p_{i+1} \in \mathscr{N}_{\epsilon(1+\rho)}(p_i)$, for all $i = 1, \dots, k-1$.

Definition 2.1.9 (ρ-approximate cluster)**.** A ρ-approximate cluster C with respect to parameters ϵ, *minPts* and ρ is a non-empty subset of $\mathscr{D}$ satisfying the following conditions:

- For all $p, q \in \mathscr{D}$ such that $p \in C$ and q is density-reachable from p, then $q \in C$;
- For all $p, q \in C$, p, there exists a point $o \in C$ such that both p and q are ρ-approximate density-reachable from o.

With respect to the original definition of cluster, ρ-approximate DBSCAN weakens the connectivity requirement to ρ-approximate connectivity. In spite of such approximation, the authors present as a quality guarantee the results of the following *sandwich theorem.*

Theorem 2.1.1. Let ϵ, *minPts* and ρ be the parameters of the ρ-approximate DBSCAN algorithm; then, it can be proven that:

- For any cluster C_1 produced by running the original DBSCAN algorithm with parameters ϵ and *minPts*, there exists a cluster C produced by running the ρ-approximate algorithm such that $C_1 \subseteq C$;
- For any cluster C_2 produced by running the original DBSCAN algorithm with parameters $\epsilon(1 + \rho)$ and *minPts*, there exists a cluster C produced by running the ρ-approximate algorithm such that $C \subseteq C_2$.

In addition to this quality guarantee, authors show that the algorithm runs in expected $O(n)$ time, coming from the usage of a quadtree-like data structure that can be built in $O(n)$ expected time and answers range queries in $O(1)$ time. Each cell of such quadtree is such to represent at least a point in the dataset and has a maximum side length of $\frac{\epsilon\rho}{\sqrt{d}}$.

2.1.6 Parallelization of DBSCAN

Another branch of research that has seen an important development in recent years consists in a number of works aiming to parallelize the execution of the DBSCAN algorithm to make it suitable to run in a distributed computing environment, such as the ones provided by Apache Hadoop or Apache Spark. Song and Lee, in their RP-DBSCAN paper [22], identify three categories of parallel DBSCAN algorithms:

- *Naïve random split algorithms*: those algorithms focus on decomposing the clustering problem into smaller problems, splitting the dataset in multiple disjoint and random subsets, on which the DBSCAN algorithm is computed before finally merging the obtained local clusters. Those are characterized by a high efficiency but low accuracy due to the impossibility of correctly estimating the density of the different neighborhoods.
- *Region split algorithms*: the idea behind those algorithms is to "smartly" partition the input dataset into contiguous and overlapping regions, such that the neighborhood of each point may be computed more precisely. They are generally more accurate than the previous, but suffer from load imbalance when used in conjunction with skewed datasets. To alleviate this problem, different strategies have been proposed for implementing such partitioning, including *even-split partitioning* (distribute the points as evenly as possible), *reduced-boundary partitioning* (minimize points in the overlapping regions) and *cost-based partitioning* (minimize the expected cost of the local clustering).
- *Graph-based algorithms*: they are based on the construction of an approximate k-nearest neighbor graph that is used for performing region queries.

A forerunner in the field of parallel DBSCAN is *PDBSCAN*, presented in [25]. Here, authors propose the usage of a *shared-nothing architecture* in which each node is assigned to a non-overlapping partition of the input space. A distributed spatial index, denoted as *dR*-tree*, is used to support data retrieval from other partitions (e.g. to compute the neighborhood of the points along the border of the partition). Local clusters to each partition are computed, then merged if there is at least a core point whose ϵ-neighborhood intersects with clusters identified in other partitions.

One of the first approaches to the parallelization of DBSCAN using the Hadoop MapReduce framework is *DBSCAN-MR* [4]. This is a region split algorithm, based on the concept of *partition with reduced boundary points*: the data points are split into equal-width slices along each dimension; slices with the lowest number of points are then selected as boundaries and associated to both partitions. After building an opportune spatial index, local clusters to each partition are computed, then merged through a relabeling phase which identifies core points shared by two clusters in two different partitions.

MR-DBSCAN [9] is another MapReduce-based exact DBSCAN algorithm, which implements instead a cost-based approach denoted as *cost-based spatial partitioning*. The input space is recursively subdivided into two smaller regions in a way that minimizes the expected cost of DBSCAN clustering on the produced partitions, computed through a cost function. An ϵ-wide margin is added to each partition in order to be able to decide whether a point is core or not by only looking to the points in the partition itself; local clusters are computed, then merged when at least a core point is shared by two clusters in two different partitions.

A first Spark-based implementation of a DBSCAN algorithm, called *RDD-DBSCAN*, is described in [3]. Here, a even-split strategy is adopted: the procedure for defining partitions is modified in a way to ensure that no one is smaller than 2ϵ or contain less than $maxPts$, which is an additional parameter of the algorithm; an ϵ-enclosure is added to each partition, then local clustering is computed and finally results are merged according to the same rule as before.

Finally, a different approach is followed in *NG-DBSCAN* [15], which uses a graph-based technique to compute approximate DBSCAN clusters based on Spark. Such algorithm, designed to work with any symmetric distance measure, exploits a data structure called *ϵ-graph*; each point is a node in the graph, whereas each edge indicates that two points are likely to be at a distance lower than ϵ. The graph is randomly initialized, then a certain number of iterations is performed to identify which of those random relationships actually hold; the final graph, clearly approximate, allows to detect the core points in the dataset without performing region queries. After marking the border and noise points, a vertex-centric procedure is used to define the clusters by spreading across the graph a “coreness” value for each core point, then proceeding to successive pruning to retrieve their seed and assign them to the correct cluster.

2.1.7 RP-DBSCAN

RP-DBSCAN [22] is one of state-of-the-art algorithms for density-based clustering, built upon the developments from both the approximate and the parallel DBSCAN research branches. The key contribution provided by RP-DBSCAN paper is the introduction of an algorithm based on random split (thus able to achieve high efficiency), but at the same time providing a theoretical guarantee of the accuracy of the results that is coherent with the one provided by Gan and Tao’s ρ-approximate DBSCAN.

The algorithm uses a grid-based approach and exploits a specific data structure, denoted as *two-level cell dictionary*, which allows to summarize the input dataset in terms of cell center and density. Specifically, the authors formalize this concept with the following definitions [22].

Definition 2.1.10 (Sub-cell)**.** A cell in a d-dimensional space is composed of $2^{d(h-1)}$ *sub-cells*, each of which being a d-dimensional hypercube with diagonal length set to $\frac{\epsilon}{2^{h-1}}$. The value of parameter h is computed from the approximation parameter ρ as:

$$h = 1 + \lceil \log_2 (\rho^{-1}) \rceil \tag{2.4}$$

Definition 2.1.11 (Two-level cell dictionary)**.** A *two-level cell dictionary* is a tree with a root node and multiple leaves, each of which consisting of multiple entries. Each root node entry represents a cell and each leaf node entry represents the belonging sub-cells. Every entry records the *center* and the *density* of the corresponding (sub)-cell.

In other words, the two-level cell dictionary summarizes the input dataset in terms of (sub)-cells, for each of which both the center and the density are known. Such cells are used for answering region queries based on the concept of proximity given by the following definition.

Definition 2.1.12 ((ϵ, ρ)-neighborhood)**.** Let C be a (sub)-cell with respect to parameters ϵ and ρ, having $\hat{q}$ as its center point. We say that C is a (ϵ, ρ)-neighbor of a point p if:

$$dist(p, \hat{q}) \leq \epsilon \tag{2.5}$$

Clearly, an (ϵ, ρ)-region query can be answered in $O(\log |cells|)$ when using an efficient tree structure. The core points in the dataset are detected by counting the number of points within their (ϵ, ρ)-neighborhoods; the definition of the clusters is then performed with the help of an additional data structure, called *cell graph*, based on the concept of cell-level reachability.

Definition 2.1.13 (Cell-level reachability)**.** Let C_1 and C_2 be two cells. We say that C_2 is *fully directly reachable* from C_1 if the following conditions apply:

- C_1 and C_2 contain at least a core point;
- There exists a core point $p \in C_1$ and a point $q \in C_2$ such that $dist(p, q) \leq \epsilon$.

We say that C_2 is *partially directly reachable* from C_1 if the following conditions apply:

- C_1 contains at least a core point;
- There exists a core point $p \in C_1$ and a point $q \in C_2$ such that $dist(p, q) \leq \epsilon$.

Definition 2.1.14 (Cell graph)**.** A *cell graph* is a graph in which:

- Vertices are the cells identified at the previous step, classified as either *core* and *non-core*;
- Edges describe the set of *fully* and *partially directly reachable* relationships between the different cells.

Cell subgraphs generated from single partitions are progressively merged in order to identify the final set of clusters. The RP-DBSCAN algorithm was implemented using Apache Spark, showing great performance improvements over all the previous parallel DBSCAN implementations (up to 180 times), reducing the problems related to load imbalance and data duplication and achieving great scalability also for very large datasets. Moreover, despite of the approximation introduced by the algorithm, the authors have shown that with low values of ρ (for example, $\rho = 0.01$) the results of the clustering are the same as DBSCAN's on benchmark datasets.

2.2 Anomaly detection

2.2.1 Overview on anomaly detection techniques

Anomaly detection is a data mining task that aims at the identification of objects (also called *outliers*) that differ significantly from other objects in the dataset [23]. Depending on the application context, the identification of outliers may either represent an interesting task *per se*, or a preprocessing step in a more complex data mining pipeline: in order to improve the quality of the produced models, outlier detection techniques are often used to identify (and remove) noise points before feeding the data to the algorithms of choice [23]. In most cases, anomaly detection is also carried out in an unsupervised fashion.

A very common approach, when performing anomaly detection tasks, is to define the "degree of anomaly" of a certain data point based on a numeric value, denoted as *outlier score*. Each point can then be classified as outlier or not depending on whether its outlier score is higher than a threshold value. Such threshold may either be a constant/empiric value or, when a rough estimation of the fraction f of outliers in the dataset is known, derived as the f-th percentile of the distribution of the outlier scores.

As in cluster analysis, due to the generality of the concept of outlier, different approaches have been proposed to perform anomaly detection tasks, which differ mainly on how the outlier score is measured. Most of the most well-known techniques for outlier detection fall within one of the following categories [23]:

- *Model-based techniques*: model-based techniques assume (or build) a model for the data, and identify as outliers those objects that do not fit well with such model.
- *Proximity-based techniques*: proximity-based techniques identify outliers as data points characterized by a high distance with respect to the remainder of the dataset, according to some measure.
- *Density-based techniques*: density-based techniques identify outliers as points belonging to regions characterized by a lower density with respect to the remainder of the dataset.
- *Clustering-based techniques*: clustering-based techniques identify outliers as data points which do not strongly belong to any of the identified clusters.

- *Classification-based techniques*: classification-based techniques make use of an opportunely trained classifier to distinguish outliers in the dataset.

2.2.2 Statistical methods

Statistical methods are a class of model-based techniques that use results from the field of statistics to infer a model of the data; objects in the dataset which appear to have a low probability with respect to the distribution defined by such model are marked as outliers [23]. A widely used assumption is to model the data generation as a random Gaussian process, either univariate or multivariate.

In the simple univariate case, the data are supposed to fit a normal distribution with mean $\mu \in \mathbb{R}$ and standard deviation $\sigma \in \mathbb{R}^+$. For each object in the dataset, it is possible to define its *z-score* as follows.

Definition 2.2.1 (z-score)**.** Let $x \in \mathcal{D}$ be a point in the dataset, for which we assume a data generation process that fits a normal distribution $\mathcal{N}(\mu, \sigma)$. The *z-score* of x is defined as:

$$z = \frac{x - \mu}{\sigma} \tag{2.6}$$

The more the point moves towards the tails of the Gaussian bell (and, consequently, the probability of generating such point decreases), the more its z-score increases; therefore, the z-score of a point may be used as a simple outlier score.

In the multivariate case, data generation is modeled as a normal process with mean $\mu \in \mathbb{R}^d$ and covariance matrix $\Sigma \in \mathbb{R}^{d,d}$. For such data distribution, a particular type of distance, known as *Mahalanobis distance*, is defined with respect to the parameters of the distribution as follows.

Definition 2.2.2 (Mahalanobis distance)**.** Let us consider a dataset $\mathcal{D} \subseteq \mathbb{R}^d$ generated from a random, multivariate Gaussian process $\mathcal{N}(\mu, \Sigma)$. For each point $x \in \mathcal{D}$, the *Mahalanobis distance* is defined as:

$$maha(x, \mu, \Sigma) = (x - \mu)\Sigma^{-1}(x - \mu)^T \tag{2.7}$$

It is possible to show that the Mahalanobis distance of a point is related to its probability according to the supposed distribution by the following relationship [23]:

$$maha(x, \mu, \Sigma) = -2 \log \mathbb{P}(x) + c(\Sigma) \tag{2.8}$$

Where $c(\Sigma)$ is a constant value once the covariance matrix is fixed. The Mahalanobis distance of a point is higher when the probability that such point has been generated by an underlying Gaussian process is lower; thus, such measure is a valid candidate for representing its outlier score. Graphically, deciding on whether a point is an outlier based on its Mahalanobis distance or not corresponds to the identification of a d-dimensional elliptic boundary around the mean of the process and elongated in the different directions to according the covariance values; all points lying outside such boundary are considered outliers.

Variants of the described method exists, which mainly differ in how the parameters of the process are estimated. For example, scikit-learn's `EllipticEnvelope` [18] uses a technique called Minimum Covariance Determinant (MCD) [19] in order to robustly estimate the covariance of the dataset without being highly affected by the presence of outliers. Those statistical methods perform well when the data generation process is known or can be modeled appropriately by means of a standard distribution, whereas they have poor performances in high-dimensional spaces.

2.2.3 Distance-based outliers

Proximity-based techniques decide on whether a data point has to be considered an outlier or not based on its distance with respect to its nearest neighbors: if the point appears to have a high k-th neighbor distance, then it is considered an outlier. Those methods, which are basically an extension of the well-known k-nearest neighbor algorithm, make use of the distance from the k-th nearest neighbor as the outlier score [23].

Another approach to outlier detection in a proximity-based fashion is described in [12]. This work formalizes the concept of outlier as in the following definition.

Definition 2.2.3 (Distance based-outlier). Let $x \in \mathscr{D}$ denote a point in the dataset and p, D two positive real values. We say that x is a *distance based-outlier* (or *DB-outlier*, in short) with respect to parameters p, D if there is at least a fraction p of objects in the dataset $\mathscr{D}$ which lie at a distance greater to D with respect to point x.

The authors propose, for the identification of DB-outliers, a number of different algorithms, including an index-based approach, a nested-loops approach and a more efficient cell-based approach. Here, points are partitioned in cells with diagonal length set to $\frac{D}{2\sqrt{d}}$, so that objects in the same cell have a maximum distance of $\frac{D}{2}$ and objects in adjacent cells have a maximum distance of D. This observation justifies the following theorem.

Theorem 2.2.1. Let C denote a cell in the previously defined grid; then, the following propositions hold:

- If C contains at least $M = N(1 - p)$ points, then no object in C is an outlier;
- If the total number of points in C and its first-level neighbors is at least M, then no object in C is an outlier;
- If the total number of points in C and its first and second-level neighbors is at most M, then every object in C is an outlier.

Distance-based methods are generally simple to implement and understand, but typically expensive, largely dependent on the selection of parameters and unable to handle data with varying densities.

2.2.4 Local outlier factor

Density-based techniques classify as outliers points in regions characterized by a lower density with respect to other regions in the dataset. One of the most popular approaches to define the outlier score for the points in the dataset in a density-based fashion is known as *Local Outlier Factor* [2]. The definition of such parameter is given through the following definitions.

Definition 2.2.4 (Reachability distance)**.** Let $p, q \in \mathscr{D}$ be two points in the dataset and k a positive integer number. The *reachability distance* of p with respect to q is defined as:

$$\textit{r-dist}_k(p, q) = \max\{\textit{k-dist}(q), \textit{dist}(p, q)\} \tag{2.9}$$

Where $\textit{k-dist}(q)$ denotes the distance of the k-th nearest neighbor from object q.

Definition 2.2.5 (Local reachability density)**.** Let $p \in \mathscr{D}$ be a point in the dataset and k a positive integer. The *local reachability density* of point p is defined as:

$$lrd_k(p) = \left(\frac{\sum_{o \in \mathscr{N}_k(p)} r\text{-}dist_k(p, o)}{|\mathscr{N}_k(p)|} \right)^{-1} \tag{2.10}$$

Where $\mathscr{N}_k(p)$ denotes the set of the k nearest neighbors to point p.

Definition 2.2.6 (Local outlier factor)**.** Let $p \in \mathscr{D}$ be a point in the dataset and k a positive integer. The *local outlier factor* of point p is defined as:

$$LOF_k(p) = \frac{\sum_{o \in \mathscr{N}_k(p)} \frac{lrd_k(o)}{lrd_k(p)}}{|\mathscr{N}_k(p)|} \tag{2.11}$$

The only parameter that is needed to compute the Local Outlier Factor is the value of k, which has to be user-specified. The LOF of a point averages the ratio between the local reachability density of the neighbors of the point and that of the point itself; the lower the density around point p with respect to the density around its neighbors is, the higher the value of the LOF is, whereas its value tends to 1 for objects in the core of a dense region.

The `LocalOutlierFactor` class provided by scikit-learn [18] represents a straightforward implementation of the described methodology. If the proportion of outliers is not specified, the algorithm uses a threshold of 1.5 as in the original LOF paper. The LOF approach generally performs well with low-dimensional data even if they present regions with varying densities, but it is expensive in higher dimensional spaces and it may need a thorough parameter selection phase (as shown by the authors, the relationship between the choice of k and the value of the LOF is not monotonic).

2.2.5 Clustering-based approaches

Clustering-based techniques build upon the idea that outliers are those data objects which cannot be clearly assigned to any of the clusters, or that form by themselves small and sparse clusters. Some algorithms, such as DBSCAN, already mark some points as outliers during the clustering process; others, such as K-means, always assign each point to a cluster, and therefore it is necessary to evaluate how “strong” such assignment is.

The most basic way to decide whether an object belongs to the cluster it was assigned to or should be considered an outlier is to measure the distance of the point with respect to the cluster representative; if its value is much greater than those of the other points in the cluster, then the vector is likely to be an outlier (i.e. it can be used as an outlier score). Clustering-based techniques are inexpensive to apply whenever the set of clusters has already been computed (i.e. as part of a data mining pipeline), but the results depend heavily on the chosen algorithm and on the selection of parameters.

2.2.6 One-class classifiers

Outlier detection can be treated as a classification problem whenever an opportune training set of labeled normal/abnormal samples is available. However, this is not the case in most scenarios: such data may be difficult to obtain and, even when it is possible, problems with high class imbalances may arise. For this reason, many classification-based techniques make use of a different type of classifier, denoted as *one-class classifier*, specifically designed for the purpose of anomaly detection.

One-class classification represents a typical semi-supervised problem: given a training set of "normal" samples, the objective is to fit a model that is able to opportunely represent the "normal" class, without any specific information on the "outlier" class. Every object that does not comply with this model is considered to be an outlier. One example of such classifier is the *one-class support vector machine* introduced in [20].

Definition 2.2.7 (One-class support vector machine)**.** Let $x_1, \ldots, x_l \in \mathcal{X} \subseteq \mathbb{R}^d$ be a set of "normal" observations and $\Phi : \mathcal{X} \to \mathcal{F}$ a feature map for the vectors in $\mathcal{X}$. A *one-class support vector machine* is a classifier implementing the decision function:

$$f(x) = \operatorname{sgn}\left(w \cdot \Phi(x) - \rho\right) \tag{2.12}$$

Where $w \in \mathcal{F}, \rho \in \mathbb{R}$ are the solutions to the following quadratic optimization problem:

$$\min_{w \in \mathcal{F}, \xi \in \mathbb{R}^l, \rho \in \mathbb{R}} \frac{1}{2}||w||^2 + \frac{1}{\nu l} \sum_i \xi_i - \rho \tag{2.13}$$

Subject to the linear constraints:

$$w \cdot \Phi(x_i) \geq \rho - \xi_i \tag{2.14}$$

$$\xi_i \geq 0 \tag{2.15}$$

The feature map (or the corresponding kernel function) and ν are the parameters of the training algorithm. In particular, the value of ν is linked to the proportion of outliers by the following theorem.

Theorem 2.2.2. Assume that the solution of a one-class SVM problem satisfies $\rho \neq 0$. Then, the following statements hold:

- ν is an upper bound on the fraction of outliers;
- ν is a lower bound on the fraction of support vectors;
- ν equals both the fraction of support vectors and outliers asymptotically with probability 1, if the data are generated independently and the kernel is analytic and non-constant.

A straightforward implementation of the one-class SVM method is provided by scikit-learn's `OneClassSVM` [18]. Such classifier uses by default a RBF kernel and $\nu = 0.5$.

2.2.7 Isolation forests

Isolation trees and isolation forests [14] are a variant of decision trees and random forests specifically designed for the outlier detection task. The idea behind such technique comes from the observation that, in trees trained by using random partitioning at each split, outliers are on average easier to isolate (i.e. they require less splits) due to their reduced number and to the unusual values of their attributes. This justifies the introduction of a decision model based on the following concepts.

Definition 2.2.8 (Isolation tree)**.** An isolation tree is a binary tree-like structure, in which each node T is either an external node with no child, or an internal node with exactly two children and a test. Each test consists of an attribute and a randomly selected threshold value that divides the data points between the two child nodes.

Definition 2.2.9 (Isolation forest)**.** An *isolation forest* is an ensemble model made up of a collection of independently trained isolation trees.

The training procedure for an isolation tree terminates whenever the maximum tree height (set to $h = \lceil \log_2 n \rceil$ in the original paper) is reached, the node contains a single object or all objects in the node are overlapping. With respect to a trained isolation forest, the anomaly score for each point x in the dataset is defined by:

$$s(x, n) = 2^{-\frac{\mathbb{E}(h(x))}{c(n)}} \tag{2.16}$$

Where $\mathbb{E}(h(x))$ denotes the average path length for object x in each of the random trees and $c(n) = 2H(n-1) - 2\frac{n-1}{n}$ is the average height of a binary tree with n objects. The value of a so-defined anomaly score is bounded between 0 and 1; it is close to 0 for “normal” points (whose path length is higher on average) and to 1 for outliers (whose path length is generally shorter).

A straightforward implementation of the described method is provided by scikit-learn’s `IsolationForest` [18], which defaults to 100 base estimators and a decision value of 0.5.

2.3 Apache Spark

Spark [1] is an open-source, general-purpose, distributed data processing library, developed and maintained by the Apache Foundation. The Spark computing framework provides a programming abstraction, based on the concept of RDDs, and transparent mechanisms for distributing and parallelizing the execution of the specified task on the nodes of a computing cluster, hiding the complexity of fault tolerance mechanisms, of job scheduling and of job synchronization.

The core component of Spark is the *Resilient Distributed Dataset*, or *RDD*: this represents a collection of objects, partitioned across the different worker nodes in the cluster, stored and processed in main memory whenever possible. In order to reduce the synchronization cost among the nodes and allow for greater fault tolerance, RDDs are immutable: whenever the content of an RDD needs to be changed, a new one is instantiated.

RDDs are split in chunks that are assigned to the different nodes in the cluster; for the sake of efficiency, operations are performed first locally on the data present in each single node, then the results are recombined after distributing them to the different nodes (operation known as *shuffle*). In the Spark terminology, a node executing part of the computation is called *worker node*; the same worker node may be used to perform different computations (tasks) in parallel, each of which is assigned to a local *executor*. The computation carried out by worker nodes is coordinated by a *driver program* (either running on a node of the cluster or on an external machine), which distributes the jobs and collects the results produced by each executor.

RDDs support two types of operations:

- *Transformations* turn an RDD to another RDD; due to the immutability of RDDs, each transformation produces a new RDD, whose contents are obtained by applying the specified operation to the elements of the input RDD;
- *Actions* return a value to the driver program, in the form of a local object in the programming language of choice.

A typical chain of operations performed on an RDD is based on a set of transformations applied in sequence on the RDDs produced by the previous one, which is concluded by a single action that is used to extract the final result. The presence of the action to conclude the chain of transformations is essential; transformations, in Spark, are indeed *lazily computed*: if no action is present, the Spark execution core will not perform any operation. When a transformation is invoked, Spark keeps track of the dependency between the input and the output RDD, but defers the computation of the contents of the new RDD to the moment in which an action is specified. This allows for a greater efficiency (the list of transformations may be optimized without changing the result before executing them) and reliability (it is always known how to recompute the contents of an RDD). Tables 2.1 and 2.2 present the main transformations and actions that are available in Spark.

A variant of the standard RDD, called *pair RDD*, also plays a central role in Spark. A pair RDD is a particular type of RDD, representing a collection of key-value pairs. In addition to all the RDDs' standard transformations and actions, pair RDDs also support specific operations, generally related to data grouping (computations are executed within groups of objects characterized by the same key). Tables 2.3 and 2.4 present some of the pair RDD-specific transformations and actions that are available in Spark.

Table 2.1. Main RDD transformations in Apache Spark

Transformation	Description
FILTER	It returns a new RDD containing only the elements of the input RDD for which the provided function evaluates to true.
DISTINCT	It returns a new RDD containing only the distinct elements of the input RDD.
SAMPLE	It returns a new RDD containing a random sample of the elements of the input RDD, whose size is the given fraction of the original one, with or without replacement.
MAP	It creates a new RDD containing exactly one element for each element x of the input RDD, obtained by applying the given function on x.
FLATMAP	It creates a new RDD containing zero or more elements for each element x of the input RDD, obtained by applying the given function on x; returned sequences are concatenated in the new RDD, without removing duplicates.
MAPTOPAIR	It creates a new pair RDD by applying the given function on each element of the original RDD; the created pair RDD will contain a tuple for each element of the original RDD.
FLATMAPTOPAIR	It creates a new pair RDD by applying the given function on each element of the original RDD; the created pair RDD will contain zero or more tuples for each element of the original RDD.
ZIPWITHUNIQUEID	It creates a new pair RDD by associating each element of the original RDD with a unique identifier.
UNION	It returns a new RDD consisting of the union of the elements belonging to two RDDs; the types of the two RDDs must be the same and duplicates are not removed.
INTERSECT	It returns a new RDD consisting of the intersection of the elements belonging to two RDDs; the types of the two RDDs must be the same.
SUBTRACT	It returns a new RDD consisting of the subtraction of the elements belonging to two RDDs; the types of the two RDDs must be the same.

Table 2.2. Main RDD actions in Apache Spark

Action	Description
Count	It returns the number of elements in the given RDD.
First	It returns the first element of the RDD.
Take	It returns a local collection containing the first *n* elements of the RDD
TakeSample	It returns a local collection containing *n* random elements of the RDD, with or without replacement.
TakeOrdered	It returns a local collection containing the first *n* sorted elements of the RDD.
Collect	It returns a local collection containing all the elements of the RDD.
Reduce	It returns a single object obtained by combining the contents of the RDD by means of the provided function, which needs to be associative and commutative. Elements are processed in pairs: they are removed from the RDD and their result is placed back, until a single one remains. The returned data type must be the same of the elements of the input RDD.
Aggregate	It returns a single object obtained by combining the contents of the RDD by means of the provided functions and an initial zero value; the functions must be associative and commutative and the returned object can be of a different type with respect to the original objects.
SaveAsTextFile	It stores the RDD in a set of files of the given output path.

Table 2.3. Main pair RDD transformations in Apache Spark

Transformation	Description
MapValues	It creates a new pair RDD where there is one pair for each pair in the input RDD, with the same key; values are obtained by applying the provided function to the values of the pairs in the original RDD.
FlatMapValues	It creates a new pair RDD where there are zero or more pairs for each pair in the input RDD, with the same key; values are obtained by applying the provided function to the values of the pairs in the original RDD.
ReduceByKey	It creates a new pair RDD where there is one pair for each distinct key *k* of the input pair RDD; the value associated with *k* in the new pair RDD is computed by combining via the provided function all the values associated with *k* in the input pair RDD. The function must be associative and commutative; the data type of the new pair RDD is the same of the input pair RDD.
AggregateByKey	It creates a new pair RDD where there is one pair for each distinct key *k* of the input pair RDD; the value associated with key *k* in the new pair RDD is computed by combining via the provided functions the values associated with *k* in the input pair RDD. The functions must be associative and commutative; the data type of the new pair RDD can be different than the one of the input RDD.
GroupByKey	It creates a new pair RDD where there is one pair for each distinct key of the input pair RDD. The value of such pair in the new pair RDD is the list of values associated with the same key in the input pair RDD.
SubtractByKey	It creates a new pair RDD containing only the pairs of the input pair RDD associated with a key that is not appearing as key in the pairs of another pair RDD. The data type of the new pair RDD is the same of the input pair RDD; the two RDDs must have the same data type for the keys.
Join	It creates a new pair RDD by joining the pairs in two pair RDDs based on their keys; each pair of the input pair RDD is combined with all the pairs of the other pair RDD with the same key. The new pair RDD has the same key data type of the input pair RDDs and a tuple as value (the pair of values of the two joined input pairs); the two RDDs must have the same data type for the keys.

Table 2.4. Main pair RDD actions in Apache Spark

Action	Description
CountByKey	It returns a local map containing, for each key, the number of elements associated with it in the input pair RDD.
CollectAsMap	It returns a local map containing all the pairs of the pair RDD. If the input pair RDD contains more than one pair with the same key, only one of those pairs is returned (usually, the last one).

Chapter 3

The proposed algorithm

In this chapter, the structure of the proposed algorithm is described in detail. In the first section, some basic definitions and theoretical results are presented and proven; the following section gives an overview of the three phases of the algorithm, which are then analyzed in more depth in the subsequent paragraphs. Finally, the last section presents some practical optimization techniques that have been applied to the base algorithm in order to improve its performance in the average case.

3.1 Definitions

3.1.1 Cell

Let us begin by introducing the concept of *cell*.

Definition 3.1.1 (Cell)**.** For any dimensionality d, an ϵ-*cell* (or simply a cell) is a d-dimensional hypercube having its diagonal length set to the value of the parameter ϵ, with the same meaning as in the DBSCAN algorithm. Each cell $C \subseteq \mathbb{R}^d$ is uniquely identified by a d-dimensional tuple of integer values, $c = (c_1, \dots, c_d) \in \mathbb{Z}^d$, which represents the coordinates of the vertex of the hypercube with the minimum values, scaled by the quantity $l = \frac{\epsilon}{\sqrt{d}}$.

Note that the value of $l = \frac{\epsilon}{\sqrt{d}}$ corresponds to the length of the side of the cell along any dimension. A generic point $p = (p_1, \dots, p_d) \in \mathbb{R}^d$ belongs to cell C if and only if the following set of inequalities applies:

$$c_i * \frac{\epsilon}{\sqrt{d}} \leq p_i < (c_i + 1) * \frac{\epsilon}{\sqrt{d}}, \forall i = 1, \ldots, d \tag{3.1}$$

Consequently, it is possible to compute the cell coordinates for any arbitrary point p by applying the following formula:

$$c_i = \left\lfloor p_i * \frac{\sqrt{d}}{\epsilon} \right\rfloor, \forall i = 1, \ldots, d \tag{3.2}$$

Where $\lfloor\cdot\rfloor$ denotes the integer flooring operator.

3.1.2 Cell types

Any given cell may be classified with respect to the number and the type of its points according to the following definitions.

Definition 3.1.2 (Dense cell)**.** For any given cell $C \subseteq \mathbb{R}^d$, we say that C is *dense* if it contains at least *minPts* points, where the *minPts* parameter has the same meaning as in the DBSCAN algorithm.

Definition 3.1.3 (Core cell)**.** For any given cell $C \subseteq \mathbb{R}^d$, we say that C is *core* if it contains at least a core point, in the sense of the DBSCAN algorithm.

It is immediate to prove the following theorems.

Theorem 3.1.1. Let $C \subseteq \mathbb{R}^d$ be a dense cell; then, all points contained in C are core points in the sense of the DBSCAN algorithm.

Proof. From the definition of ϵ-cell, we have that the distance between any given points $p_1, p_2 \in C$ is at most ϵ. Since C is dense, any point $p \in C$ will have at least *minPts* neighboring points with a distance that is at most ϵ (all the points within the cell); hence, by the definition of core point, all points in C are core in the sense of the DBSCAN algorithm. □

Theorem 3.1.2. Let $C \subseteq \mathbb{R}^d$ be a core cell; then, no point contained in C can be an outlier in the sense of the DBSCAN algorithm.

Proof. From the definition of ϵ-cell, we have that the distance between any given points $p_1, p_2 \in C$ is at most ϵ. Since C is core, there exists at least a point $p^* \in C$ that is core; all the other points in the cell will be neighbors of that point. Hence, according to the definition of outlier, they can be only border or core as well. □

3.1.3 Neighboring cells

The following definition deals with the concept of proximity between cells.

Definition 3.1.4 (Neighboring cells). Given any two non-empty cells $C_1, C_2 \subseteq \mathbb{R}^d$, we say that C_1 and C_2 are *neighbors* if there may exist two points, $p_1 \in C_1$ and $p_2 \in C_2$, such that:

$$dist(p_1, p_2) < \epsilon \tag{3.3}$$

Where $dist(\cdot, \cdot)$ is an arbitrary distance function in $\mathbb{R}^d$.

In other words, two non-empty cells are considered neighboring whenever the minimum possible distance between any couple of points in the two cells is less than the value of ϵ. We denote as $\mathcal{N}(c)$ the set of neighbors of a given cell C. Note that, by definition, each cell is a neighbor of itself (the minimum distance between the points in the cell is zero). Figures 3.1 and 3.2 depict the shape of a neighborhood respectively for a two-dimensional and a three-dimensional cell.

An important property deriving from the above definition is that the maximum number of neighbors for any given cell is constant once we fix the dimensionality d of the problem, regardless of the parameter ϵ; the value of such constant will be denoted as k_d in the following. A very loose bound on k_d is given by the following theorem.

Theorem 3.1.3. For any fixed number of dimensions d, an upper bound to the value of k_d is given by:

$$k_d = O\left(2\left\lceil\sqrt{d}\right\rceil + 1\right)^d \tag{3.4}$$

Where $\lceil\cdot\rceil$ denotes the integer ceiling operator.

Proof. Let us consider a cell $C \subseteq \mathbb{R}^d$, identified by $c = (c_1, \ldots, c_d) \in \mathbb{R}^d$. A necessary (although not sufficient) condition for a cell $C' \subseteq \mathbb{R}^d$ to be neighbor of C is that its coordinates $c'_i = c_i + j$ (where $j \in \mathbb{Z}$) satisfy, along any fixed dimension i, the inequality:

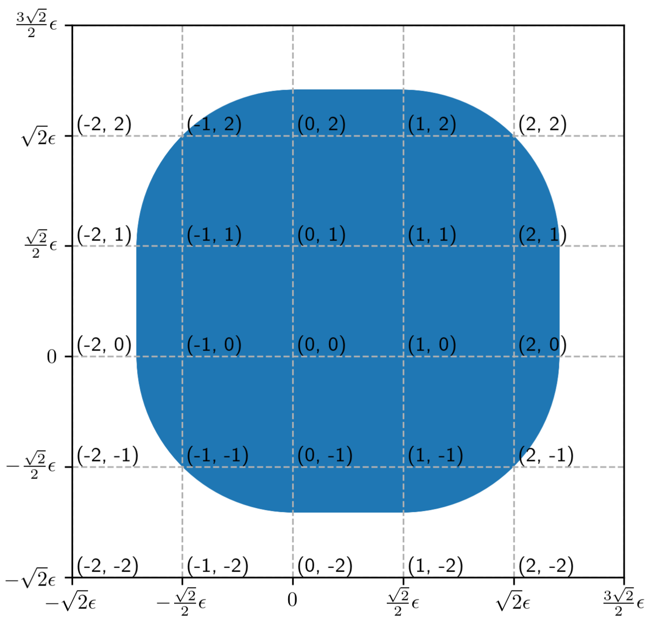

Figure 3.1. Two-dimensional neighborhood of cell (0,0)

$$lc_i - \epsilon \leq lc_i' \leq lc_i + \epsilon \Rightarrow \epsilon\left(\frac{c_i}{\sqrt{d}} - 1\right) \leq \frac{\epsilon}{\sqrt{d}}(c_i + j) \leq \epsilon\left(\frac{c_i}{\sqrt{d}} + 1\right) \tag{3.5}$$

Simplifying the above expression, we obtain:

$$-\sqrt{d} \leq j \leq \sqrt{d} \tag{3.6}$$

Possible values for j are therefore those between $-\lceil\sqrt{d}\rceil$ and $\lceil\sqrt{d}\rceil$, which identify $2\lceil\sqrt{d}\rceil+1$ different cells. In other words, all the possible neighbors of C are contained within a hypercube made up of $2\lceil\sqrt{d}\rceil + 1$ cells along each of the d directions; hence, their total number is:

$$\left(2\lceil\sqrt{d}\rceil + 1\right)^d \tag{3.7}$$

$\square$

The number of cells that are actually at a minimum distance lower than ϵ is generally lower than this, especially for higher dimensions. Table 3.1 shows the value of k_d for $d = 2, \dots, 9$.

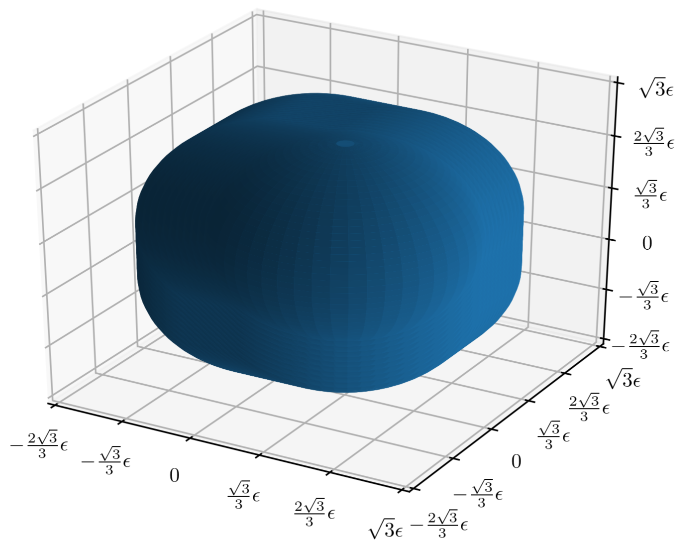
$\sqrt{3}\epsilon$
$\frac{2\sqrt{3}}{3}\epsilon$
$\frac{\sqrt{3}}{3}\epsilon$
0
$-\frac{\sqrt{3}}{3}\epsilon$
$-\frac{2\sqrt{3}}{3}\epsilon$

Figure 3.2. Three-dimensional neighborhood of cell (0,0,0)

Table 3.1. Some notable values for k_d

d	Upper bound	Actual k_d
2	25	21
3	125	117
4	625	609
5	16807	3903
6	117649	28197
7	823543	197067
8	5764801	1278129
9	40353607	8077671

The procedure used for the proof above justifies the introduction of Algorithm 3.1 for the generation of the coordinates of a given cell's neighbors. This is performed by means of a recursive function, Generate-Neighbors-Rec, which computes all the combinations of values between $c_i - \delta$ and $c_i + \delta$ (where $\delta = \lceil\sqrt{d}\rceil$) along each of the axes $i = 1, \dots, d$; the coordinates of all cells in the hypercube with side $2\delta+1$ centered in the considered cell are therefore generated. Whenever a new combination is computed, the Min-Distance function is applied in order to decide whether the two cells are effectively neighbors or not (i.e. the distance between two points placed on two opposite cell boundaries is at most ϵ).

In order to show how to measure the minimum cell distance, let us denote two arbitrary cells as $C_1, C_2 \subseteq \mathbb{R}^d$, with coordinates $c_1, c_2 \in \mathbb{Z}^d$, and let us suppose that, along some direction i, $c_{2i} > c_{1i}$; then, the points $p_1 \in C_1$ and $p_2 \in C_2$ with minimum distance along the i-th axis will be characterized by:

$$p_{1i} = (c_{1i} + 1 - \lambda) * l \tag{3.8}$$

$$p_{2i} = c_{2i} * l \tag{3.9}$$

Where λ is an arbitrarily small positive value. The minimum distance between the two cells along direction i is therefore:

$$p_{2i} - p_{1i} = l\,[c_{2i} - (c_{1i} + 1 - \lambda)] \xrightarrow{\lambda \to 0} l\,(c_{2i} - c_{1i} - 1) = \frac{\epsilon}{\sqrt{d}}\,(c_{2i} - c_{1i} - 1) \tag{3.10}$$

Generalizing the above results and accounting for the trivial case in which $c_{1i} = c_{2i}$, the formula for axis distance can be rewritten as:

$$dist_i(p_1, p_2) = \begin{cases} l\,(|c_{2i} - c_{1i}| - 1) & \text{if } c_{1i} \neq c_{2i} \\ 0 & \text{if } c_{1i} = c_{2i} \end{cases} \tag{3.11}$$

To compute the actual distance, all the axis distances are finally squared and summed. If this is found to be less than ϵ, then the generated cell C_2 is a neighbor of the considered cell C_1 (and vice versa); otherwise, it can be discarded.

The implemented procedure is slightly different from the one described above for simplicity and to avoid problems with additional floating-point roundings, but it follows the same logic. Indeed, this computes the integer axis distances, without multiplying by l; those are squared and summed, then the result is scaled by $\frac{1}{\sqrt{d}}$ to take into account the side length for the considered dimension. To decide whether the two cell are neighbors or not, this value is simply compared to 1.

Algorithm 3.1 Neighbors generation procedure

```
function MIN-DISTANCE($cell_1, cell_2, dim$)
    $dist \leftarrow 0$

    for $i$ in $[1, dim]$ do
        ▷ Compute distance along an axis
        $axisDist = |cell_1[i] - cell_2[i]| - 1$

        ▷ Update total distance
        if $axisDist > 0$ then
            $dist \leftarrow dist + axisDist^2$
        end if
    end for

    return $\frac{dist}{\sqrt{dim}}$
end function

function GENERATE-NEIGHBORS($cell, dim$)
    $delta \leftarrow$ CEIL($\sqrt{dim}$)
    $newCell \leftarrow$ EMPTY-LIST
    $neighbors \leftarrow$ EMPTY-LIST

    GENERATE-NEIGHBORS-REC($0, delta, dim, cell, newCell, neighbors$)
    return $neighbors$
end function

function GENERATE-NEIGHBORS-REC($x, delta, dim, cell, newCell, neighbors$)
    ▷ Check if an entire cell position has been generated
    if $x = dim$ then
        ▷ Add the cell to the neighbors if its minimum distance is at most 1
        if MIN-DISTANCE($cell, newCell, dim$) $< 1$ then
            $neighbors \leftarrow$ APPEND($neighbors, newCell$)
        end if
        return
    end if

    ▷ Otherwise, generate a component and go to the next
    for $i$ in $[cell[x] - delta, cell[x] + delta]$ do
        $newCell[x] \leftarrow i$
        GENERATE-NEIGHBORS-REC($x + 1, delta, dim, cell, newCell, neighbors$)
    end for
end function
```

3.1.4 Grid

Finally, let us introduce the concept of grid.

Definition 3.1.5 (Grid)**.** For any input space $\mathbb{R}^d$, a *grid* is a complete and non-overlapping partition $\mathscr{G} = C_1, \dots, C_k$ of that input space into ϵ-cells.

Clearly, if n is the size of the input dataset, the number of cells in the grid is $O(n)$. Indeed, in the worst case, each point belongs to a different cell.

3.2 Overview of the algorithm

Built upon the recent developments of DBSCAN-like clustering techniques, the algorithm we propose is aimed at fulfilling two main objectives: achieving high scalability in big data contexts (with a theoretical linear worst-time complexity guarantee) and providing an exact result (i.e. the same result that would be obtained by running the original DBSCAN algorithm on the same dataset). Very high-level, this is structured as a grid-based procedure that can conceptually be subdivided into three steps:

- *Cell construction and grid definition*: in this step, the input dataset is parsed and each of its points is assigned to the corresponding ϵ-cell;
- *Core points identification*: in this step, the grid constructed at the previous step is exploited to discover all the core points in the dataset;
- *Outliers identification*: in this step, all the non-core points are checked to finally identify the outliers in the dataset.

The first two phases are, in their baseline ideas, largely based on Gunawan's [8], while improved and generalized for working with $d \geq 2$. Note that this workflow does not involve the definition of DBSCAN-like clusters: outliers in the dataset may indeed directly be obtained when the core points are known. In the following, each step is described more in detail, with the help of a graphic example. For this purpose, the 20-points, two-dimensional toy dataset in Figure 3.3 will be used, on which we suppose to run the algorithm with parameters $\epsilon = \sqrt{2}$ and $minPts = 5$.

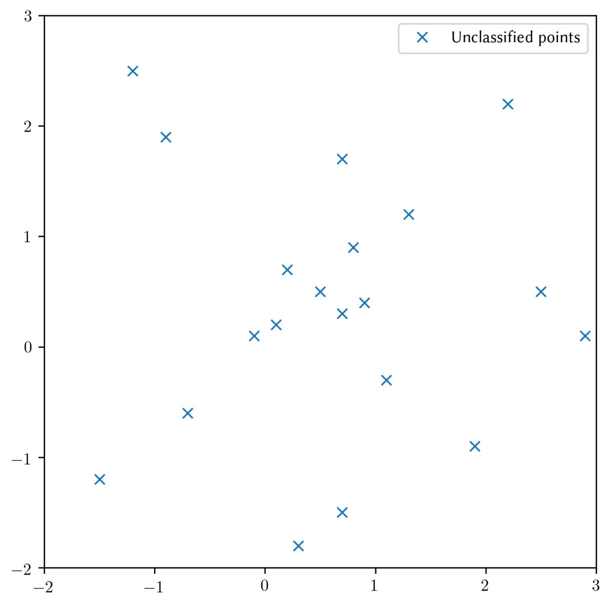

Figure 3.3. Overview of the example dataset

3.3 Cell construction and grid definition

3.3.1 Overview

The cell construction and grid definition phase is aimed at parsing of the input dataset from secondary memory and defining the cells that will be used throughout the different steps of the process. In particular, the procedure implementing this step performs the following actions:

- It loops through the points in the input dataset;
- For each point, it computes its cell coordinates by applying the formula: $c_i = \left\lfloor p_i * \frac{\sqrt{d}}{\epsilon} \right\rfloor$;
- It assigns each point to its corresponding cell, by constructing a key-value map having as key the cell coordinates and as value the list of points belonging to such cell.

A principle implementation of the procedure is shown in Algorithm 3.2.

Algorithm 3.2 Grid definition procedure

```
function CREATE-GRID($inputFile, dim, eps, minPts$)
    $allPoints \leftarrow$ EMPTY-DICT

    for $point$ in $inputFile$ do
        $cell \leftarrow$ EMPTY-LIST($dim$)

        ▷ Compute cell coordinates
        for $i$ in $[1, dim]$ do
            $cell[i] \leftarrow$ FLOOR($point[i] * \frac{\sqrt{dim}}{eps}$)
        end for

        ▷ Assign the point to the cell
        $allPoints[cell] \leftarrow$ APPEND($allPoints[cell], point$)
    end for

    return $allPoints$
end function
```

3.3.2 Time complexity

It is possible to prove the following theorem.

Theorem 3.3.1. The cell construction and grid definition phase runs in $O(n)$ worst-case time complexity, where n is the size of the input dataset, for any data dimensionality.

Proof. The algorithm loops through all points once, computing the cell coordinates by applying a single floating-point operation to all its components. Assuming a $O(1)$ processing time for each point, the overall complexity is therefore:

$$n * O(1) = O(n) \tag{3.12}$$

□

3.3.3 Example

The results of running the grid definition phase on the example dataset are shown in Figure 3.4. As it can be seen, since the value of the parameter ϵ is set to $\sqrt{2}$, all the points have been subdivided into cells with a side length of 1, identified by the coordinates of its bottom left vertex: in this case, the value of the scaling factor is indeed $\frac{\epsilon}{\sqrt{d}} = \frac{\sqrt{2}}{\sqrt{2}} = 1$.

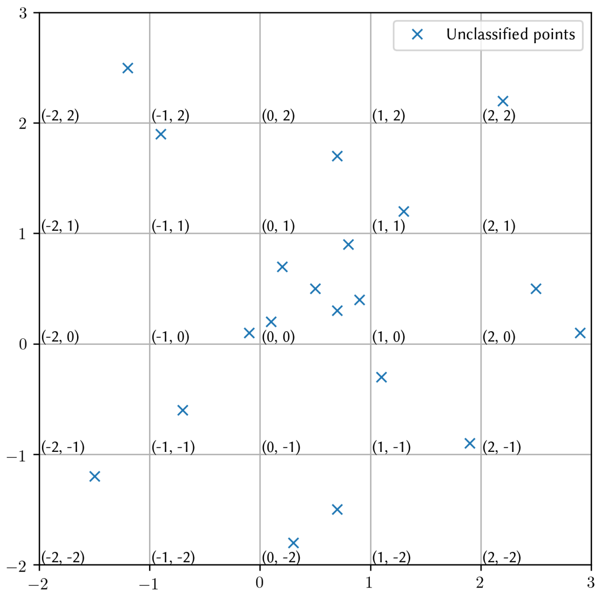

Figure 3.4. Results of the grid definition phase on the example dataset

3.3.4 Parallelization

The parallelization of the grid definition procedure is straightforward; indeed, in this phase, each point can be processed independently of the others by simply applying a MapToPair transformation. The output of such transformation is a collection of key-value pairs in the form (*cell coordinates, point*), where the cell coordinates are computed as in the sequential algorithm.

As an implementation concern, we additionally have to handle with care all the cases in which repeated entries (i.e. points with the same coordinates) are present in the input dataset. Subsequent steps, indeed, make use of ReduceByKey transformations to aggregate the partial results computed for each point; grouping based solely on its coordinates would combine all the values referring to its multiple instances, thus causing the algorithm to produce incorrect outcomes. This may be avoided by introducing a ZipWithUniqueId transformation to assign to each point a unique identifier; in this way, by grouping based on such identifier, it is possible to distinguish between instances with the same coordinates, thus providing correct results for each individual data point.

3.4 Core points identification

3.4.1 Overview

In the sense of the DBSCAN algorithm, a point is *core* whenever its ϵ-neighborhood contains at least *minPts* points. Based on the results of Theorem 3.1.1, for each cell one of the following two situations can verify:

- The considered cell is *dense*: in this case, no additional processing is necessary, since Theorem 3.1.1 allows us to immediately state that all the points in the cell are core;
- The considered cell is *non-dense*: in this case, it is necessary to compute the distance of each point inside the cell to all the points in its neighboring cells (including the cell itself); if the point has at least *minPts* neighbors (i.e. points in neighboring cells having distance less than ϵ), then the point is to be considered core.

This observation justifies the procedure in Algorithm 3.3 for the identification of core points in the dataset. The procedure loops through all cells; if the cell is dense, then the list of points in the cell is copied in the dictionary of core points. Otherwise, the algorithm loops through all the vectors v_1 in the cell; for each of them, it considers all the vectors v_2 in the neighboring cells (generated via the Generate-Neighbors procedure) and checks the distance from v_1 to v_2. If that is less than ϵ, a counter is incremented; if the final value of the counter is at least *minPts*, then the point is core and it is appended to the list of core points of the cell.

3.4.2 Time complexity

It is possible to prove the following theorem.

Theorem 3.4.1. The core points identification phase runs in $O(n)$ worst-case time complexity, where n is the size of the input dataset, for any data dimensionality.

Proof. The function loops through all the cells via variable c_1; assuming that size computation can be executed in constant time (e.g. a length variable is stored for each cell), the following situations can verify.

Algorithm 3.3 Core points identification procedure

```
function FIND-CORE-POINTS($allPoints, dim, eps, minPts$)
    $corePoints \leftarrow$ EMPTY-DICT

    ▷ Loop through all cells
    for $cell$ in $allPoints$ do
        $numPts \leftarrow$ SIZE($allPoints[cell]$)

        if $numPts \geq minPts$ then
            ▷ Dense cell, all points are core points
            $corePoints[cell] \leftarrow allPoints[cell]$
        else
            $neighbors \leftarrow$ GENERATE-NEIGHBORS($cell$)

            ▷ For all points in the cell...
            for $v_1$ in $allPoints[cell]$ do
                $countNeighbors \leftarrow 0$

                ▷ ... check all the neighboring cells ...
                for $neighbor$ in $neighbors$ do
                    ▷ ... consider all the points in each neighbor ...
                    for $v_2$ in $allPoints[neighbor]$ do
                        ▷ ... if the distance between the two vectors is less than $\epsilon$ ...
                        if DISTANCE($v_1, v_2$) $< eps$ then
                            ▷ ... then increment the number of neighbors
                            $countNeighbors \leftarrow countNeighbors + 1$
                        end if
                    end for
                end for

                ▷ Point is core if it has at least $minPts$ neighbors
                if $countNeighbors \geq minPts$ then
                    $corePoints[cell] \leftarrow$ APPEND($corePoints[cell], v_1$)
                end if
            end for
        end if
    end for

    return $corePoints$
end function
```

Case 1: c_1 is dense. In this case, we just need to copy a pointer to the entire list of points of the cell into the core points map. Hence, the time complexity for any number of dense cells is:

$$O(n) * O(1) = O(n) \tag{3.13}$$

Case 2: c_1 is not dense. In this case, all the points within the cell, which are at most $minPts - 1$, need to be compared to each point in the neighboring cells. The number of comparisons we need to perform is therefore:

$$\sum_{c_1} \sum_{v_1 \in c_1} \sum_{c_2 \in \mathcal{N}(c_1)} \sum_{v_2 \in c_2} 1 \tag{3.14}$$

Operating the same substitution as in Gunawan [8], we note that $c_2 \in \mathcal{N}(c_1) \Leftrightarrow c_1 \in \mathcal{N}(c_2)$; as a consequence, we can write:

$$\sum_{c_1} \sum_{v_1 \in c_1} \sum_{c_2 \in \mathcal{N}(c_1)} \sum_{v_2 \in c_2} 1 = \sum_{c_2} \sum_{v_2 \in c_2} \sum_{c_1 \in \mathcal{N}(c_2)} \sum_{v_1 \in c_1} 1 \tag{3.15}$$

Given that $|c_1| = O(minPts)$ and $|\mathcal{N}(c_2)| = k_d$, we have:

$$\sum_{c_2} \sum_{v_2 \in c_2} \sum_{c_1 \in \mathcal{N}(c_2)} \sum_{v_1 \in c_1} 1 = \sum_{c_2} \sum_{v_2 \in c_2} O(minPts * k_d) \tag{3.16}$$

Finally, since $\sum_{c_2} \sum_{v_2 \in c_2} 1 = n$, we obtain:

$$\sum_{c_2} \sum_{v_2 \in c_2} O(minPts * k_d) = O(n * minPts * k_d) \tag{3.17}$$

The overall worst-case time complexity of this phase is therefore:

$$O(n) + O(n * minPts * k_d) = O(n * minPts * k_d) \tag{3.18}$$

□

3.4.3 Example

In order to illustrate the core points identification phase for the example dataset, let us consider what happens for two example cells with coordinates $c_1 = (0, 0)$ and $c_2 = (1, -1)$.

As for cell c_1, it may be immediately noticed that the cell is dense: indeed, it contains six points, a value which is greater than $minPts = 5$. Therefore, cell c_1 is marked as dense and all of its points as core, as shown in Figure 3.5.

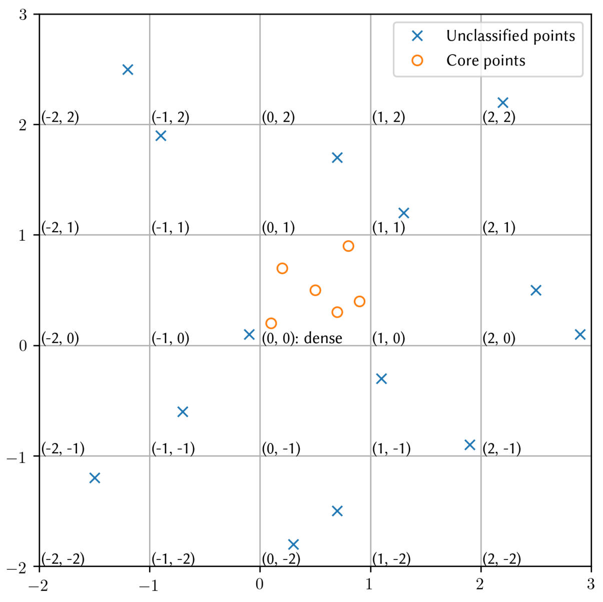

Figure 3.5. Dense cells marking on the example dataset

As for cell c_2, the cell is not dense as it contains two points only; consequently, it is necessary to check, for both of them, the distance with all the points in the neighboring cells. The results are as follows:

- As shown in Figure 3.6, point $p_1 = (1.1, -0.3)$ happens to have nine neighbors (those pointed by a green arrow), a value which is greater than $minPts$; thus, the point and its cell are marked as core;

- Conversely, as shown in Figure 3.7, point $p_2 = (1.9, -0.9)$ happens to have only two more points within its ϵ-neighborhood, which means that the point is not core. Note that, whereas several points are included in the neighboring cells of c_2, many of them (those pointed by a red arrow) do not effectively lie in the ϵ-neighborhood of point p_2.

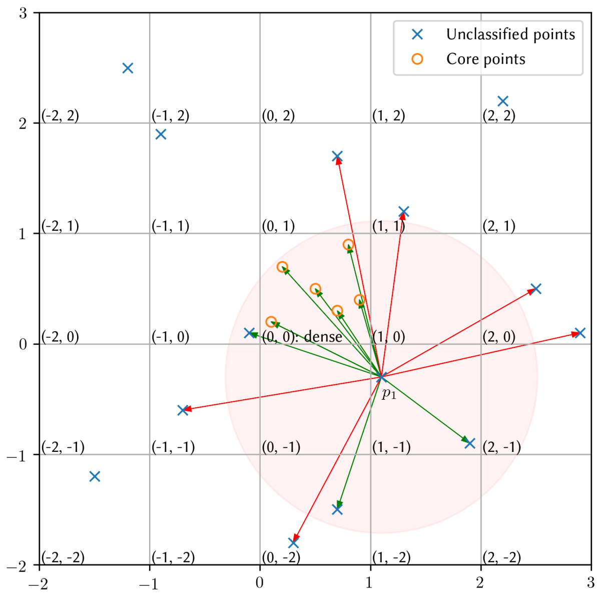

Figure 3.6. Neighbor check for the point p_1 on the example dataset

All the other cells and points are processed as described above, yielding the result shown in Figure 3.8, in which the ϵ-neighborhoods of the identified core points are also drawn.

3.4.4 Parallelization

The parallelization of the core points identification phase is more complex due to the necessity of computing the distances between points belonging to different cells, whenever those are not dense. In order to accomplish this goal, the adopted strategy is to emit the points to be checked as tuples in the form (*neighbor*, *point*) so that, by means of a JOIN operation with the original dataset, it is possible to create pairs of vectors for which the distance can be easily computed.

This phase can furtherly be subdivided into three consecutive steps:

- *Identification of the dense cells*: for this purpose, it is necessary to count the number of points belonging to each cell through of a REDUCEBYKEY transformation, whose results are used to construct a cell map that is distributed to all executors, storing for each cell identifier its type (dense/non-dense).

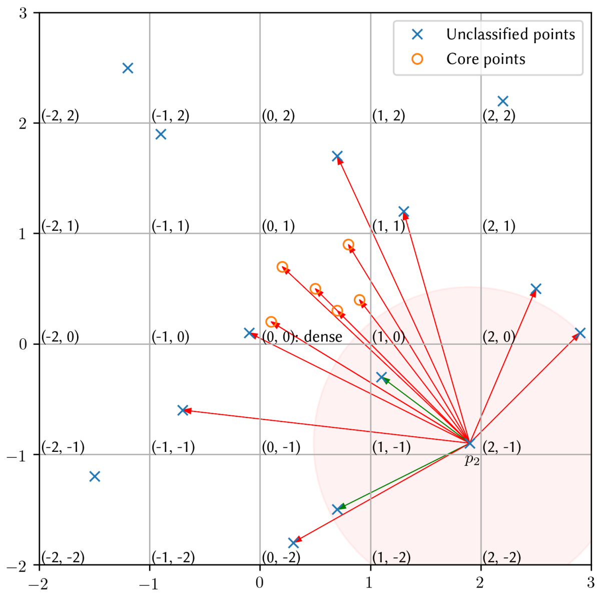

Figure 3.7. Neighbor check for the point p_2 on the example dataset

- *Emission of the points to check*: in order to identify the core points belonging to non-dense cells, the first step is to emit all the points from non-dense cells on their neighbors, so that the required comparisons can be performed. To this purpose, a FlatMapToPair transformation is used; for each vector v_1 belonging to a non-dense cell c_1 (which can be identified by using the information contained in the cell map), a number of pairs in the form $(c_2, (c_1, v_1))$ is emitted, one for each neighbor c_2 of c_1.

- *Identification of core points from non-dense cells*: the points to be checked are joined by key through a Join transformation with the entire dataset in the form (c_2, v_2), thus originating pairs in the form $(c_2, (v_2, (c_1, v_1)))$. The distance between the two points is computed and used, in a MapToPair transformation, for producing pairs having as key the tuple (c_1, v_1) and as value an integer, either 1 (if the distance is at most ϵ) or 0 (otherwise). Finally, a succession of a ReduceByKey and Filter transformations are used to retrieve the number of effective neighbors for each point and select only those having at least *minPts*.

The overall set of core points is then produced by applying a Union transformation to the

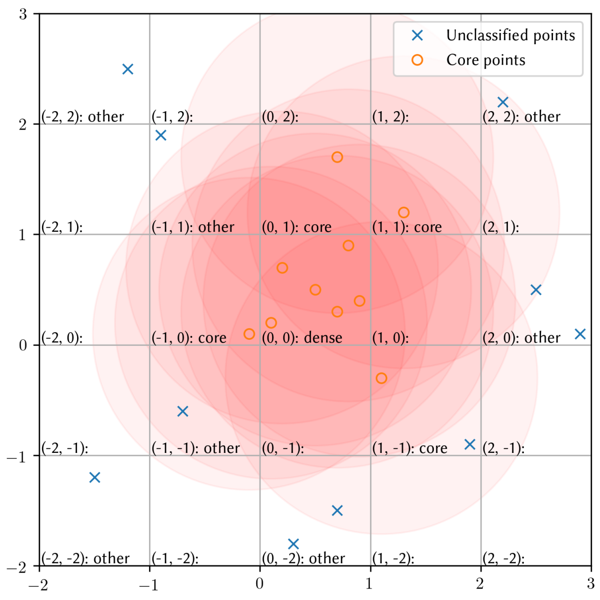

Figure 3.8. Results of the core points identification phase on the example dataset

results of Step 3 with the points belonging to dense cells as identified during Step 1.

3.5 Outliers identification

3.5.1 Overview

Conceptually, the outliers identification phase is very similar to the core points identification phase. In this case, no further processing is needed for the core cells: indeed, thanks to Theorem 3.1.2, we know that they cannot contain any outlier. For all the non-core cells, instead, it is necessary to check the distance to all points in the neighboring cells; the considered point is an outlier if and only if none of those distances is lower than ϵ.

The conceptual implementation of the procedure for the identification of the outliers is shown in Algorithm 3.4.

Algorithm 3.4 Outliers identification procedure

```
function FIND-OUTLIERS($allPoints, corePoints, dim, eps, minPts$)
    $outliers \leftarrow$ EMPTY-DICT

    ▷ Loop through all cells
    for $cell$ in $allPoints$ do
        ▷ Check that the cell does not contain core points
        if $cell$ not in $corePoints$ then
            $neighbors \leftarrow$ GENERATE-NEIGHBORS($cell$)

            ▷ For all points in the cell...
            for $v_1$ in $allPoints[cell]$ do
                $isOutlier \leftarrow true$

                ▷ ... check all the neighboring cells ...
                for $neighbor$ in $neighbors$ do
                    ▷ ... consider all the core points in each neighbor ...
                    for $v_2$ in $allPoints[neighbor]$ do
                        ▷ ... if the distance between the two vectors is less than $\epsilon$ ...
                        if DISTANCE($v_1, v_2$) $< eps$ then
                            ▷ ... then the point is not an outlier
                            $isOutlier \leftarrow false$
                        end if
                    end for
                end for

                ▷ Point is outlier if it has no core neighbor
                if $isOutlier$ then
                    $outliers[cell] \leftarrow$ APPEND($outliers[cell], v_1$)
                end if
            end for
        end if
    end for

    return $outliers$
end function
```

3.5.2 Time complexity

It is possible to prove the following theorem.

Theorem 3.5.1. The outliers identification phase runs in $O(n)$ worst-case time complexity, where n is the size of the input dataset, for any data dimensionality.

Proof. The algorithm loops through every cell. If the cell is core, then no operation needs to be performed; otherwise, all the points in the cell (at most $minPts - 1$, since a non-core cell is certainly not dense) need to be compared to all the points in the neighboring core cells (at most k_d). Thus, the overall complexity is equivalent to the one of the core points identification phase:

$$O(n * minPts * k_d) \tag{3.19}$$

□

3.5.3 Example

Also in this case, in order to illustrate the outliers identification phase on the toy dataset, an example cell with coordinates $c = (0, -2)$ has been chosen. This cell is not core (otherwise all of its non-core points can immediately be classified as border points, since they fall within the ϵ-neighborhood of at least a core point) and has two points, for which we need to check the distance from all the core points in the neighboring cells:

- As shown in Figure 3.9, point $p_3 = (0.7, -1.5)$ happens to have one core point within its ϵ-neighborhood, which is a sufficient condition not to classify it as an outlier; the point will consequently be marked as a border point;
- Point $p_4 = (0.3, -1.8)$, instead, happens to have all the core points in the nearby cells at a distance greater than the parameter ϵ (see Figure 3.10); thus, it is classified as an outlier.

The same procedure is applied for every non-core cell in order to identify the outliers in the dataset. The final result is shown in Figure 3.11.

3.5.4 Parallelization

The parallelization of the outliers identification phase follows the same ideas of the core points identification phase. It can be structured in four steps:

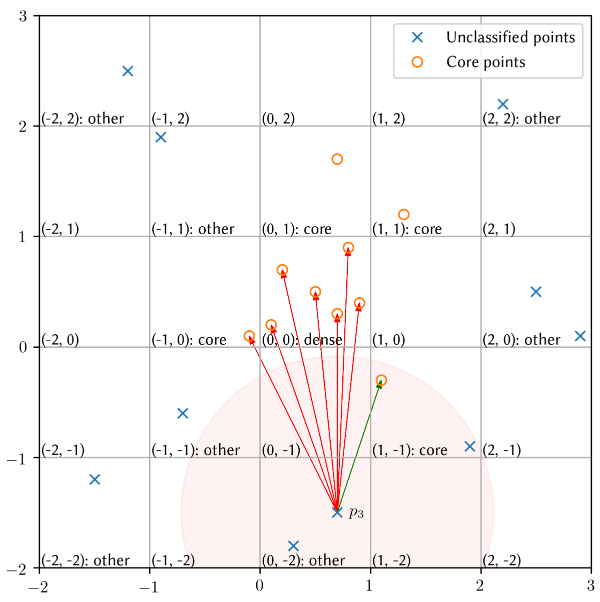

Figure 3.9. Neighbor check for the point p_3 on the example dataset

- *Identification of the core cells*: the cell map constructed during the previous phase is updated by marking as core all the cells which appear at least once in the set of core points, then broadcast to all the executors.

- *Identification of outliers from non-core cells with no core neighbors*: the updated cell map is used to identify the non-core cells having no core neighbor by means of a Filter transformation; all the points belonging to these cells are outliers, since they are surely not in the neighborhood of a core point.

- *Emission of the points to check*: similarly to what happens in the previous phase, in this case the points belonging to non-core cells need to be emitted on all their neighbors in order to compute distances.

- *Identification of outliers from non-core cells with at least one core neighbor*: the points to be checked are joined by key through a Join transformation with the set of core points, in order to compute the distances between each generated pair. Each point to be checked

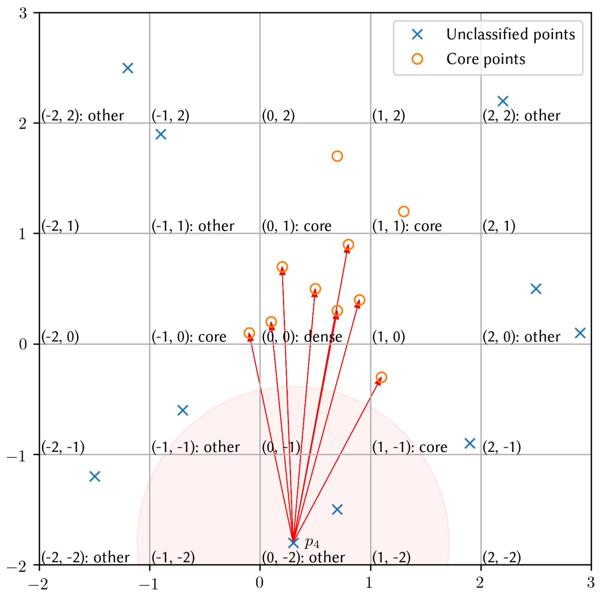

Figure 3.10. Neighbor check for the point p_4 on the example dataset

is associated with a boolean value, true if the distance is at least ϵ or false otherwise, through MapToPair transformation. Finally, a succession of a ReduceByKey and Filter transformations is used to select only the points for which, after all the reduction by means of a boolean AND operation, the flag remains true.

The entire set of outliers can be obtained by applying a Union transformation to the results of Step 2 and 4.

3.6 Implementation optimizations

3.6.1 Optimizations overview

The algorithm as described up to now works as expected, and always returns the expected outcomes for any combination of input parameters. However, its performance may not be optimal for some specific cases; in the following paragraphs, some optimizations which allow to reduce the running time while keeping the overall logic of the algorithm are presented. All

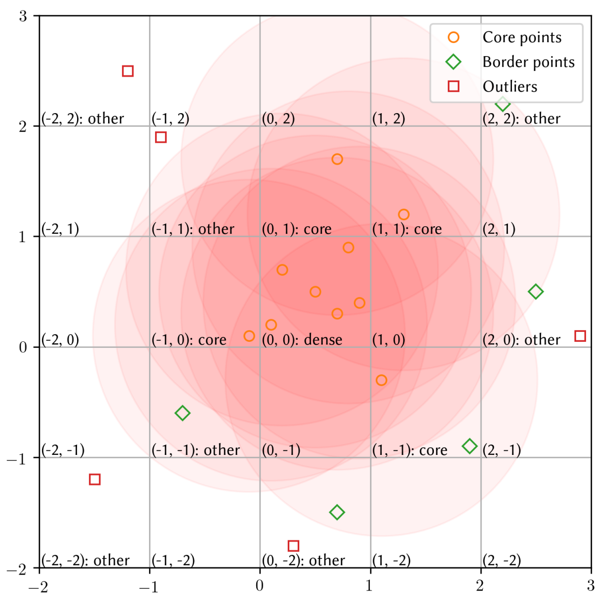

Figure 3.11. Results of the outliers identification phase on the example dataset

those techniques focus specifically on reducing the running time of the JOIN operations, which are the most computationally expensive in the entire flow.

3.6.2 Broadcast join

In some cases, the number of the points to check in the core points and outlier identification phases is small enough to collect them in a local map. The broadcast join optimization allows to eliminate the costly JOIN transformation by collecting them in an opportune structure, which is then broadcast to all executors; this can be used to perform the join operation by means of a FLATMAPTOPAIR transformation.

Since local Java maps do not allow for repeated keys, a GROUPBYKEY operation is also needed in order to generate pairs (*cell*, [*points*]), which are then collected and broadcast. The FLATMAP-TOPAIR transformation, applied to the entire dataset, checks whether the cell is present in the map; if so, it computes the distance between the current vector and all the vectors associated to the cell, emitting a pair ((*cell*, *vector*), 0/1) for each of them. From this point on, the algorithm proceeds as before.

The effect of such optimization is variable; as a general trend, it performs best for higher values of ϵ: indeed, in this case more cells are dense and less points need to be emitted on nearby cells for distance computation. However, since its application may generate out-of-memory errors at runtime whenever the map of points to be checked does not fit in memory, this optimization is disabled by default in the developed code.

3.6.3 Grouping before joining

During the performed tests, it has been possible to notice that the performances of the Join transformation decrease almost linearly with the size of the data to be joined. The purpose of this optimization, therefore, is to consistently reduce the cardinality of one of the join operands when the value of ϵ is lower and, consequently, the number of points to be checked is higher. This is achieved by applying, immediately before the join, a GroupByKey transformation on the dataset side, whose cardinality is generally lower than the one of the points to be checked, especially for low ϵ.

Although the quality of such optimization from a conceptual point of view may be disputed due to the inherent complexity of the grouping operation, in practice it allows to obtain speedups up to 500% for certain datasets and low values of ϵ, while not severely affecting performances with high ϵ. Moreover, grouping before joining allows to reduce the number of comparisons for each point in the average case; it is indeed possible to stop computing the distance from the considered point to the vectors in the joined cell if:

- The number of the point's neighbors reaches the target value of *minPts* during the core points identification phase;
- The point is discovered not to be an outlier due to the presence of a neighboring core point during the outliers identification phase.

For these reasons, it has been chosen to always apply the grouping before joining optimization, independently of the type of dataset and of the value of the different parameters.

Chapter 4

Experimental results

In this chapter, the algorithm described in Chapter 3 is tested on multiple datasets and with different parameters combinations in order to characterize the quality of the results and the scalability of its performance in big data scenarios. After a brief description concerning the implementation of the algorithm, the following sections present the results of the executed tests and discuss how they compare to the ones obtained by means of reference and state-of-the-art outlier detection and parallel DBSCAN techniques.

4.1 Algorithm implementation

The parallel version of the proposed algorithm was implemented using the Java Spark APIs. The choice of such programming language was dictated by the performance penalties introduced by the usage of a non-JVM native language (such as Python) in conjunction with a JVM-native library such as Spark (written in Scala), deriving from the communication cost between the language interpreter and the Java Virtual Machine. For comparison, a sequential version of the algorithm was also implemented, which has been used in the first set of tests due to its higher performance on small-sized datasets (there is no overhead related to scheduling and communication among nodes); the Java language was used in this case as well, in order to facilitate code sharing between the two versions.

For the parallel algorithm, an optional parameter was additionally introduced to control the number of partitions in which to split the input data; if not specified, the number of HDFS chunks of the input file is used. It is worth noticing that such parameter does not influence the results of the computation, which is indeed always dependent only on ϵ and *minPts*; rather, it may improve the degree of parallelism that is achieved by the system, thus reducing the running time.

4.2 Overview of the performed tests

In order to evaluate the performance of the proposed algorithm, a number of tests were designed, aiming to characterize:

- The *quality* of the results, with respect to the ones produced by other well-known outlier detection algorithms; to this purpose, some small, synthetic datasets have been employed, which are manageable by all the tested algorithms and for which quality measures may be simply computed and interpreted;
- The *scalability* of the algorithm with the variation of its parameters, with respect to reference parallel implementations of DBSCAN; to this purpose, some real-world datasets have been employed, which are large enough not to be manageable by traditional, sequential data mining algorithms.

4.3 Quality-related testing

4.3.1 Datasets description

The first set of tests has been performed on synthetic datasets, with the goal to measure the quality of the results produced by the new algorithm on different data distributions and to compare them with available implementations of well-known reference algorithms. Some of these datasets were generated by us, while some others are benchmark datasets extensively used in literature for testing the performances of clustering algorithms. A description of the generated dataset is provided in the following list:

- *Blobs*: a two-dimensional, 10000 points dataset featuring five globular clusters with similar densities with added random noise; 99% of the dataset is generated using scikit-learn's make_blobs, with parameters centers=5, cluster_std=0.5, random_state=100, while the remaining 1% is represented by random, two-dimensional noise;

- *Blobs-vd*: a two-dimensional, 10000 points dataset featuring three globular clusters with varying densities with added random noise; 99% of the dataset is generated using scikit-learn's make_blobs, with parameters centers=3, cluster_std=[0.5, 1, 1.5], random_state=100, while the remaining 1% is represented by random, two-dimensional noise;
- *Circles*: a two-dimensional, 10000 points dataset featuring two circle-shaped, concentric clusters with added random noise; 99% of the dataset is generated using scikit-learn's make_circles, with parameters factor=0.5, noise=0.05, random_state=100, while the remaining 1% is represented by random, two-dimensional noise;
- *Moons*: a two-dimensional, 10000 points dataset featuring two moon-shaped clusters with added random noise; 99% of the dataset is generated using scikit-learn's make_moons, with parameters noise=0.05, random_state=100, while the remaining 1% is represented by random, two-dimensional noise.

As for the benchmark datasets, the following were used:

- *Chameleon* [11]: a collection of four, two-dimensional datasets with 8000-10000 points each, characterized by the presence of oddly-shaped clusters with added random noise;
- *Cure* [7]: a two-dimensional dataset consisting of 4000 points and characterized by the presence of variable-sized circular and elliptic clusters with added random noise.

4.3.2 Tested algorithms

Given the small size of the tested datasets, the sequential version of the algorithm was used to perform all the quality-related tests without incurring in scheduling and synchronization costs, which are predominant with respect to the actual execution time.

In order to evaluate the quality of the detection, the results produced by the new algorithm were compared against those produced by the original DBSCAN (for which we checked that the same set of outliers was being marked as such) and by some of the traditional algorithms for outlier detection presented in Chapter 2 (specifically, Robust Covariance Estimation [19], Isolation Forests [14], Local Outlier Factor [2] and One-Class SVM [20]). For all of them, we used their Python implementation as provided by the scikit-learn [18] library.

4.3.3 Parameter selection

As for the selection of the algorithm parameters, for the first set of tests we adopted the usual DBSCAN estimation technique: we fixed the value of *minPts*, then drawn the graph of the distance to the *minPts*-th neighbor against the number of points; the value of ϵ was then chosen in the uppermost part of the elbow zone of such graph, with some rounding. For the outlier detection algorithms described in Chapter 2, the value of the contamination factor ν was set to the actual proportion of outliers in each dataset (0.01 for those generated by us, variable for the benchmark ones), while the other parameters were left to their default values.

4.3.4 Evaluation metrics

In order to evaluate the quality of the obtained result, the metric of choice for the first set of tests is the *adjusted Rand index* [10], as computed by scikit-learn's `adjusted_rand_score` function. This constitutes a variation of the "standard" Rand index that is "adjusted for chance", according to the formula:

$$ARI = \frac{RI - \mathbb{E}[RI]}{\max RI - \mathbb{E}[RI]} = \frac{RI - \mathbb{E}[RI]}{1 - \mathbb{E}[RI]} \tag{4.1}$$

Where $\mathbb{E}[\cdot]$ is the expected value operator and *RI* the "standard" Rand index, which is computed as:

$$RI = \frac{\text{true positives} + \text{true negatives}}{\text{number of points}} \tag{4.2}$$

Obviously, the evaluation of such parameter requires the knowledge of the "true" labels for each data point, which is in this case assured by the synthetic nature of the datasets. For both the Rand index and its adjusted variant, values close to 0 represent a completely random labeling, while values close to 1 represent a perfect labeling up to a permutation of the identifiers.

In relation to the "standard" Rand index, ARI provides a much more unbiased evaluation of the detection results, since it takes into account the expected similarity between the true and predicted labels as computed by a random model, which is generally not null. This is particularly evident for the outlier detection task, for which we can expect the great majority of data points, whose number is in any case predominant with respect to the number of outliers, to be correctly marked, thus returning an elevate value of the unadjusted index even when the quality of the labeling is clearly poor.

4.3.5 Testing environment

All the sequential code was run on a Windows 10 PC equipped with an Intel Core i7-7700HQ CPU and 16 GB of RAM. For both the Java JDK and the Python interpreter, the latest available versions were used (13.0.2 and 3.8.2, respectively).

4.3.6 Results

A summary of the outcomes of the experiments on the generated datasets is presented in Table 4.1; a visual comparison is shown in Figures 4.1 to 4.4.

In all the tests on the generated datasets, the new algorithm outperformed all the others in terms of result quality. Clearly, performances are better for the datasets with clusters characterized by a homogeneous density, on which ARI values of up to 0.95800 are recorded (on both the *Blobs* and *Moons* dataset); nevertheless, good quality scores are obtained also for the other datasets with an opportune choice of the parameters (the minimum ARI value, registered for the *Blobs-vd* dataset, is 0.85751). In particular, it can be seen that the new algorithm is capable of correctly identifying the great majority of outliers and data points in the dataset; notable exceptions include outliers that are randomly generated in dense areas which no density-based algorithm would ever identify as such (as shown for example on the *Blobs-vd* and *Moons* datasets in Figures 4.2 and 4.4) or by themselves constituting small clusters, as in the *Blobs* dataset (Figure 4.1). A reduced number of errors is also related to unidentified data points located on the border of the dense areas, especially for the *Blobs-vd* dataset in Figure 4.2. In any case, as expected, results for the DBSCAN algorithm match exactly the ones provided by the new algorithm, once that cluster labels have been removed.

As for the other tested algorithms, the worst performing overall is the statistical Robust Covariance Estimation-based method (with ARI values as low as 0.06939 for the *Blobs-vd* dataset), since it assumes an elliptical shape for the data which is not reflected in any of the tested datasets, leading to a large number of misclassified points. The Isolation Forest and One-Class SVM algorithms provide mixed results, performing better on the datasets featuring globular clusters: on the *Blobs* dataset, the two algorithms obtain an ARI value of 0.78450 and 0.74115 respectively. Isolation Forests fail in particular on the *Circles* dataset (ARI value: 0.09917), in which the outliers are indeed very close to actual data points; the same trend can be observed for the One-Class SVM (ARI value: 0.22872), which is not able to fit an opportune separating hyperplane for the two classes. The results provided by the Local Outlier Factor algorithm are much more consistent among the different executions (with ARI values consistently higher than 0.8), but the algorithm suffers (due to its density-based nature) in the test using clusters with varying densities, obtaining an ARI score of 0.69270.

Table 4.1. Rand index comparison for the generated datasets

Dataset	Algorithm	Parameters	ARI
Blobs	New	$\epsilon = 0.6, minPts = 5$	0.95800
	Robust Covariance Estimation	$\nu = 0.01$	0.37906
	Isolation Forest	$\nu = 0.01$	0.78450
	Local Outlier Factor	$\nu = 0.01$	0.86638
	One-Class SVM	$\nu = 0.01$	0.74115
Blobs-vd	New	$\epsilon = 0.8, minPts = 5$	0.85781
	Robust Covariance Estimation	$\nu = 0.01$	0.06938
	Isolation Forest	$\nu = 0.01$	0.63169
	Local Outlier Factor	$\nu = 0.01$	0.69270
	One-Class SVM	$\nu = 0.01$	0.73729
Circles	New	$\epsilon = 0.02, minPts = 5$	0.87112
	Robust Covariance Estimation	$\nu = 0.01$	0.18878
	Isolation Forest	$\nu = 0.01$	0.09917
	Local Outlier Factor	$\nu = 0.01$	0.81518
	One-Class SVM	$\nu = 0.01$	0.22872
Moons	New	$\epsilon = 0.02, minPts = 5$	0.95800
	Robust Covariance Estimation	$\nu = 0.01$	0.43946
	Isolation Forest	$\nu = 0.01$	0.33888
	Local Outlier Factor	$\nu = 0.01$	0.93824
	One-Class SVM	$\nu = 0.01$	0.59420

Results for the experiments on benchmark datasets are summarized in Table 4.2 and shown in Figures 4.5-4.9. Also in this case, the new algorithm demonstrates good performance on all the datasets (ARI values range from the 0.78507 obtained for the *cluto-t8-8k* dataset to the

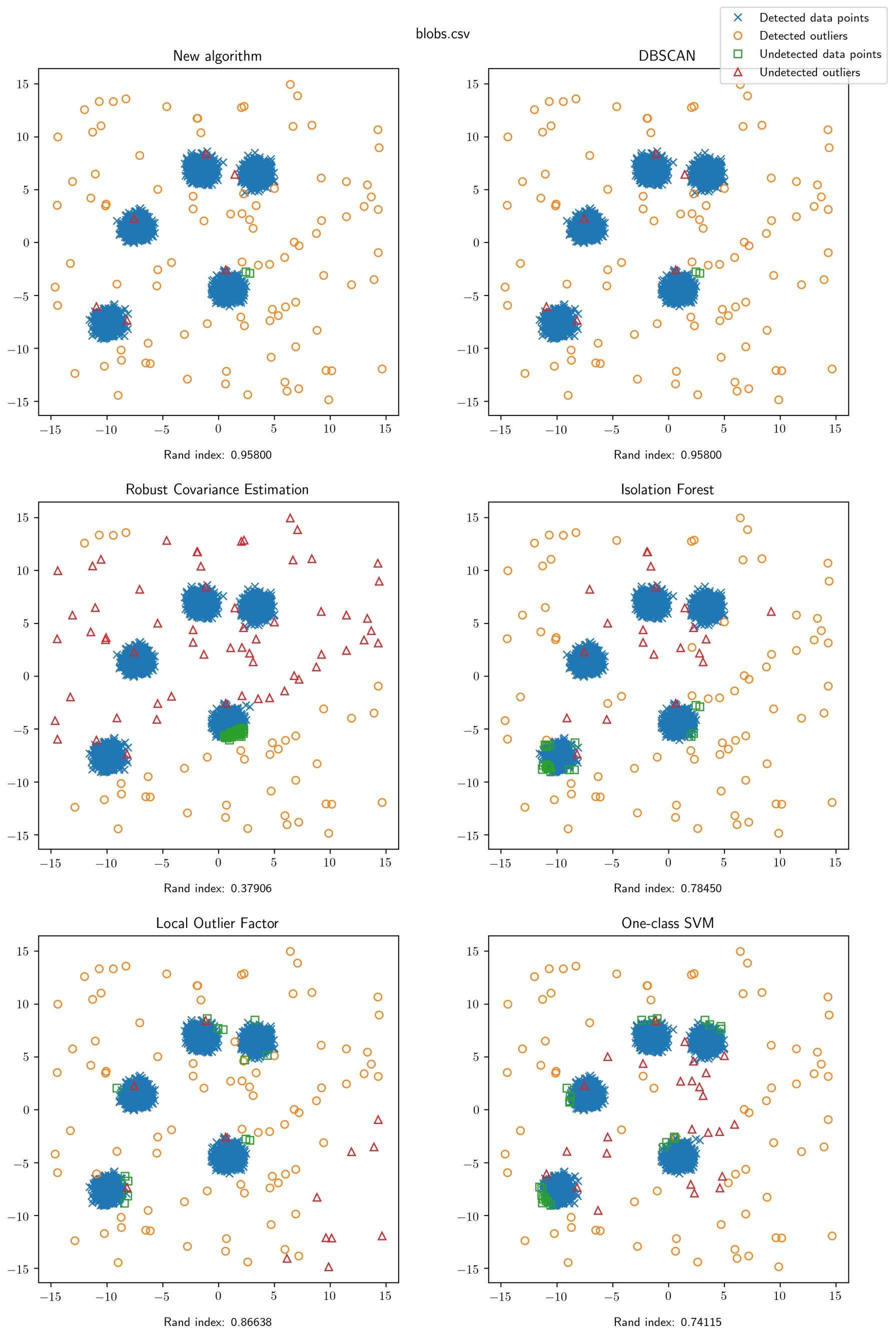

Figure 4.1. Results on the *Blobs* dataset

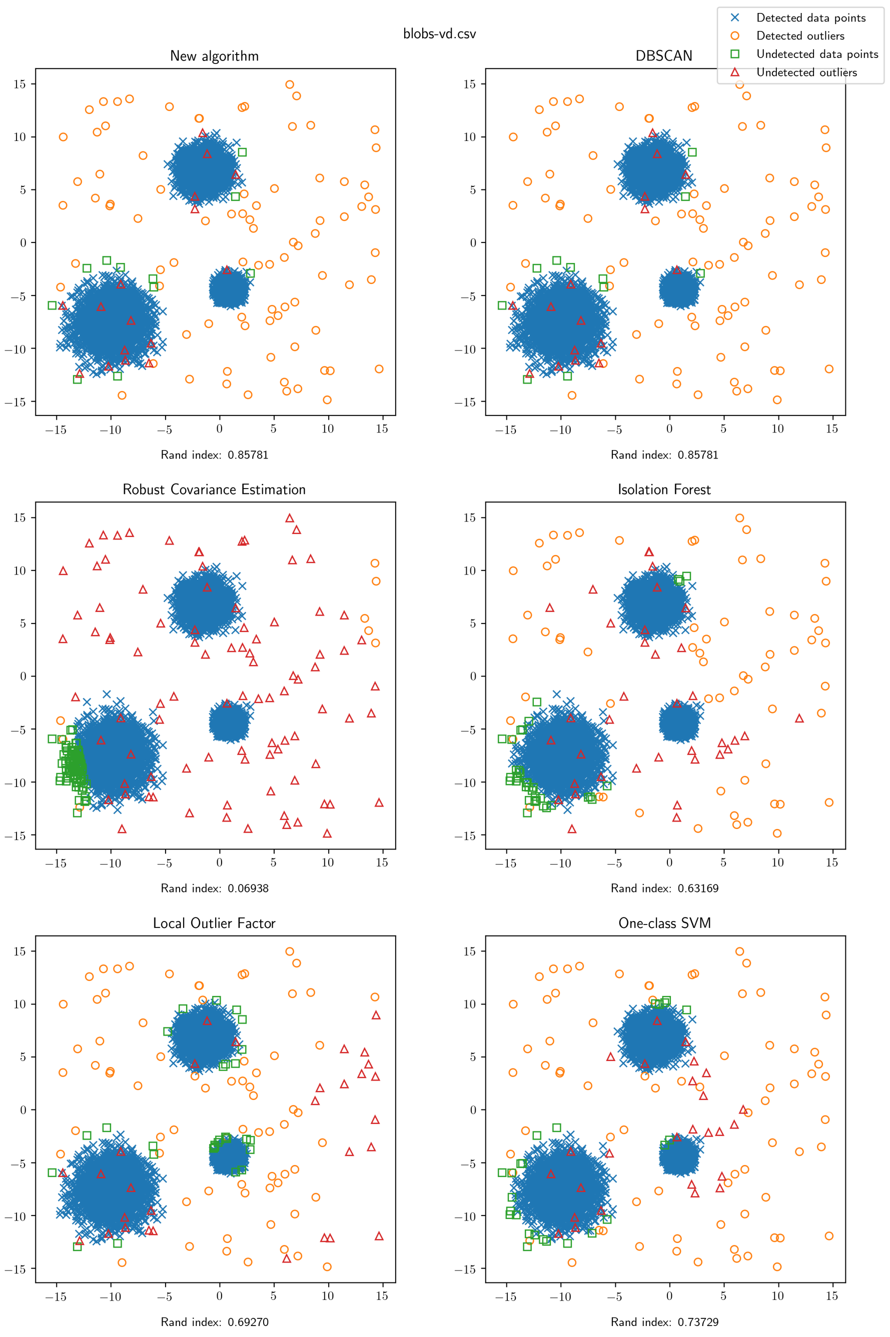

Figure 4.2. Results on the *Blobs-vd* dataset

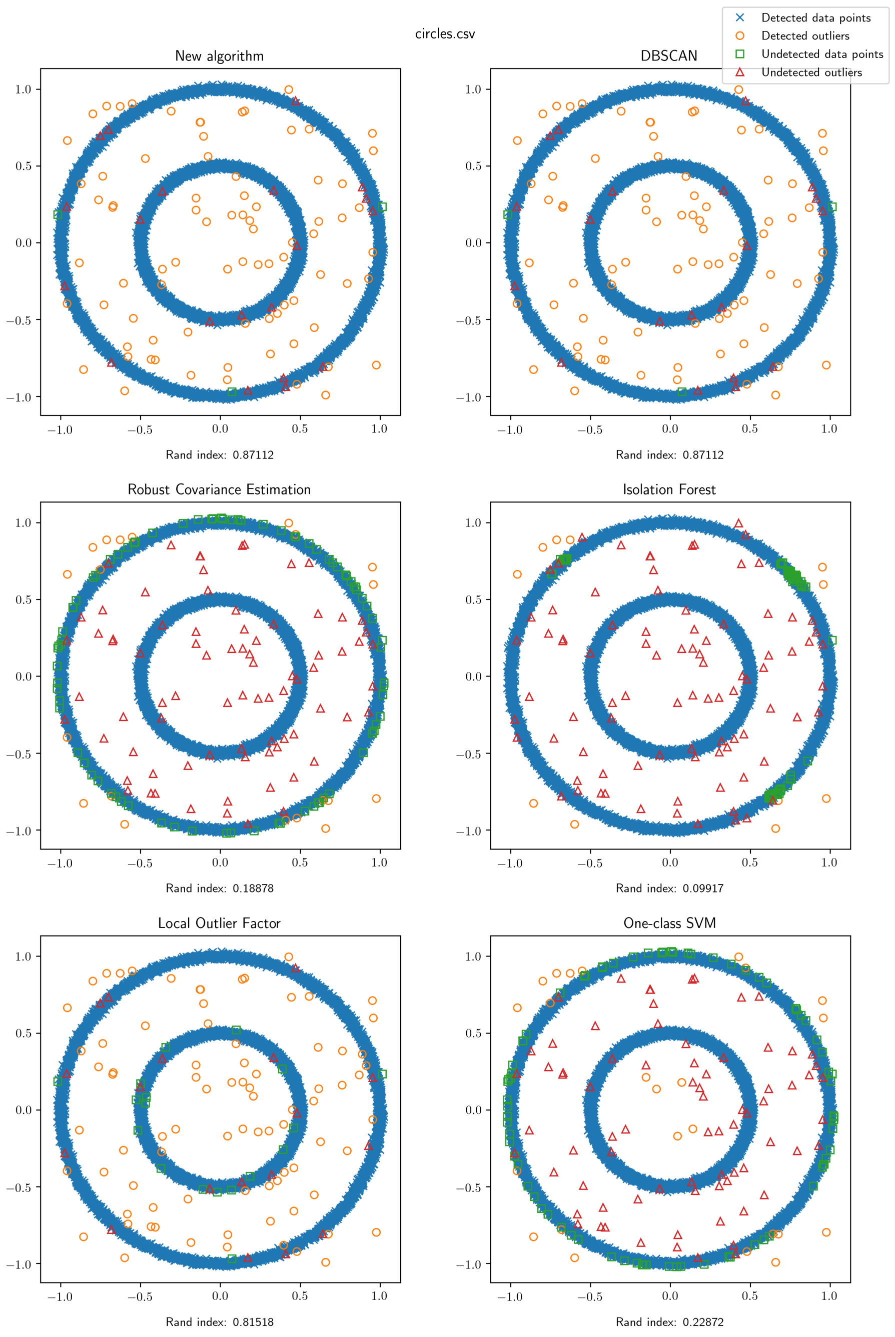

Figure 4.3. Results on the *Circles* dataset

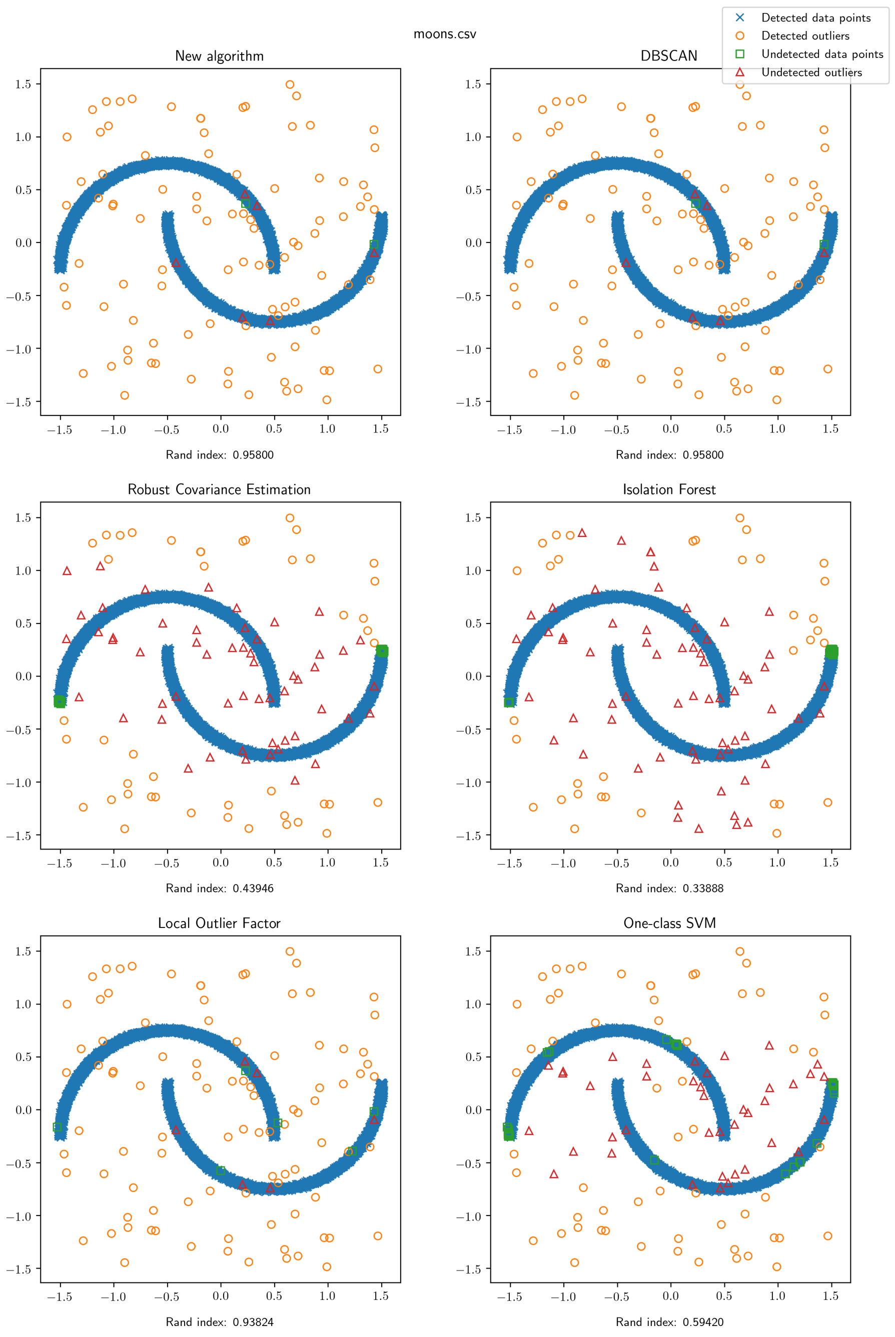

Figure 4.4. Results on the *Moons* dataset

0.88192 obtained for the *cluto-t7-10k* dataset), being outperformed by Local Outlier Factor only on *cluto-t8-8k* (registering an ARI score of 0.87300) due to the presence of clusters with different densities with outliers very close to one another. Again, problems are mainly related to very clustered outliers (such as in the *cluto-t4-8k* and *cluto-t5-8k* datasets in Figures 4.5 and 4.6) and undetected data points on the clusters' borders (like in the *cluto-t8-8k* dataset, in Figure 4.8). The worst-performing is still the statistical method (with an absolute lowest of 0.02769 ARI score for the *cure-t2-4k* dataset), whereas the results of both Isolation Forests and One-Class SVMs confirm to be very dependent on the distribution of the data, yet less promising than in the previous tests (ranging from the very poor 0.10351 obtained by the One-Class SVM on the *cluto-t5-8k* dataset to the average 0.54303 obtained by means of an Isolation Forest on the *cluto-t4-8k* dataset).

Table 4.2. Rand index comparison for the benchmark datasets

Dataset	Algorithm	Parameters	ARI
cluto-t4-8k	New	$\epsilon = 7, minPts = 10$	0.80787
	Robust Covariance Estimation	$\nu = 0.1$	0.28230
	Isolation Forest	$\nu = 0.1$	0.54303
	Local Outlier Factor	$\nu = 0.1$	0.49339
	One-Class SVM	$\nu = 0.1$	0.43017
cluto-t5-8k	New	$\epsilon = 5, minPts = 10$	0.80705
	Robust Covariance Estimation	$\nu = 0.15$	0.16433
	Isolation Forest	$\nu = 0.15$	0.27899
	Local Outlier Factor	$\nu = 0.15$	0.60001
	One-Class SVM	$\nu = 0.15$	0.10351
cluto-t7-10k	New	$\epsilon = 10, minPts = 10$	0.88192
	Robust Covariance Estimation	$\nu = 0.08$	0.17878
	Isolation Forest	$\nu = 0.08$	0.26501
	Local Outlier Factor	$\nu = 0.08$	0.70620
	One-Class SVM	$\nu = 0.08$	0.22965
cluto-t8-8k	New	$\epsilon = 12, minPts = 10$	0.78507
	Robust Covariance Estimation	$\nu = 0.04$	0.14175
	Isolation Forest	$\nu = 0.04$	0.36608
	Local Outlier Factor	$\nu = 0.04$	0.87300
	One-Class SVM	$\nu = 0.04$	0.31859
cure-t2-4k	New	$\epsilon = 0.08, minPts = 10$	0.87480
	Robust Covariance Estimation	$\nu = 0.05$	0.02769
	Isolation Forest	$\nu = 0.05$	0.26621
	Local Outlier Factor	$\nu = 0.05$	0.78362
	One-Class SVM	$\nu = 0.05$	0.10577

All in all, the new algorithm was capable of providing good-quality results for each of the tested datasets, without requiring any domain knowledge for parameter estimation (e.g. it is not needed to know an estimate of the proportion of outliers in the dataset). For our tests, a very simple technique has been used to decide the value of the algorithm parameters; we believe that possibly even better quality figures may be obtained using a more thorough approach to parameter estimation.

4.4 Scalability-related testing

4.4.1 Datasets description

The second set of tests, aimed at demonstrating the scalability of the algorithm with very large amounts of data, was performed using some publicly available, real-world datasets extensively adopted in the related literature. Those include:

- *Geolife* [26]: a collection of GPS trajectories from 182 users over a five years period (April 2007 to August 2012), consisting in a series of timestamped three-dimensional points (latitude, longitude, altitude in feet); although the dataset contains points from different areas of the world, a very high concentration of tracks has been registered around the city of Beijing, which makes the dataset heavily skewed. As a preprocessing step, all the trajectory files were joined together upfront in the form of a CSV file in which all the unnecessary features were removed.

- *OpenStreetMap* [17]: a set of sample GPS points collected by OpenStreetMap contributors over the first seven and a half year of the platform; it consists of a series of sorted, latitude-longitude pairs in the form of integer values obtained by multiplying the actual coordinates by 10^7.

4.4.2 Tested algorithms

In order to evaluate the scalability, the performances of the parallel algorithm were compared to the ones of RP-DBSCAN [22], given that such approximated algorithm was shown to outperform all of the other parallel implementations of DBSCAN by large margins. The code for

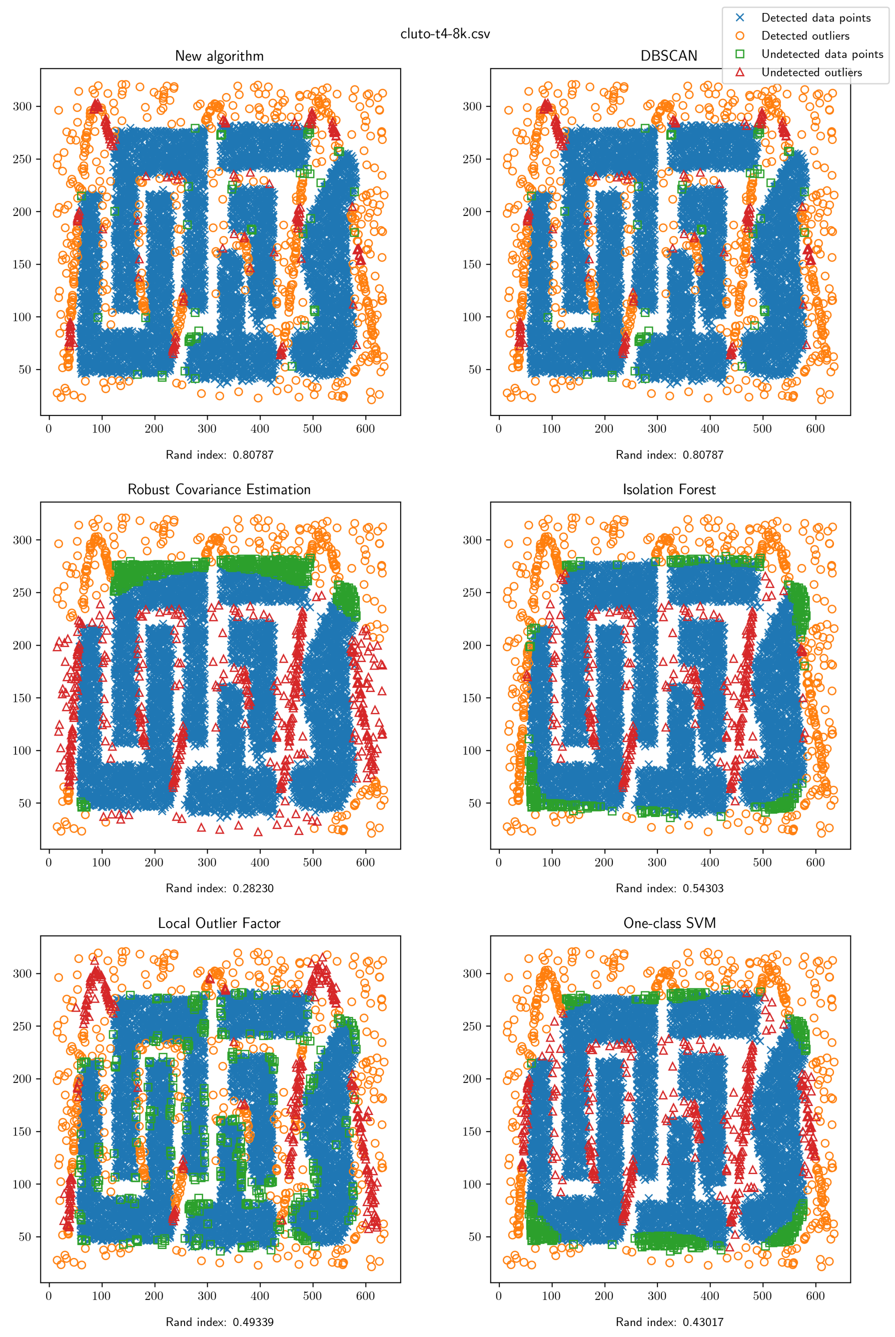

Figure 4.5. Results on the *cluto-t4-8k* dataset

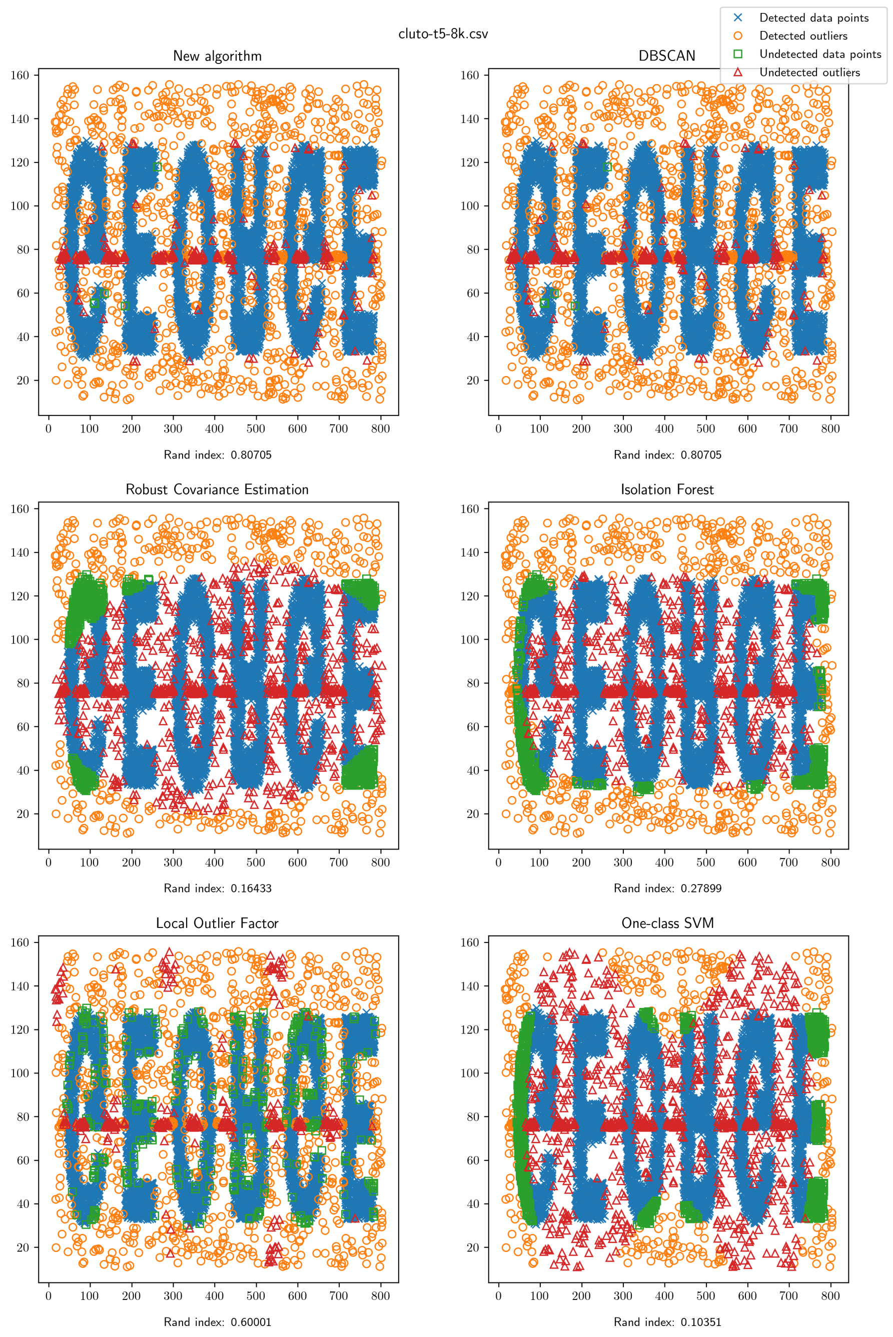

Figure 4.6. Results on the *cluto-t5-8k* dataset

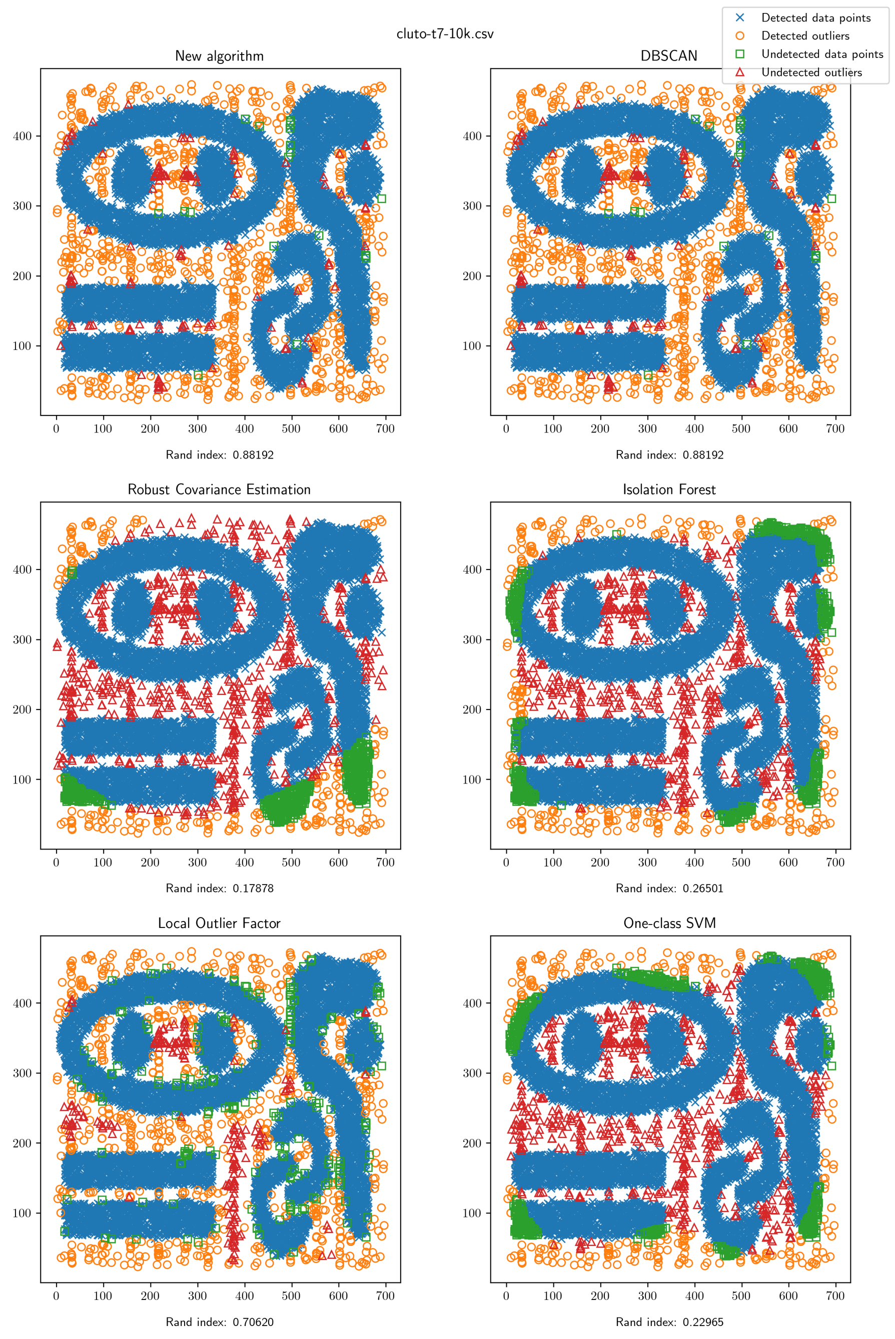

Figure 4.7. Results on the *cluto-t7-10k* dataset

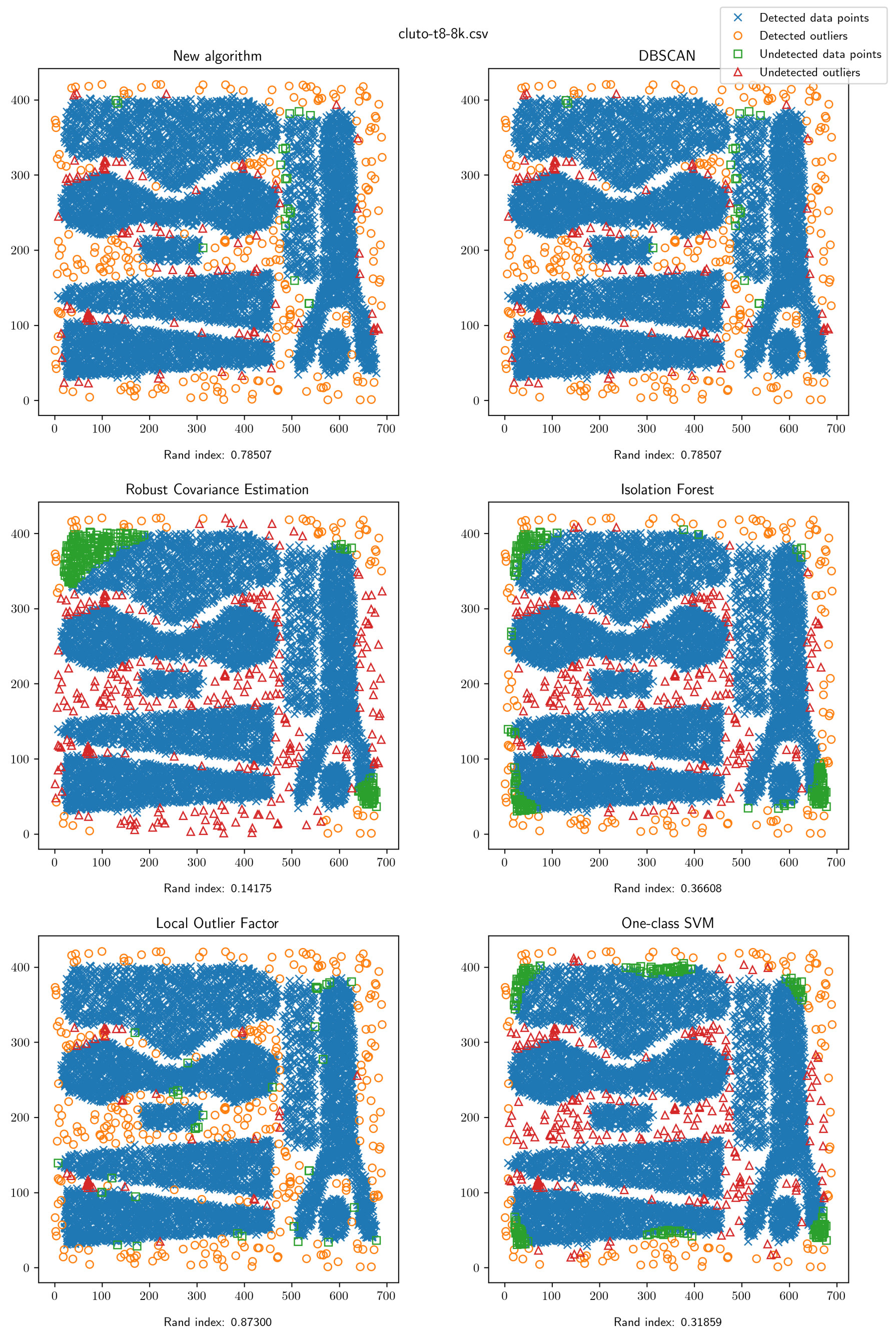

Figure 4.8. Results on the *cluto-t8-8k* dataset

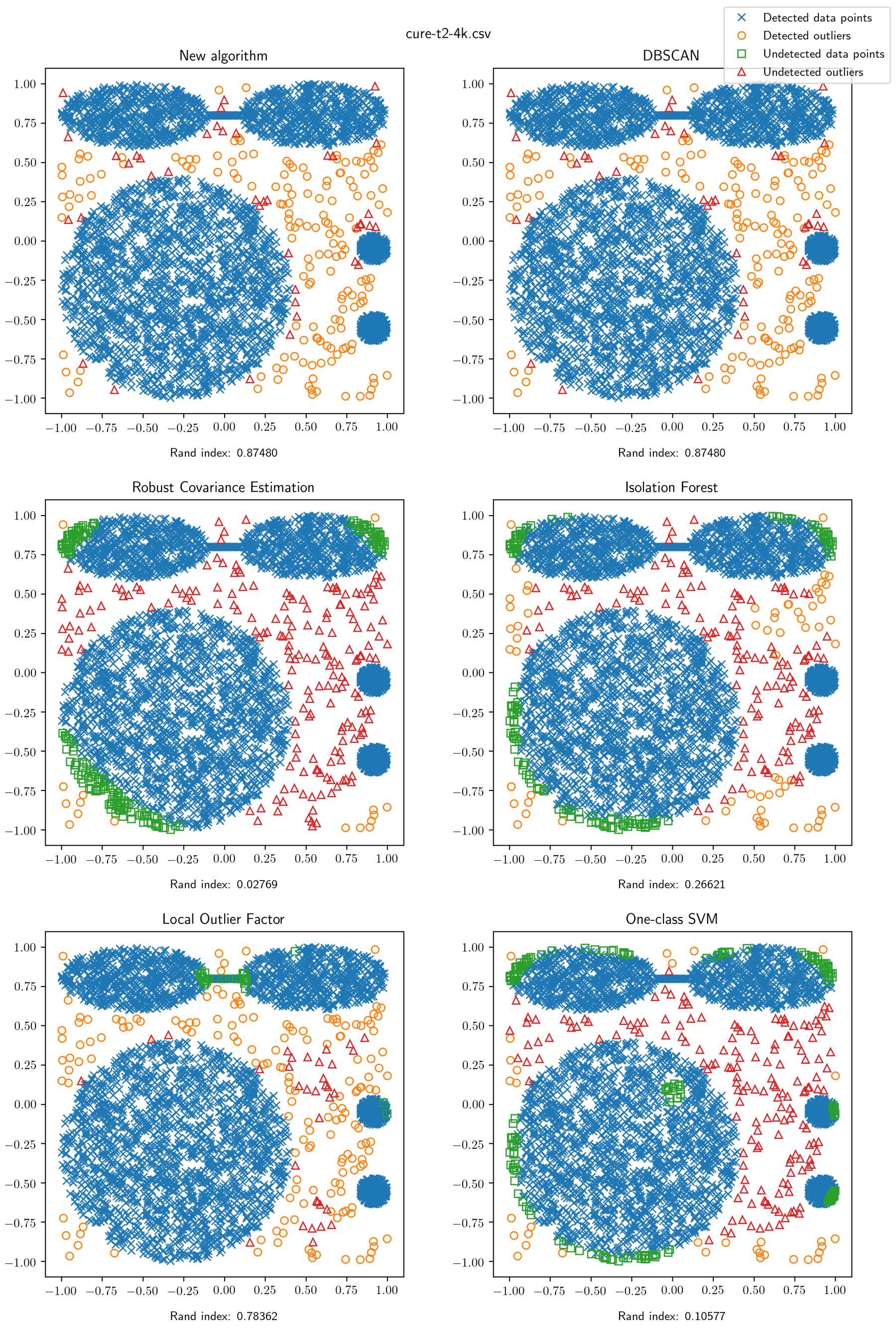

Figure 4.9. Results on the *cure-t2-4k* dataset

RP-DBSCAN was obtained by recompiling the code publicly made available by the authors (latest version to date is from January 28th, 2020), after applying the following modifications:

- The cluster configuration was updated to reflect the available one;
- The output file writing phase was updated to provide only the results relative to outliers (those having the cluster identifier set to -1).

4.4.3 Parameters selection

For the second set of tests, the algorithm was run by fixing *minPts* to 100 for all the executions, while testing multiple values of the other parameters (such as ϵ and the number of partitions) to characterize the overall scalability. The choice of *minPts* is in line with the one made by RP-DBSCAN authors. A general conception, which is also reflected by some initial tests performed on the new algorithm, is that the impact of such parameter is indeed minimal if compared to ϵ. As for RP-DBSCAN, the approximation parameter was fixed to $\rho = 0.01$ in all runs, as suggested by the authors.

4.4.4 Evaluation metrics

In this case, the metric of choice has been the execution time (in seconds) as reported by the Spark web interface. All the tests were run at least five times so to derive general trends which do not depend on the specific load factor of the cluster at any given moment; the average and the standard deviation were used as aggregate values for each series of tests. In addition, due to the approximation introduced by RP-DBSCAN, a qualitative comparison between the results produced by the two algorithms was also performed.

4.4.5 Testing environment

The parallel code was tested on the machines of the *SmartData@Polito* cluster located at Politecnico di Torino, Italy; a description of the available computing facilities can be found on the laboratory's website [21]. A specific Yarn queue was dedicated to the execution of the jobs, to which a total of 100 CPU *vcores* and 800 GB of main memory were statically assigned. Two allocation schemes of such dedicated resources have been defined:

- *Configuration #1*: this configuration uses 100 executors, each consisting of a single CPU core and 8 GB of main memory;
- *Configuration #2*: this configuration uses 50 executors, each consisting of two CPU cores and 16 GB of main memory.

All tests on the smaller *Geolife* dataset were performed using the first configuration, while both configurations were tested on the larger *OpenStreetMap* dataset. Note that, on such dataset, jobs using RP-DBSCAN could not run in the first configuration due to memory limitations (Yarn containers fail due to memory shortage). The Spark version is 2.4.0, running on Scala 2.11.12; the JVM version is 1.8.0_181. For all jobs, the following additional configuration options were specified in `spark-submit`:

- The maximum result size (`spark.driver.maxResultSize`) was set to `0` (unlimited), to avoid problems related to the size of the results collected by the driver program;
- The object serializer (`spark.serializer`) was set to use `KryoSerializer`, which has better performance with respect to the standard Java serializer;
- The network timeout (`spark.network.timeout`) was set to `300s`, in order to support the execution of the most complex stages without timeout-related failures.

4.4.6 Qualitative result analysis

Figures 4.10 and 4.11 show the results obtained by running the algorithm on a two-dimensional version of the *Geolife* and on the *OpenStreetMap* datasets respectively. Parameters were set to $\epsilon = 0.1$ and $\epsilon = 10^6$ to correspond to a precision of 0.1 degrees along the latitude/longitude axes, while in both cases $minPts = 100$ was used.

From an exclusively qualitative point of view, it can be noticed that the algorithm effectively provides a good quality result, marking as outliers all of those points which do not belong to dense areas. These include, for the most part, points which do not lie within any of the continents, corresponding to flights or other forms of travel by sea; small groups of observations recorded in very isolated locations (such as Antartica) are also correctly identified. In addition to that, the algorithm detects much of the noise that is present in the *OpenStreetMap* dataset, including a strip of evenly distributed interference that is recorded around the central part of the globe (described also in the dataset's documentation).

All in all, these results justify, in our view, the applicability of the proposed algorithm for the mining of outliers in large datasets in a density-based fashion.

4.4.7 Scalability with respect to ϵ

A first set of tests was run to compare the performances of the proposed algorithm (in terms of running time) with those of RP-DBSCAN, fixing *minPts* to 100 and the number of partitions to the quantity of the HDFS chunks. For both datasets, four values of ϵ were chosen in the neighborhood of those used in the original RP-DBSCAN paper (some higher, some lower). Tables 4.3 and 4.4, along with the corresponding Figures 4.12 and 4.13, present the results of the execution of the new algorithm and of RP-DBSCAN on the two datasets. Some sample statistics were also computed for both datasets in correspondence of each tested value of ϵ; they are summarized in Tables 4.5 and 4.6.

As for the *Geolife* dataset, no general trend may be derived; depending on the specific ϵ, either the new algorithm or RP-DBSCAN happens to be slightly faster than the other. Indeed, due to the significant skewness, with such selection of parameters most points fall within very few cells (in the case with $\epsilon = 200$, 40% of points are assigned to the most populous one); in a way, this facilitates the work of RP-DBSCAN (which summarizes points at the cell level), but instead impacts on the performance of the new algorithm, since these very dense cells happen to participate in average in several join operations with their neighbors (depending on the value of ϵ). We believe that, with a wiser parameter choice (e.g. taking into account domain-specific considerations), the algorithm performances could scale much better than shown in the executed tests.

The analysis of the larger *OpenStreetMap* dataset provides very different results, in which the new algorithm shows significant performance advantage with respect to RP-DBSCAN for almost all tested values of the parameter ϵ, which is more and more evident the lower ϵ gets. Both algorithms show a decreasing running time for increasing values of ϵ, which is somehow expected due to the reduction in the number of cells. Moreover, in this case the size of the densest ones has less impact because, due to the higher uniformity in data distribution, there

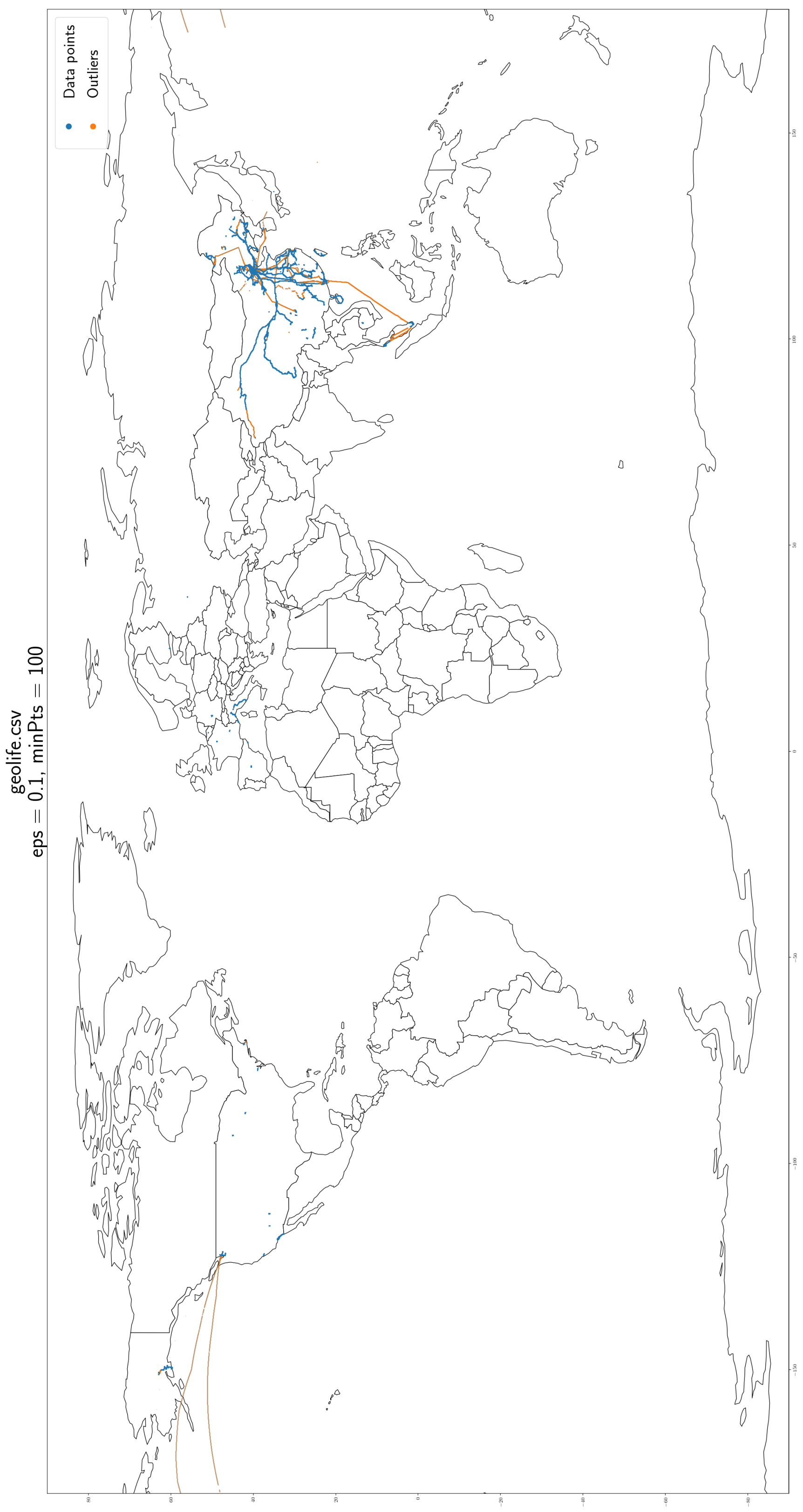

Figure 4.10. Results on the *Geolife* dataset

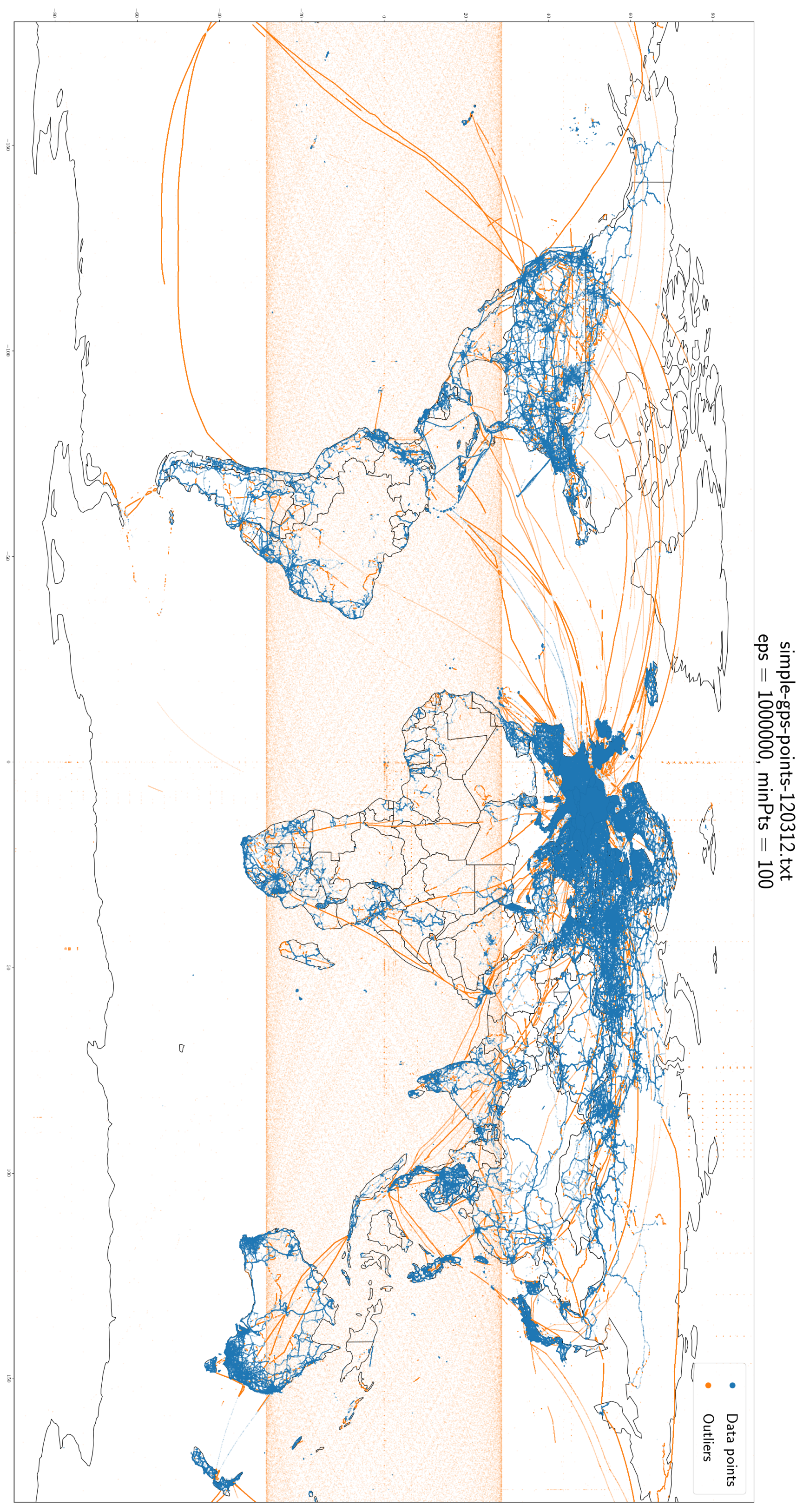

Figure 4.11. Results on the *OpenStreetMap* dataset

Table 4.3. Running times (in seconds) for the *Geolife* dataset with variable ϵ

RP-DBSCAN							
ϵ	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
25	45.40	45.11	47.18	47.93	44.79	46.082	1.387432881
50	44.50	42.82	44.55	42.78	43.70	43.670	0.862960022
100	43.41	39.75	47.75	44.68	44.50	44.018	2.880272557
200	42.95	49.18	50.61	50.90	52.66	49.260	3.738134562

New algorithm							
ϵ	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
25	54.34	58.59	55.91	59.58	60.00	57.684	2.455041751
50	40.83	43.66	44.46	44.60	41.58	43.026	1.754563099
100	41.85	38.99	38.55	41.50	39.10	39.998	1.692835393
200	51.56	53.72	52.17	52.10	58.23	53.556	2.733958668

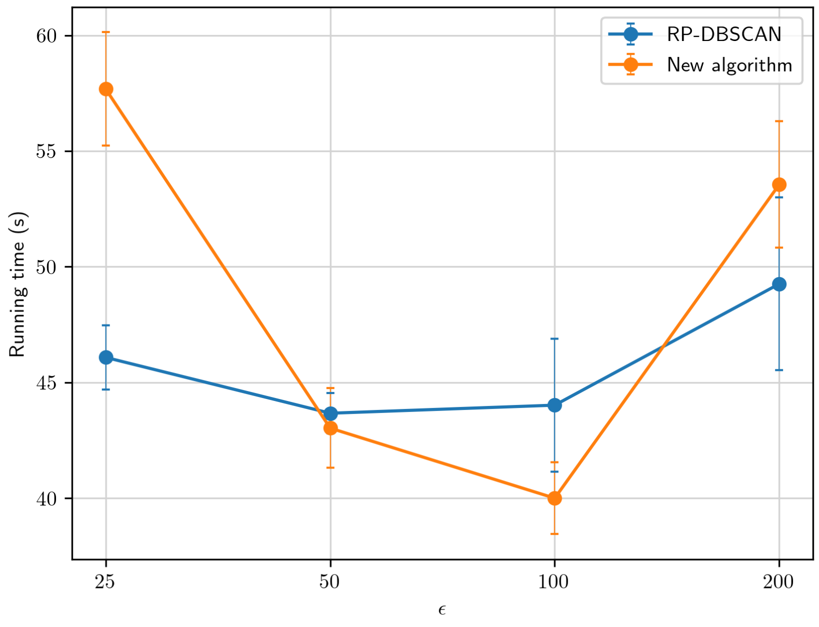

Figure 4.12. Performance on the *Geolife* dataset with variable ϵ

Table 4.4. Running times (in seconds) for the *OpenStreetMap* dataset with variable ϵ

RP-DBSCAN							
ϵ	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
250000	4305	4331	4556	4297	4211	4340.0	128.8914272
500000	1806	1870	1910	1780	1836	1840.4	51.40817056
1000000	1148	1145	1126	1152	1076	1129.4	31.47697571
2000000	780	675	748	714	791	741.6	45.10266068

New algorithm (configuration #1)							
ϵ	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
250000	1018	1093	1029	1028	987	1031.0	38.60699418
500000	863	898	830	905	850	869.2	31.83865575
1000000	753	739	726	745	745	741.6	10.03992032
2000000	723	730	744	765	738	740.0	16.07793519

New algorithm (configuration #2)							
ϵ	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
250000	1029	951	947	971	970	973.6	32.81463088
500000	773	773	834	767	798	789.0	27.84780063
1000000	768	707	765	758	737	747.0	25.42636427
2000000	672	715	699	674	674	686.8	19.30543965

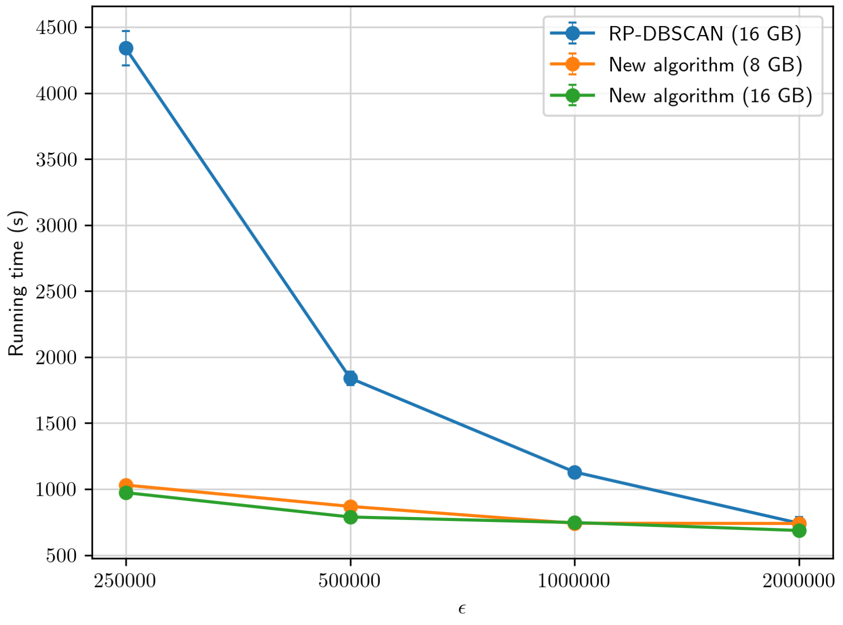

Figure 4.13. Performance on the *OpenStreetMap* dataset with variable ϵ

Table 4.5. Statistics for the *Geolife* dataset

Category	Parameter	$\epsilon = 25$	$\epsilon = 50$	$\epsilon = 100$	$\epsilon = 200$
Points	Total	24876978 (100%)	24876978 (100%)	24876978 (100%)	24876978 (100%)
	Core	24844886 (99.87%)	24858744 (99.92%)	24867347 (99.96%)	24873503 (99.99%)
	Border	6440 (0.03%)	3405 (0.02%)	2881 (0.01%)	977 (<0.01%)
	Outlier	25652 (0.10%)	14829 (0.06%)	6750 (0.03%)	2498 (0.01%)
Cells	Total	17510 (100%)	7563 (100%)	3230 (100%)	1685 (100%)
	Dense	2941 (16.80%)	1613 (21.33%)	798 (24.71%)	449 (26.65%)
	Core	5434 (31.03%)	2650 (35.04%)	1044 (32.32%)	605 (35.90%)
	Other	9135 (52.17%)	3300 (43.63%)	1388 (42.97%)	631 (37.45%)
Points per cell	Max	3445913	3671480	6236325	9995454
	Min	1	1	1	1
	Avg	1420.730	3289.300	7701.851	14763.785
Neighbors per cell	Max	45	26	31	15
	Min	1	1	1	1
	Avg	19.128	13.173	9.060	9.540

are very few of them that (in average) need to take part in a join operation (i.e. neighbors of dense cells are very likely to be dense as well). The new algorithm running on configuration #2 achieves small advantages over the one running on configuration #1, possibly due to the deployment of couple of executors on the same physical machines, which to a certain extent improves data locality while not affecting much job parallelism.

All in all, the proposed algorithm shows good scalability in terms of parameter selection, especially for large datasets, allowing to mine with ease anomalies in a density-based fashion for wide ranges of input parameters.

4.4.8 Scalability with respect to the number of points

A second set of tests was designed to characterize the scalability of the two algorithms with respect to the number of points in the input data. To this purpose, random samples were

Table 4.6. Statistics for the *OpenStreetMap* dataset

Category	Parameter	$\epsilon = 250000$	$\epsilon = 500000$	$\epsilon = 1000000$	$\epsilon = 2000000$
Points	Total	2770233904 (100%)	2770233904 (100%)	2770233904 (100%)	2770233904 (100%)
	Core	2762662319 (99.73%)	2767287625 (99.89%)	2768896461 (99.95%)	2769656678 (99.98%)
	Border	2227934 (0.08%)	747881 (0.03%)	253302 (0.01%)	70840 (<0.01%)
	Outlier	5343651 (0.19%)	2198398 (0.08%)	1084141 (0.04%)	506386 (0.02%)
Cells	Total	3660554 (100%)	1695943 (100%)	835796 (100%)	445231 (100%)
	Dense	1531376 (41.83%)	779818 (45.98%)	357612 (42.79%)	152947 (34.35%)
	Core	1010148 (27.60%)	348925 (20.57%)	125354 (15.00%)	53748 (12.07%)
	Other	1119030 (30.57%)	567200 (33.45%)	352830 (42.21%)	238536 (53.58%)
Points per cell	Max	2725412	3320727	5918794	13568291
	Min	1	1	1	1
	Avg	756.780	1633.448	3314.486	6222.015
Neighbors per cell	Max	21	21	21	21
	Min	1	1	1	1
	Avg	10.580	10.354	9.846	10.344

extracted at 1%, 25%, 50% and 75% of the size of the original datasets and used to run the two algorithms. The parameters were set to $\epsilon = 100, minPts = 100$ for the *Geolife* samples, $\epsilon = 1000000, minPts = 100$ for the *OpenStreetMap* samples. Results are shown in Tables 4.7 and 4.8 and in the corresponding figures 4.14 and 4.15.

As in the previous set of tests, performances on the *Geolife* dataset are very variable depending on the sample size; in the two extremes, the new algorithm performs better on average, while in the other cases RP-DBSCAN results to be the fastest between the two. We believe this is due to the high skewness of the dataset, which is statistically preserved through the sampling operation.

Results are again much more promising on the larger *OpenStreetMap* dataset, in which the new algorithm runs consistently faster than RP-DBSCAN for each value of the sample fraction. Furthermore, it also appears to scale much better, showing an approximately linear evolution of the running time with respect to the sample size. A further small improvement is achievable

Table 4.7. Running times (in seconds) for the *Geolife* dataset with variable sample size

RP-DBSCAN							
Sample size	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
1%	22.99	21.84	19.13	19.63	19.55	20.628	1.691898342
25%	24.42	24.48	22.95	24.32	23.83	24	0.640429543
50%	32.63	31.58	30.75	31.7	32.69	31.87	0.808919032
75%	36.51	34.31	38.76	34.7	36.97	36.25	1.80639143
100%	43.41	39.75	47.75	44.68	44.5	44.018	2.880272557

New algorithm							
Sample size	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
1%	19.58	18.58	17.72	18.69	19.88	18.89	0.86040688
25%	28.44	29.79	29.27	29.71	27.64	28.97	0.916487861
50%	34.77	34.91	33.83	34.26	33.68	34.29	0.54758561
75%	38.17	37.57	37.15	37.33	39.31	37.906	0.874231091
100%	41.85	38.99	38.55	41.5	39.1	39.998	1.549603175

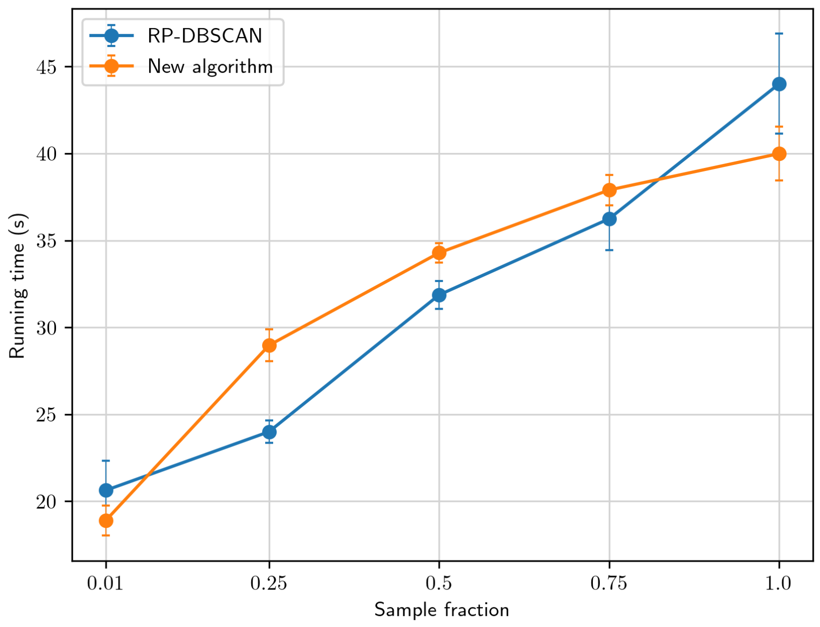

Figure 4.14. Performance on the *Geolife* dataset with variable sample size

Table 4.8. Running times (in seconds) for the *OpenStreetMap* dataset with variable sample size

RP-DBSCAN							
Sample size	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
1%	202	219	214	221	218	214.8	7.596051606
25%	752	713	694	685	723	713.4	26.29258451
50%	847	792	891	787	783	820.0	47.46577715
75%	1022	1121	1027	1048	1132	1070.0	52.63553932
100%	1148	1145	1126	1152	1076	1129.4	31.47697571

New algorithm (configuration #1)							
Sample size	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
1%	119	105	114	105	108	110.2	6.140032573
25%	235	248	248	258	242	246.2	8.497058314
50%	316	309	338	346	334	328.6	15.51773179
75%	497	545	523	505	563	526.6	27.47362371
100%	753	739	726	745	745	741.6	10.03992032

New algorithm (configuration #2)							
Sample size	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
1%	102	108	109	101	103	104.6	3.646916506
25%	204	206	204	201	210	205.0	3.31662479
50%	293	307	302	297	311	302.0	7.280109889
75%	435	447	396	466	429	434.6	25.79341001
100%	768	707	765	758	737	747.0	25.42636427

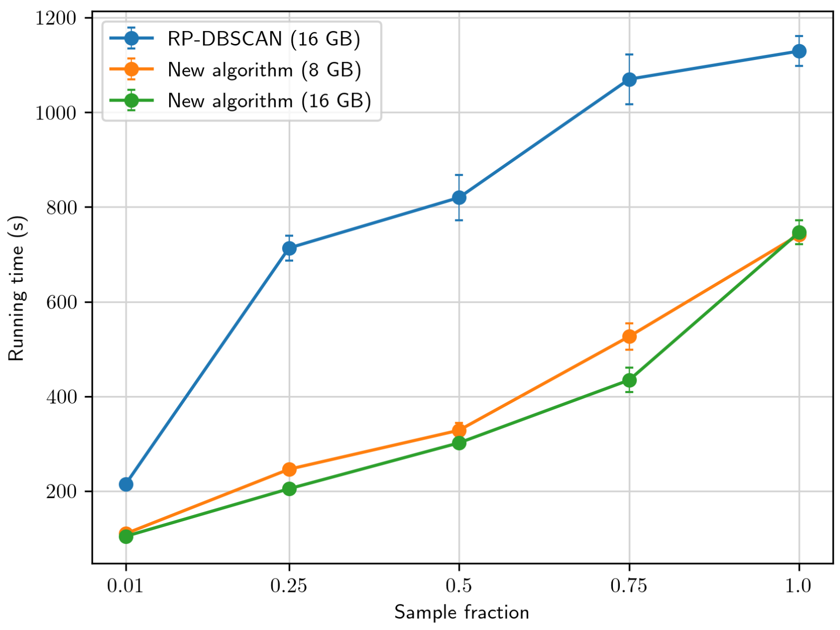

Figure 4.15. Performance on the *OpenStreetMap* dataset with variable sample size

by running the new algorithm using configuration #2, justified by the same reasons as above.

To sum up, in this case as well the algorithm shows good scalability in relation to the number of data points in the input set; the larger the input dataset is, the more consistent the advantage the algorithm achieves over RP-DBSCAN is.

4.4.9 Scalability with respect to the number of partitions

The final set of tests was aimed at the characterization of the scalability of the new algorithm with respect to the number of partitions. In this case, the assumption to test is that, by dividing the input data in smaller chunks, the overall running time decreases due to the faster processing of such partitions (if the recombination cost does not improve significantly). All tests were run using as parameters $minPts = 100$ and $\epsilon = 100$ (for *Geolife*) or $\epsilon = 1000000$ (for *OpenStreetMap*), while varying the value of the analyzed parameter by means of the appropriate switches for the two algorithms.

Again, results on the *Geolife* dataset do not seem to show any relevant trend in terms of the evolution of the running time with respect to the number of partitions for neither of the two tested algorithms. Due to the significant skewness, the processing of the partitions containing very dense cells still accounts for the most predominant portion of the running time.

A more interesting trend for the two algorithms can be derived for the larger *OpenStreetMap* dataset. Indeed, the increase in the number of partitions seems to have a completely opposite effect on the two algorithms: while the new one shows some performance improvement, the additional fragmentation hampers RP-DBSCAN, whose running time increases almost linearly. For the new algorithm, a reduction of more than 20% (in the case of configuration #2, on which as before it performs slightly better than configuration #1) can be registered when the quantity of chunks is quadrupled (from 500 to 2000), while in the same range the running time for RP-DBSCAN more than doubles. Further splitting of the input data does not appear to provide any additional benefit to the new algorithm, at least for the tested configurations.

In the end, data partitioning appears to be a very effective technique to optimize the performance of the new algorithm, with speedups of up to five times with respect to RP-DBSCAN's using the same number of partitions.

Table 4.9. Running times (in seconds) for the *Geolife* dataset with variable number of partitions

RP-DBSCAN							
Partitions	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
6	43.41	39.75	47.75	44.68	44.50	44.018	2.880272557
12	45.85	49.91	51.69	57.40	43.66	49.702	5.352529309
18	47.67	40.85	50.59	42.14	41.34	44.518	4.359274022
24	52.94	42.36	54.88	49.32	50.45	49.990	4.782572530
30	43.41	45.35	41.46	44.33	48.30	44.570	2.529654127

New algorithm							
Partitions	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
6	41.85	38.99	38.55	41.50	39.10	39.998	1.549603175
12	41.28	44.53	41.33	42.19	39.34	41.734	1.879236547
18	35.92	35.39	39.23	38.80	40.93	38.054	2.337825913
24	41.84	38.79	40.47	34.63	39.00	38.946	2.708916758
30	44.52	39.85	38.41	42.16	42.40	41.468	2.378627756

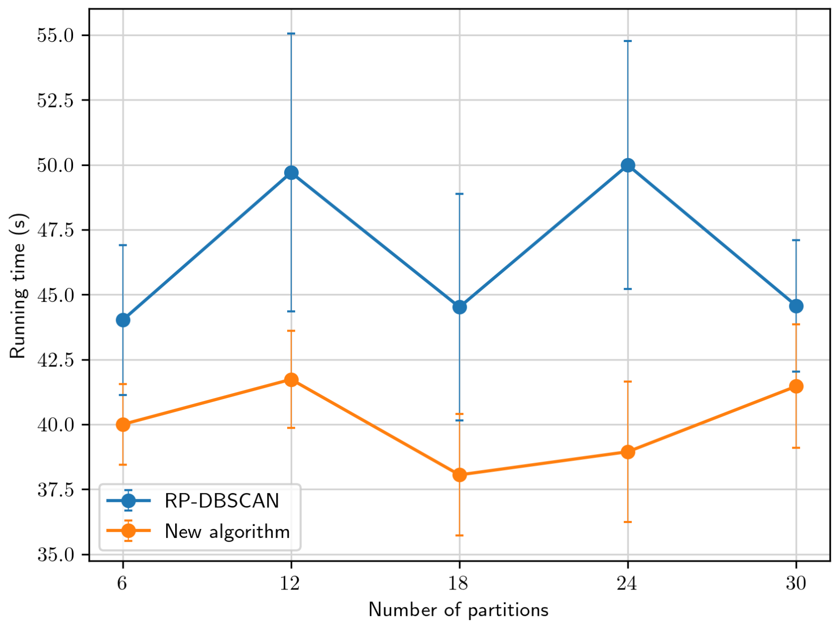

Figure 4.16. Performance on the *Geolife* dataset with variable number of partitions

Table 4.10. Running times (in seconds) for the *OpenStreetMap* dataset with variable number of partitions

RP-DBSCAN							
Partitions	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
500	1223	1189	1267	1203	1178	1212.0	35.04283094
1000	1943	1978	1822	1860	1955	1911.6	67.02462234
1500	2327	2281	2355	2375	2361	2339.8	37.21827508
2000	2884	2896	2857	2972	2885	2898.8	43.36703817

New algorithm (configuration #1)							
Partitions	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
500	770	771	772	796	731	768.0	23.35594143
1000	686	683	671	660	696	679.2	13.95349419
1500	641	616	654	617	645	634.6	17.18429516
2000	660	618	665	632	593	633.6	29.90484911

New algorithm (configuration #2)							
Partitions	Run 1	Run 2	Run 3	Run 4	Run 5	Mean	Stddev
500	711	724	764	704	731	726.8	23.33880888
1000	556	579	513	640	611	579.8	49.05812879
1500	593	517	601	605	612	585.6	38.95895276
2000	560	608	554	573	512	561.4	34.65256123

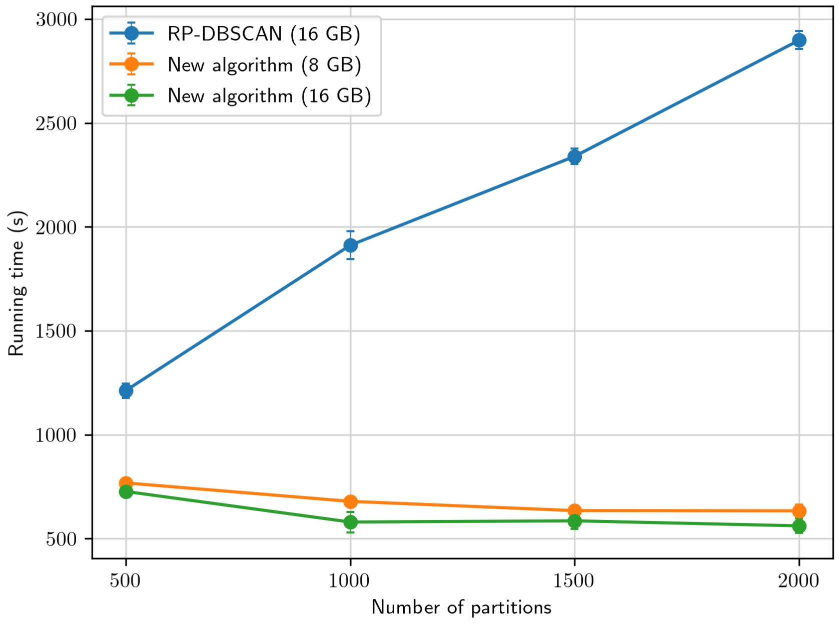

Figure 4.17. Performance on the *OpenStreetMap* dataset with variable number of partitions

4.4.10 RP-DBSCAN's approximation quality

Since the new algorithm allows to exactly retrieve outliers according to the DBSCAN definition, as a final consideration it is possible to make some superficial qualitative analysis of the results produced by a ρ-approximate algorithm such as RP-DBSCAN.

In Table 4.11, the outliers detected by RP-DBSCAN on the *Geolife* dataset for varying values of the parameter ϵ are split in three categories: actual outliers, false positives (i.e. data points marked as outliers) and false negatives (i.e. outliers marked as data points). As a general trend, it can be stated that ρ-approximate algorithms appear to have the tendency to identify a superset of the actual outliers as such. Indeed, in all cases except for the first, the algorithm detects all the DBSCAN outliers, plus a consistent proportion of false positives (around 15-20% of the output size). In the case with $\epsilon = 25$, a small number of false negative is also present.

Similar considerations may be given also for the larger *OpenStreetMap* dataset, whose results are reported in Table 4.12. Again, RP-DBSCAN shows a tendency to overestimate the set of outliers in the dataset by quite a margin of false positives; actual outliers are mined for the most part (a very reduced number of false negatives are present in all runs).

Table 4.11. RP-DBSCAN detection accuracy on the *Geolife* dataset

ϵ	New	RP-DBSCAN	Correct	False positives	False negatives
25	25652	30297	25632	4665	20
50	14829	17143	14829	2314	0
100	6750	8536	6750	1786	0
200	2498	3096	2498	598	0

Table 4.12. RP-DBSCAN detection accuracy for RP-DBSCAN on the *OpenStreetMap* dataset

ϵ	New	RP-DBSCAN	Correct	False positives	False negatives
250000	5343651	6594305	5343151	1251154	500
500000	2198398	2612656	2198224	414432	174
1000000	1084141	1225326	1083932	141394	209
2000000	506386	547805	505966	41839	420

Chapter 5

Conclusions

This work had the objective to introduce an algorithm to detect outliers in very large datasets, characterized by a strict bound on worst-time complexity, a good detection quality and a good scalability with respect to its parameters. In the end, all such goals may be considered reached. With respect to the other techniques, the new algorithm provides a very high result quality, with adjusted Rand scores of up to 0.95. Moreover, it scales better than other parallel (yet approximated) DBSCAN solutions (with speedups of up to 450% in some cases), while maintaining the accuracy of the results.

We believe that this algorithm could perform reasonably well in a wide range of application domains in which the concepts of distance among points and density are well-defined. One very relevant use case is represented, for example, by location data.

Upon the basic structure of the algorithm presented in this work, the applicability of several optimizations may surely be investigated in future ones. In particular, we acknowledge that the following points may be relevant branches of further development:

- The usage of some kind of cell-level spatial indices to improve the efficiency of range query operations;
- The implementation of a "clever" partitioning scheme that is able to achieve better load balancing among executors.

Bibliography

[1] Apache Foundation. Apache Spark - Unified Analytics Engine for Big Data. https://spark.apache.org/. [Online; accessed 03/01/2020].

[2] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. LOF: Identifying Density-Based Local Outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, page 93–104, New York, NY, USA, 2000. Association for Computing Machinery.

[3] I. Cordova and T. Moh. DBSCAN on Resilient Distributed Datasets. In *2015 International Conference on High Performance Computing Simulation (HPCS)*, pages 531–540, July 2015.

[4] Bi-Ru Dai and I-Chang Lin. Efficient Map/Reduce-Based DBSCAN Algorithm with Optimized Data Partition. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, page 59–66, USA, 2012. IEEE Computer Society.

[5] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[6] 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.

[7] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. CURE: An Efficient Clustering Algorithm for Large Databases. *SIGMOD Records*, 27(2):73–84, June 1998.

[8] Ade Gunawan. A Faster Algorithm for DBSCAN. Master's thesis, Technische Universiteit Eindhoven, March 2013.

[9] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. MR-DBSCAN: a scalable MapReduce-based DBSCAN algorithm for heavily skewed data. *Frontiers of Computer Science*, 8, February 2014.

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

[11] George Karypis, Eui-Hong (Sam) Han, and Vipin Kumar. Chameleon: Hierarchical Clustering Using Dynamic Modeling. *Computer*, 32(8):68–75, August 1999.

[12] Edwin M. Knorr and Raymond T. Ng. Algorithms for mining distance-based outliers in large datasets. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, page 392–403, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[13] Bing Liu. A fast density-based clustering algorithm for large databases. In *2006 International Conference on Machine Learning and Cybernetics*, pages 996–1000, August 2006.

[14] F. T. Liu, K. M. Ting, and Z. Zhou. Isolation Forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422, Dec 2008.

[15] Alessandro Lulli, Matteo Dell'Amico, Pietro Michiardi, and Laura Ricci. NG-DBSCAN: Scalable Density-Based Clustering for Arbitrary Data. *Proc. VLDB Endow.*, 10(3):157–168, November 2016.

[16] Shaaban Mahran and Khaled Mahar. Using grid for accelerating density-based clustering. In *2008 8th IEEE International Conference on Computer and Information Technology*, pages 35–40, July 2008.

[17] OpenStreetMap. Bulk GPS point data. https://blog.openstreetmap.org/2012/04/01/bulk-gps-point-data/. [Online; accessed 03/29/2020].

[18] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Andreas Müller, Joel Nothman, Gilles Louppe, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[19] Peter Rousseeuw and Katrien Driessen. A Fast Algorithm for the Minimum Covariance Determinant Estimator. *Technometrics*, 41:212–223, August 1999.

[20] Bernhard Schölkopf, Robert Williamson, Alex Smola, John Shawe-Taylor, and John Platt.

Support vector method for novelty detection. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, page 582–588, Cambridge, MA, USA, 1999. MIT Press.

[21] SmartData@Polito. Computing facilities. https://smartdata.polito.it/computing-facilities/. [Online; accessed 04/16/2020].

[22] Hwanjun Song and Jae-Gil Lee. RP-DBSCAN: A Superfast Parallel DBSCAN Algorithm Based on Random Partitioning. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1173–1187, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Always learning. Pearson, 2014.

[24] Cheng-Fa Tsai and Chien-Tsung Wu. GF-DBSCAN: A New Efficient and Effective Data Clustering Technique for Large Databases. In *Proceedings of the 9th WSEAS International Conference on Multimedia Systems and Signal Processing*, MUSP'09, page 231–236, Stevens Point, Wisconsin, USA, 2009. World Scientific and Engineering Academy and Society (WSEAS).

[25] Xiaowei Xu, Jochen Jäger, and Hans-Peter Kriegel. A Fast Parallel Clustering Algorithm for Large Spatial Databases. *Data Mining and Knowledge Discovery*, 3(3):263–290, September 1999.

[26] Yu Zheng, Hao Fu, Xing Xie, Wei-Ying Ma, and Quannan Li. *Geolife GPS trajectory dataset*, Geolife GPS trajectories 1.3 edition, August 2012.