NG-DBSCAN: a Scalable, Approximate Density-Based Clustering Algorithm for Arbitrary Similarity Metrics

Alessandro Lulli
University of Pisa
ISTI, CNR, Pisa, Italy

Matteo Dell'Amico
Symantec Research Labs

Pietro Michiardi
EURECOM, Campus
SophiaTech, France

Laura Ricci
University of Pisa
ISTI, CNR, Pisa, Italy

ABSTRACT

We present NG-DBSCAN, an approximate density-based clustering algorithm that can operate with arbitrary similarity metrics. The distributed design of our algorithm makes it scalable to very large datasets; its approximate nature makes it fast, yet capable of producing high quality clustering results. We provide a detailed overview of the various steps of NG-DBSCAN, together with their analysis. Our results, which we obtain through an extensive experimental campaign with real and synthetic data, substantiate our claims about NG-DBSCAN's performance and scalability.

1. INTRODUCTION

Clustering algorithms are fundamental in data analysis, providing an unsupervised way to aid understanding and interpreting data by grouping objects together, according to a user-provided notion of similarity. Within clustering algorithms, Ester et al. [8] introduced with DBSCAN the idea of *density-based* clustering: grouping data packed together in high-density regions of the feature space. DBSCAN is very well known and appreciated (it received the KDD test of time award in 2014) thanks to two very desirable features: first, it separates "core points" appearing in dense regions of the feature spaces from outliers ("noise points") which are classified as not belonging to any cluster; second, it recognizes clusters of arbitrary shape rather than "ball-shaped" ones, which are all similar to a given centroid.

While DBSCAN's outputs are very desirable, two limitations restrict its applicability to cases that are increasingly common: first, it is difficult to implement it on very large databases as its scalability is limited; second, existing implementations do not lend themselves well to heterogeneous data sets where similarity between items is best represented via arbitrarily complex functions. In this work, we target both problems, proposing an *approximated, scalable, distributed* DBSCAN implementation which is able to handle *arbitrary distance functions between items*, and can therefore handle items having *arbitrary data*, rather than being limited to points in an Euclidean space.

While the original DBSCAN paper claimed $\mathcal{O}(n \log n)$ running time (for data in d-dimensional Euclidean spaces), this has been recently proven wrong by Gan and Tao [9]: for $d \geqslant 3$, any correct DBSCAN implementation requires at least $\Omega\left(n^{4/3}\right)$ time. When the data size becomes large enough, this complexity becomes difficult to handle; for this very reason, Gan and Tao proposed an *approximated* single-machine DBSCAN implementation for Euclidean spaces with linear running time.

Several distributed DBSCAN implementations exist [5, 10, 12, 17]: they are based on the concept of *partitioning* the feature space, running a single-machine DBSCAN implementation on each partition, and then "stitching" the work done on the border of each partition. This approach is effective only when the dimensionality is low: when data has large dimensionality, the amount of work to connect what is done on each partition becomes unbearably large.

As we discuss in Section 2, the definition of DBSCAN itself simply requires a distance measure between items; a large majority of existing implementations, though, only consider data as points in a d-dimensional space, and only support Euclidean distance between them. This is the case for the distributed implementations referred to above, which base the partitioning on the requirement that pieces of data are points in an Euclidean space. This is inconvenient in the ever more common case of heterogeneous data sets, where data items fed to the clustering algorithm are made of one or more fields with arbitrary type: consider, for example, the case of textual data where edit distance is a desireable measure. Furthermore, the computational cost of the last "stitching" step grows quickly as the number d of dimensions increases, even if the intrinsic data dimensionality remains low.

The typical way of coping with such limitations is extracting an array of numeric features from the original data: for example, textual data is converted to a vector via the `word2vec` [4] algorithm. Then, Euclidean distance between these vectors is used as a surrogate for the desired distance function between data. Our proposal, NG-DBSCAN (described in Section 3), gives instead the full flexibility of specifying an *arbitrary* distance function. Recognizing from the contribution of Gan and Tao that computing DBSCAN exactly imposes limits to scalability, our approach computes instead an *approximation* to the exact DBSCAN clustering. Rather than partitioning an Euclidean space – which is impossible with arbitrary data and has problems with high dimensionality, as discussed before – our algorithm is based on a vertex-centric design,

whereby we compute a *neighbor graph*, a distributed data structure describing the "neighborhood" of each piece of data (i.e., a set containing its most similar items). We compute the clusters based on the content of the neighbor graph, whose acronym gives the name to NG-DBSCAN.

NG-DBSCAN is implemented in Spark, and it is suitable to be ported to frameworks that enable distributed vertex-centric computation; in our experimental evaluation we evaluate both the scalability of the algorithm and the quality of the results, i.e., how close these results are to the results of an exact computation of DBSCAN. We compare NG-DBSCAN with competing DBSCAN implementations, on real and synthetic datasets. All details on the experimental setup are discussed in Section 4.

Our results, reported in Section 5, show that NG-DBSCAN outperforms competing DBSCAN implementations, while the approximation imposes small or negligible impact on the results. Furthermore, we investigate the case of clustering text based on a `word2vec` transformation: we show that – if one is indeed interested in clustering text based on edit-distance similarity – the penalty in terms of clustering quality is substancial, unlike what happens with the approach enabled by NG-DBSCAN.

We consider that this line of research opens the door to several interesting and important contributions. With the concluding remarks of Section 6, we outline our future research lines, including an adaptation of this approach to streaming data, and supporting regression and classification using similar approaches.

Summary of Contributions. We propose NG-DBSCAN, an approximated and distributed implementation of DBSCAN.

NG-DBSCAN's main merits are:

- **Efficiency.** NG-DBSCAN outperforms other existing distributed implementations of DBSCAN, with the approximation having a small to negligible impact on the results.
- **Versatility.** NG-DBSCAN's vertex-centric approach enables distribution without needing an Euclidean space to partition. It therefore allows domain experts to represent similarity between items through *arbitrary* distance measures, allowing them to tailor their definition according to domain-specific knowledge.

Our experimental evaluation supports these claims through an extensive comparison between NG-DBSCAN and alternative implementations, on a variety of real and synthetic datasets.

2. BACKGROUND AND RELATED WORK

In this Section we first revisit the DBSCAN algorithm, then we discuss existing distributed implementations of density-based clustering. We conclude with an overview of *ad-hoc* techniques for clustering text and/or high-dimensional data.

2.1 The DBSCAN Algorithm

DBSCAN is defined by Ester et al. as a sequential algorithm [8]. Data points are clustered by density, which is defined through two parameters: ε and MinPts. The *ε-neighborhood* of a point p is the set of points within distance ε from p.

Core points are those with at least MinPts points in their ε-neighborhood. Other points are either *border* or *noise* points: border points have at least one core point in their ε-neighborhood, whereas noise points do not. Noise points are assigned to no cluster.

A cluster is formed by the set of *density-reachable* points from a given core point c: those in c's ε-neighborhood and, recursively, those that are density-reachable from core points in c's ε-neighborhood. DBSCAN identifies clusters by iteratively picking unlabeled core points and identifying their clusters by exploring density-reachable points, until all core points are labeled. Note that DBSCAN clustering results can vary slightly if the order in which clusters are explored changes, since border points with several core points in their ε-neighborhood may be assigned to different clusters.

For 17 years, the time complexity of DBSCAN has been believed to be $O(n \log n)$. Recently, Gan and Tao [9] discovered that the complexity is in fact $\Omega\left(n^{4/3}\right)$ – which explains why existing implementations only evaluated DBSCAN for rather limited numbers of points – and proposed an approximate algorithm, ρ-DBSCAN, running in $O(n)$ time. Unfortunately, the data structure at the core of the algorithm does not permit to handle arbitrary similarity measures, and only Euclidean distance is used in both the description and experimental evaluation.

We remark that the definition of DBSCAN revolves on the ability of finding the ε-neighborhood of each data point: as long as a distance measure is given, the ε-neighborhood of a point p is well-defined no matter what the type of p is. NG-DBSCAN does not impose any limitation on the type of data points nor on the properties of the distance function.

2.2 Distributed Density-Based Clustering

MR-DBSCAN [10] is the first proposal targeting a parallel implementation of DBSCAN realized as a 4-stage MapReduce algorithm: partitioning, clustering, and two stages devoted to merging. This approach concentrates on defining a clever partitioning of data in a d-dimensional Euclidean space, where each partition is assigned to a worker node. A modified version of PDBSCAN [21], a popular parallel DBSCAN implementation, is executed on the subspace of each partition. Nodes within distance ε from a partition's border are replicated and one of the stages is in charge of merging clusters between different partitions. Unfortunately, the evaluation of MR-DBSCAN does not compare it to other DBSCAN implementations, and only considers points in a 2D space.

DBSCAN-MR [5] is a similar approach which again implements DBSCAN as a 4-stage MapReduce algorithm. Key differences are the use of a k-d tree for the single-machine implementation, and a partitioning algorithm that recursively divides data in slices to minimize the number of boundary points and to balance the computation.

MR. SCAN [20] is another proposal adopting a similar 4-stage implementation, this time exploiting GPGPU acceleration for the local clustering stage. Authors only implemented a 2D version, but claim it is feasible to extend the approach to any d-dimensional Euclidean space.

PARDICLE [17] is a proposal for an approximated algorithm, focused on density estimation rather than exact ε-neighborhood queries. PARDICLE is implemented with MPI, and adjusts the precision of the estimation according to how close the density of a given area is with respect to the threshold separating areas having core and non-core points.

DBCURE-MR [12] is a density-based MapReduce algorithm which is not equivalent to DBSCAN: rather than circular ε-neighborhoods, it is based on ellipsoidal τ-neighborhoods. DBCURE-MR is again implemented as a 4-stage MapReduce algorithm.

Table 1 summarizes current parallel implementations of density-based clustering algorithms, together with their execution environment, and their features. All these approaches have in common the fact that the algorithm is distributed by partitioning a d-dimensional space, and only supports Euclidean distance. Our approach to parallelization does not involve data partitioning, and is instead based

Table 1: Overview of parallel density-based clustering algorithms.

Name	Parallel model	DBSCAN Computation	Approximated	Partitioner required	Data object type	Distance function supported
ρ-DBSCAN	single machine	yes	yes	data on a grid	point in n-D	Euclidean
MR-DBSCAN	MapReduce	yes	no	yes	point in 2-D	Euclidean
DBSCAN-MR	MapReduce	yes	no	yes	point in 2-D	Euclidean
MR. SCAN	MRNet + GPGPU	yes	no	yes	point in n-D	Euclidean
PARDICLE	MPI	yes	yes	yes	point in n-D	Euclidean
DBCURE-MR	MapReduce	no	no	yes	point in n-D	Euclidean
NG-DBSCAN	**MapReduce**	yes	yes	**no**	**arbitrary type**	**arbitrary distance**

on a vertex-centric design, which ultimately is the key to support arbitrary data and similarity measures between points, and to avoid scalability problems due to high-dimensional data.

2.3 Density-Based Clustering for High-Dimensional Data

We conclude our discussion of related work with density-based clustering approaches suitable for text and high-dimensional data in general.

Tran et al. [19] propose a method to identify clusters with different densities. Instead of defining a threshold for a local density function, low-density regions separating two clusters can be detected by calculating the number of shared neighbors. If the number of shared neighbors is below a threshold, then the two objects belong to two different clusters. Tran et al. report that their approach has high computational complexity, and the algorithm was evaluated using only a small dataset (below 1 000 objects). In addition, as the authors point out, this approach is unsuited for finding clusters that are very elongated or have particular shapes.

Zhou et al. [22] define a different way to identify dense regions. For each object p, their algorithm computes the ratio between the size of p's ε-neighborhood and those of its neighbors, to distinguish nodes that are at the center of clusters. Unfortunately, this approach is once again only evaluated and compared with DBSCAN in a 2D Euclidean space.

3. NG-DBSCAN: APPROXIMATE AND GENERAL DBSCAN

NG-DBSCAN is an approximate, distributed, scalable algorithm for density-based clustering, supporting arbitrary similarity metrics. The algorithm proceeds in two phases. We call the output of the first phase *ε-graph*: its nodes are data points, and each node's neighbors are subset of its ε-neighborhood. The second phase performs the actual clustering, gathering data from the ε-graph rather than performing exact and expensive ε-neighborhood queries. The final output assigns nodes to clusters. The efficiency of NG-DBSCAN and its approximation are both due to the fact that the ε-graph does not encode the full ε-neighborhood of each node.

We adopt the *vertex-centric*, or "think like a vertex" programming paradigm, in which computation is partitioned by and logically performed at the vertexes of a graph, and vertexes exchange messages. The vertex-centric approach is widely used due to its scalability and expressivity [16].

In the following, we describe the details of NG-DBSCAN. For clarity of exposition, we gloss over the technicalities of its distributed implementation, focusing instead on the principles of the underlying algorithm. NG-DBSCAN is implemented using the Apache Spark framework, and we have used features such as caching for efficiency. The source code of NG-DBSCAN is available for review online.[1]

3.1 Phase 1: Building the ε-Graph

The goal of the first phase is to build the ε-graph: in Phase 2, this graph is used to differentiate between core and non-core nodes and to compute density-reachability. As discussed before, a node's neighbors in the ε-graph are a subset of its ε-neighborhood. Hence, NG-DBSCAN is efficient and approximated because a nodes' neighbors in the ε-graph are not necessarily its full ε-neighborhood. Doing that would guarantee an exact computation of DBSCAN, but it would require a much larger computational cost ($\mathcal{O}(n^2)$ for generic distance functions).

In Phase 2 we will look up a node's neighbors in the ε-graph rather than performing exact ε-neighborhood queries. We are therefore interested in computing efficiently an ε-graph with enough edges to *i)* allow distinguishing core nodes from non-core ones; *ii)* preserving density-reachability between core nodes, such that clusters are not split. In our experimental evaluation we have seen that density-reachability is generally not an issue once enough information to distinguish core and non-core points is available in the ε-graph.

3.1.1 The Algorithm

To build the ε-graph, we use an auxiliary structure called *neighbor graph*, which is a directed graph having data items as nodes and distances between them as edge labels. The neighbor graph is initialised by connecting each node to k random other nodes, where k is an NG-DBSCAN parameter. At each iteration of the algorithm we consider pairs of nodes (x, y) separated by 2 hops in the neighbor graph: if the distance between them is smaller than the k^{th} smallest label on an outgoing edge e from either node, then edge e is discarded from the neighbor graph and replaced by (x, y). Through this algorithm, as soon as we discover a pair of nodes at distance ε or less, we add the corresponding edge to the ε-graph.

The neighbor graph and its evolution have been inspired by the approach that Dong at al. [7] used to compute approximated k-nearest neighbor (k-NN) graphs. Indeed, by letting our algorithm run indefinitely, the neighbor graph converges to an approximation of a k-NN graph: in our case, rather than being interested in finding the k nearest neighbors of an item, we want to be able to distinguish whether that item is a core point. Hence, as soon as a node has M_{max} neighbors in the ε-graph, where M_{max} is an NG-DBSCAN parameter, we consider that we have enough information about that node and we remove it from the neighbor graph to speed up the computation. In summary, in addition to the parameters of DBSCAN (ε and MinPts), we introduce two parameters: k and M_{max}. The effect of these parameters is extensively studied

[1] `https://github.com/alessandrolulli/gdbscan`

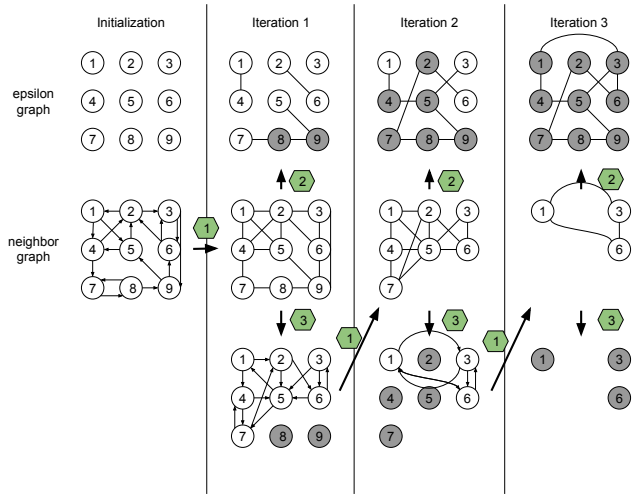


Figure 1: ε-NN construction phase.

in Sections, respectively, 5.1.2 and 5.1.3: we have found experimentally that choices of $k = 10$ and $M_{max} = \max(MinPts, 2k)$ provide consistently good results.

Since NG-DBSCAN accepts arbitrary distance functions, computing some of them can be very expensive: a solution for this is memoization (i.e., caching results to avoid computing the distance function between the same elements several times). Writing a solution to perform memoization is almost trivial in a high-level language such as Scala,[2] but various design choices – such as choice of data structure for the cache and/or eviction policy – are available, and choosing appropriate ones depends on the particular function to be evaluated. We therefore consider memoization as an orthogonal problem, and rely on users to provide a distance function which performs memoization if it is useful or necessary.

3.1.2 Illustrative example

Figure 1 illustrates Phase 1 with a running example; in this case, for simplicity, $k = M_{max} = 2$. The algorithm is initialised by creating an ε-graph with no edges and a neighbor graph with $k = 2$ outgoing edges per nodes chosen at random.

Each iteration proceeds through three steps, indicated in Figure 1 with hexagons labeled 1, 2, and 3. In step 1, the directed neighbor graph is transformed in an undirected one. Then, through the transition labeled 2, edges are added to the ε-graph if their distance is $\leq\varepsilon$. For instance, edge $(2, 6)$ is added to the ε-graph in the first iteration. Finally, in step 3 each node explores its two-hop neighborhood and builds a new neighbor graph while keeping connections to the k closest nodes. Nodes with at least M_{max} neighbors in the ε-graph are deactivated (marked in grey) and will not appear in the neighbor graph in the following iteration.

3.1.3 Implementation Details

To limit the number of message exchanges, we adopt two techniques. The first is an "altruistic" mechanism to compute neighborhoods: each node computes distances between all its neighbors in the neighbor graph, and sends them the k nodes with the smallest distance. With this optimization, it is not necessary to collect, at each node, information about each of their neighbors-of-neighbors. The second technique prevents a node with many neighbors to send an excessive number of messages. We introduce a parameter ρ to avoid bad performance in degenerate cases, bounding the number of messages to ρk.

[2]See, e.g., `http://stackoverflow.com/a/16257628`.

Algorithm 1: ε-graph construction.

```
1  ε-NN = ∅
2  removedNodes = ∅
3  neighbor = RandomInitialization()
4  nodeNumber = count(neighbor)
5  while i < iter ∧ ¬(T_n > nodeNumber ∧ T_r > nodeRemoved) do
6      H = {ReverseMap(n) ∀n ∈ neighbor}
7      T = {CheckNeighborhood(n, removedNodes) ∀n ∈ H}
8      ε-NN = {Reduceεnn(n, l, removedNodes) ∀(n, l) ∈T}
9      nodeNumber = count(neighbor)
10     neighbor = {ReduceNeighbor(n, l) ∀(n, l) ∈T}
11     i = i + 1
12 end
```

NG-DBSCAN is implemented in Spark, and this framework is sometimes known to use large quantities of memory. To avoid memory issues, we have implemented an option to divide a single logical iteration into multiple ones. Specifically, an optional parameter $\mathcal{S}$ allows splitting each iteration in $t = \lceil \hat{n}/\mathcal{S} \rceil$ Spark sub-iterations, where $\hat{n}$ is the number of nodes currently in the neighbor graph. When this option is set, at most $\mathcal{S}$ nodes use the altruistic approach to explore distances between neighbors in each sub-iteration. Each node is activated exactly once within the t sub-iterations, therefore this option has no impact of the final results which are equivalent to the ones with a single iteration.

3.1.4 Termination Condition

Deciding when to stop algorithm execution is a challenge that we tackle next. In addition to setting a maximum number of iterations (see the variable *iter* in Line 5 in Algorithm 1), which ensures termination in degenerate cases of datasets with a majority of noise points, phase 1 terminates according to two threshold parameters: $\mathcal{T}_n$ and $\mathcal{T}_r$.

Intuitively, the idea is as follows. Our algorithm proceeds by examining, in each iteration, only active nodes in the neighbor graph: the number of active nodes $a(t)$ decreases as the algorithm runs. Hence, it would be tempting to wait for T^* iterations, such that $a(T^*) = 0$. However, the careful reader will recall that noise points cannot be deactivated: as such, a sensible alternative is to set the stop condition to T^* such that $a(T^*) < \mathcal{T}_n$.

Now, the above inequality alone is somehow difficult to tune: too small values of $\mathcal{T}_n$ might result in the algorithm stopping too early, ultimately leading to poor clustering quality. To overcome such problem, we introduce an additional threshold, that operates on the differential number of nodes that have been deactivated between two subsequent iterations: formally, let $i(t)$ be the number of nodes deactivated in iteration t; then we complement our stop conditions by finding T^* such that $\frac{di(T^*)}{dt} < \mathcal{T}_r$.

With forward reference to Figure 3 on page 8, we see quantities $a(t)$, $i(t)$ and $\frac{di(t)}{dt}$. Neither of the two above conditions alone would achieve a good stopping criteria: using either $\mathcal{T}_n$ or $\mathcal{T}_r$ would stop the algorithm too early. Indeed, $\frac{di(T^*)}{dt} < \mathcal{T}_r$ can be satisfied either at the early stage of the algorithm or toward its convergence. Inequality $a(T^*) < \mathcal{T}_n$ imposes the algorithm to progress past the few first iterations. Then, towards convergence, the $\mathcal{T}_r$ inequality allows the algorithm to continue past the $\mathcal{T}_n$ threshold, but avoids running the algorithm for too long.

We have found empirically (see Section 5.1.1) that setting $\mathcal{T}_n = 0.7n$ and $\mathcal{T}_r = 0.01n$ yields fast convergence while keeping the results similar to those of an exact DBSCAN computation.

Algorithm 2: ε-graph construction: procedures.

```
1  procedure ReverseMap(Node n)
2      forall the u ∈Neighborhood(n) do
3          EMIT(n, u)
4          if computingNode(u) then
5              EMIT(u, n)
6          end
7      end
8  procedure CheckNeighborhood(Node n, removedNodes)
9      if computingNode(n) then
10         forall the u ∈ Neighborhood(n).LIMIT(ρk)∪{n} do
11             if ¬removedNodes.contains(u) then
12                 l = ∅
13                 forall the v ∈ Neighborhood(n) ∪ {n} \ {u}
                   do
14                     l = l ∪ ((v, DISTANCE(u, v)))
15                 end
16                 EMIT(u, l)
17             end
18         end
19     else
20         if ¬removedNodes.contains(n) then
21             EMIT(n, Neighborhood(n))
22         end
23     end
24 procedure ReduceεNN(Node n, List⟨(Node, Distance)⟩ l,
   removedNodes)
25     forall the (v, d) ∈ l do
26         if d ⩽ ε then
27             NeighborεNN(n) = NeighborεNN(n) ∪ {v}
28         end
29     end
30     if |NeighborεNN(n)| ⩾ M_max then
31         removedNodes = removedNodes ∪ {n}
32     end
33     EMIT(n, NeighborεNN(n))
34 procedure ReduceNeighbor(Node n, List⟨(Node, Distance)⟩ l)
35     orderedList = ORDERDESC(l).LIMIT(k)
36     EMIT(n, orderedList)
```

3.1.5 Phase 1 in Detail

Algorithm 1 shows the pseudo-code of the ε-graph construction phase.[3] Each iteration of phase 1 involves the execution of the following procedures: `ReverseMap`, `CheckNeighborhood`, `ReduceNeighbor` and `ReduceεNN`.[4]

`ReverseMap` (Algorithm 2, line 1) corresponds to step 1 of Figure 1. We convert the neighbor graph to an undirected graph, avoiding nodes deactivated in the current iteration due to the $\mathcal{S}$ parameter described in Section 3.1.3 (line 4).

`CheckNeighborhood` (line 8) performs the "altruistic" neighborhood exploration described in Section 3.1.3, where each active node computes the distances between each neighbor. Nodes deactivated in the current iteration due to the $\mathcal{S}$ parameter propagate their current neighborhood witouth updating it (line 21). Also, each node sends messages only for a subset of its neighbors to limit the number of messages sent thanks to the parameter ρ (line 10).

`ReduceεNN` corresponds to step 2 of Figure 1, and updates the ε-graph with the newly discovered edges. If a node ends up with more than M_{max} neighbors, it is removed from the neighbor graph (line 31).

Finally, `ReduceNeighbor` corresponds to step 3 of Figure 1, and builds a new directed neighbor graph with each nodes' k closest discovered neighbors.

3.1.6 Complexity Analysis

Unless the conditions for early termination are met, Phase 1 runs for a user-specified number of iterations. Since the number of nodes in the neighbor graph decreases with time, the first iterations are the most expensive (i.e., when a node is removed from the neighbor graph is never added again). In this work, we study the complexity of the first iteration, which has the highest cost since all nodes are present in the neighbor graph. In case logical iterations and Spark sub-iterations exists because the $\mathcal{S}$ parameter is set (see Section 3.1.3), here we refer to the cost of a logical iteration, in terms of number of times the distance function is called and of number of messages.

The `ReverseMap` procedure generates $2m$ messages, where $m = kn$ is the number of edges in the neighbor graph. The number of messages sent while executing this procedure are hence $\mathcal{O}(kn)$.

The `CheckNeighborhood` procedure (Algorithm 2, line 8) computes distances between at most ρk neighbors of each node, where the neighbor list has a size $2kn$ as described above, and each node has at least k neighbors. The worst case is when neighbor lists are distributed as unevenly as possible, that is when $n/(\rho-1)$ nodes have ρk neighbors, and all the others only have k. In that case, we would have $\mathcal{O}(n/\rho)$ nodes computing $\mathcal{O}\left(\rho^2 k^2\right)$ comparisons, and $\mathcal{O}(n)$ nodes computing $\mathcal{O}(k^2)$ comparisons. The results is

$$\mathcal{O}\left(\frac{n}{\rho}\rho^2 k^2 + nk^2\right) = \mathcal{O}(\rho n k^2).$$

Each comparison generates a message, so the number of emitted messages in this procedure is also $\mathcal{O}(\rho n k^2)$.

At the end of an iteration there are two phases, updating respectively the ε-graph and the neighbor graph. In each of these phases, a message is triggered by another message sent by `CheckNeighborhood`, and therefore the message complexity is again $\mathcal{O}(\rho n k^2)$.

In conclusion, the total message and computational complexity for an iteration of Phase 1 is $\mathcal{O}(\rho n k^2)$. Note that, in general, ρ and k should take small values (the default values we suggest in Section 5.1 are $\rho = 3$ and $k = 10$), therefore the computation cost is dominated by n.

3.2 Phase 2: Discovering Dense Regions

The ε-graph computed in Phase 1 is now used to compute the clusters which are the final output of our algorithm. The (expensive) ε-neighborhood queries of DBSCAN are replaced by neighborhood lookups in the ε-graph.

In its original description, DBSCAN is a sequential algorithm. We base our parallel implementation on the realization that a set of density-reachable core nodes corresponds to a connected component in the ε-neighborhood graph – the graph where each core node is connected to all core nodes in its ε-neighborhood. As such, our Phase 2 implementation uses the ε-graph as an approximation of the ε-neighborhood graph, and builds on a distributed algorithm to compute connected components. Differently from simply finding connected components, our algorithm must distinguish between core nodes (which are a proxy for generating clusters), noise points (that do not participate to this phase) and border nodes (which must

[3] We gloss over the pseudo code of auxiliary functions, such as `Neighborhood`, `Distance` and `Limit`, which can be easily understood from their method name.

[4] The naming convention of such operation is reminiscent of the MapReduce programming model that we use in this work, supported by the Apache Spark framework. Each phase of the computation relies on an implicit synchronization barrier, that determines message exchange between the machines executing the algorithms on partitioned data.

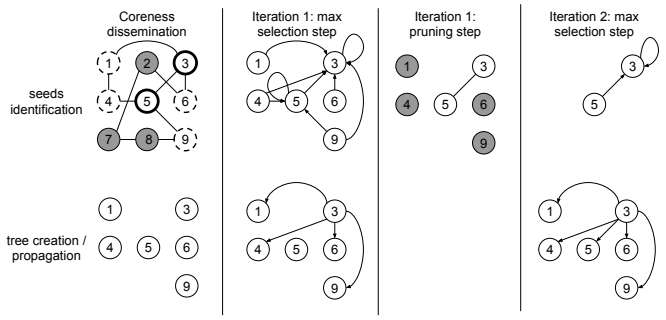


Figure 2: Discovery dense regions phase.

be treated with care, as they do not generate clusters). Our algorithm is inspired by Cracker [14], an efficient, distributed method to find connected components in a graph.

Algorithm 3: Discovering dense regions.

Input : an undirected ε-NN graph
Output: a graph where every node is labelled with the seed of its dense region

1 u.Active = $True\ \forall n$
2 $PropagationTree = (n, \emptyset)$
3 $t = 1$
4 $G =$ `Coreness_Dissemination(`ε-NN`)`
5 **while** $G = \emptyset$ **do**
6 $\quad H = \{$ `Max_Selection_Step(`n`)` $\forall n \in G\}$
7 $\quad G = \{$ `Pruning_Step(`$n, PropagationTree$`)` $\forall n \in H\}$
8 **end**
9 $G^* =$ `Seed_Propagation(`T`)`
10 **return** G^*

3.2.1 The Algorithm

We attribute node roles based on their properties in the ε-graph: nodes with at least $MinPts-1$ neighbors are considered core;[5] between non-core nodes, those with core nodes as neighbors are considered border nodes, while others will be treated as noise. Noise nodes are immediately deactivated, and they will not contribute to the computation anymore.

Like several other algorithms for graph connectivity, our algorithm requires a total ordering between nodes, such that each cluster will be labeled with the smallest or largest node according to this ordering. A typical choice is an arbitrary node identifier; for performance reasons that we discuss in the following, we use the node with the largest degree instead and resort to the node identifier to break ties in favor of the smaller ID. In the following, we will refer to the $(\text{degree}, \text{nodeID})$ pair as *coreness*; as a result of the algorithm, each cluster will be tagged with the ID of the highest coreness node in its cluster. We will call *seed* of a cluster the node with the highest coreness.

Phase 2 is illustrated in Algorithm 3; the algorithm proceeds in three steps: after an initalization step called *coreness dissemination*, an iterative step called *seed identification* is performed until convergence. Clusters are finally built in the *seed propagation* step. We describe them in the following, with the help of the running example in Figure 2.

[5]The $MinPts - 1$ value stems from the fact that, in the original DBSCAN implementation, a node itself counts when evaluating the cardinality of its ε-neighborhood.

Algorithm 4: `Max_Selection_Step` (u).

Input : a node $u \in G$
1 $NN_G(u) = \{v : (u \leftrightarrow v) \in G\}$
2 v_{max} = `maxCoreNode`$(NN_G(u) \cup \{u\})$
3 **if** u *is not-core* **then**
4 $\quad$ `EMIT` (u, v_{max})
5 $\quad$ `EMIT` (v_{max}, v_{max})
6 **else**
7 $\quad$ **forall the** $v \in NN_G(u) \cup \{u\}$ **do**
8 $\quad\quad$ `EMIT` (v, v_{max})
9 $\quad$ **end**
10 **end**

Algorithm 5: `Pruning_Step`(u, T).

Input : a node $u \in H$ and the propagation tree T
1 $NN_H(u) = \{v : (u \to v) \in H\}$
2 v_{max} = `maxCoreNode`$(NN_H(u))$
3 **if** u *is not-core* **then**
4 $\quad u$.Active = $False$
5 $\quad$ `EMITTREE` (v_{max}, u)
6 **else**
7 $\quad$ **if** $|NN_H(u)| > 1$ **then**
8 $\quad\quad$ **forall the** $v \in NN(u) \setminus v_{max}$ **do**
9 $\quad\quad\quad$ `EMIT` (v, v_{max})
10 $\quad\quad\quad$ `EMIT` (v_{max}, v)
11 $\quad\quad$ **end**
12 $\quad$ **end**
13 $\quad$ **if** $u \notin NN_H(u)$ **then**
14 $\quad\quad u$.Active = $False$
15 $\quad\quad$ `EMITTREE` (v_{max}, u)
16 $\quad$ **end**
17 $\quad$ **if** `IsSeed` (u) **then**
18 $\quad\quad u$.Active = $False$
19 $\quad$ **end**
20 **end**

Coreness dissemination. In this step, each node sends a message with its coreness value to its neighbors in the ε-graph. For example. in Figure 2, nodes 3 and 5 have the highest coreness; 1, 4, 6 and 9 are border nodes, and the others are noise. We omit the pseudocode for brevity.

Note that, although the following step modify the graph structure, coreness values are *immutable*.

Seed Identification. This step finds the seeds of all clusters, and builds a set of trees that we call *propagation forest* that ultimately link each core and border node to their seed. This step proceeds by alternating two sub-steps until convergence: *Max Selection Step* and *Pruning Step*. The ε-graph is iteratively simplified, until only seed nodes remain in it; at the end of this step, information to reconstruct clusters is encoded in the propagation forest.

With reference to Algorithm 4, in the Max Selection Step each node identifies the current neighbor with maximum coreness as its proposed seed (Line 2); each node will create a link between each of its neighbors – plus themselves – and the seed it proposes. Border nodes have a special behavior (Line 3)): they only propose a seed for themselves and their own proposed seed rather than for their whole neighborhood (Line 8)). In the first iteration of Figure 2, for example, node 4 – which is a border node – is responsible for creating edges $(4, 5)$ and $(5, 5)$. On the other hand, node 5 – which is a core node – identifies 3 as a proposed seed, and creates edges $(4, 3)$, $(5, 3)$, and $(3, 3)$.

In the Pruning Step presented in Algorithm 5, nodes not proposed as seeds (i.e., those with no incoming edges) are deactivated

(Line 14). An edge between deactivated nodes and their outgoing edge with highest coreness is created (Line 15). For example, in the first iteration of the algorithm, node 4 is deactivated and the $(4, 3)$ edge is created in the propagation forest.

Eventually, the seeds remain the only active nodes in the computation. Upon their deactivation, seed identification terminates and seed propagation is triggered.

Seed Propagation. The output of the seed identification step is the propagation forest: a dyrected acyclic graph where each node with zero out-degree is the seed of a cluster, and the root of a tree covering all nodes in the cluster. Clusters are generated by exploring these trees; the pseudocode of this phase is omitted for brevity.

3.2.2 Discussion

Phase 2 of NG-DBSCAN is implemented on the "blueprint" of the Cracker algorithm, to retain its main advantages: a node-centric design, which nicely blends with phase 1 of NG-DBSCAN, and a technique to speed up convergence by deactivating nodes that cannot be seeds of a cluster, which also contributes to decrease the message complexity of the algorithm. For these reasons, the complexity analysis of Phase 2 follows the same lines of that of Cracker, albeit the two algorithms substantially differ in their output: we defer the interested reader to [14] for analysis of the Cracker algorithm. In [14] it has been shown empirically that Cracker requires a number of messages and a number of iterations that respectively scale as O(log(m)) messages and O(log(n)) iterations, where n is the number of nodes and m the number of edges.

Note also that the choice of total ordering between nodes does not have an effect on the final results of our algorithm. However, the time needed to reach convergence depends on the number of iterations – which is equal to the height of the tallest tree in the propagation forest. Our choice of coreness, driven by degree, is a heuristic that performs very well in this respect, since links in the propagation forest point towards the densest areas in the clusters, resulting in trees that are wide rather than tall.

4. EXPERIMENTAL SETUP

We evaluate NG-DBSCAN thorugh a comprehensive set of experiments, evaluating well-known measures of clustering quality on real and synthetic datasets, and comparing it to alternative approaches. In the following, we provide details about our experimental setup.

4.1 Experimental Platform

All the experiments have been conducted on a cluster running Ubuntu Linux consisting of 17 nodes (1 master and 16 slaves), each equipped with 12 GB of RAM, a 4-core CPU and a 1 Gbit interconnect. Both the implementation of our approach and the alternative algorithms we use for our comparative analysis use Apache Spark [1] API. Our source code is publicly available.[6]

4.2 Evaluation Metrics

We now discuss the metrics we use to analyse the performance of our approach and the most important parameters of NG-DBSCAN. We also proceed with manual investigation of the clusters we obtain on some dataset, using domain knowledge to evaluate their quality.

We study the role of the parameters of our approach and the clustering quality using well-known measures of quality [6, 13]:

- **Compactness**: measures how closely related the items in a cluster are. We obtain the compactness by computing the average pairwise similarity among items in each cluster. Higher values are preferred.
- **Separation**: measures how well clusters are separate from each other. Separation is obtained by computing the average similarity between items in different clusters. Lower values are preferred.
- **Recall**: this metric relates two different data clusterings. Using clustering C as a reference, all node pairs that belong to the same cluster in C are generated. The recall of clustering D is the fraction of those pairs that are in the same cluster in D as well. In particular, we use as a reference the exact clustering we obtain with the standard DBSCAN implementation of the SciKit library. Higher values of recall are preferred.

Note that computing the above metrics is computationally as hard as computing the clustering we intend to evaluate. For this reason, we resort to uniform sampling: instead of computing the all-to-all pairwise similarity between items, we pick items uniformly at random, with a sampling rate of 1%.[7]

Additionally, we also consider algorithm **Speed-Up**: this metric measures the algorithm runtime improvement when increasing the number of cores dedicated to the computation, using 4 cores (a single machine) as the baseline.

4.3 The Datasets

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

- *Twitter Dataset*. We collected 5 602 349 geotagged tweets sent in USA the week between 2012/02/15 and 2012/02/21. Each tweet is in JSON format. This dataset is used to evaluate NG-DBSCAN in two distinct cases: (i) using the latitude and longitude values to cluster tweets using the Euclidean distance metric, (ii) using the text field to cluster tweets according to the Jaro-Winkler metric [11].
- *Spam Dataset*. A subset of SPAM emails collected by Symantec Research Labs, between 2010-10-01 and 2012-01-02, which is composed by 3 886 371 email samples. Each item of the dataset is formatted in JSON and contains the common features of an email, such as: subject, sending date, geographical information, the bot-net used for the SPAM campaign as labeled by Symantec systems, and many more. For instance, a subject of an email in the dataset is "19.12.2011 Rolex For You -85%" and the sending day is "2011-12-19".

In addition we also use synthetically generated input data using the SciKit library [18]. We generated three different types of input data called, respectively, circle, moon and blobs. These graph are usually considered as a baseline for testing clustering algorithms in a d-dimensional space.

4.4 Alternative Approaches

We compare NG-DBSCAN to existing algorithms that produce data clustering. Namely, we use the following alternatives:

- DBSCAN: this approach uses the SciKit library DBSCAN implementation. Clustering results obtained with this method can be thought of as our baseline, to which we compare NG-DBSCAN, in terms of clustering recall.

[6] https://github.com/alessandrolulli/gdbscan

[7] We increase the sampling rate up to 10% for clusters with less than 10 000 elements.

- SPARK-DBSCAN: this approach uses a parallel DBSCAN implementation for Apache Spark.[8] This work is an implementation of the algorithm called MR-DBSCAN in the literature, see details in Section 2. We treat this method as our direct competitor, and compare the runtime performance and clustering quality.

- k-MEANS: we convert text to vectors using `word2vec` [4], and cluster those vectors using the k-MEANS implementation in Spark's MLLib library [2]. We consider this approach, as described in [3], as a baseline for clustering quality; we evaluate it as an alternative to NG-DBSCAN for text data.

5. RESULTS

Through our experiments, we first study the role of NG-DBSCAN's parameters. Then, we evaluate clustering quality and the scalability with respect to SPARK-DBSCAN with 2D and n-dimensional datasets. Finally, we study the ability of NG-DBSCAN to use arbitrary similarity metrics, by performing text clustering.

5.1 Analysis of the Parameter Space

Our approach has the following parameters: *i)* $\mathcal{T}_n$ and $\mathcal{T}_r$ regulate the termination mechanism; *ii)* k, the number of neighbors to build the neighbor graph, which influences the construction of the ε-NN graph; *iii)* M_{max}, the number of neighbors a node in the ε-NN graph must collect to be deactivated; *iv)* ρ, which limits the number of comparisons in extreme cases during Phase 1; *v)* $\mathcal{S}$, it is a parameter inherited by our implementation to limit the memory requirement in a given iteration. It limits the amount of nodes involved in the computation of an iteration.

5.1.1 Termination Mechanism

We start our evaluation by analyzing the termination mechanism; we use here the Twitter dataset (latitude and longitude values). Figure 3 shows the number of active (Active) and removed (Removed_Tot) nodes, and the removal rate (Removed) in subsequent iterations of the NG-DBSCAN algorithm. To help understanding the analysis, we include in the Figure also the clustering recall that we compute in every iteration of the algorithm. The results we present are obtained using $k = 10$ and $M_{max} = 20$; analogous results can be obtained with different configurations.

In the first 10 iterations, the number of active nodes is approximately equal to the entire vertex set of the ε-graph. This is the time required to start finding useful edges. Then, the number of active nodes rapidly decreases, indicating that a large fraction of the nodes reach convergence. Towards the last iterations, the number of active nodes reaches a plateau (due to noise points).

With reference to Section 3.1.4, the $\mathcal{T}_n$ threshold, that indicates the fraction of active nodes required before declaring the algorithm termination, avoids too early terminations that might occur if we only used the $\mathcal{T}_r$ threshold and corresponding inequality. Instead, the $\mathcal{T}_r$ parameter, which measures the rate at which nodes are deactivated in subsequent iterations, avoids both early terminations due to the $\mathcal{T}_n$ threshold and a lengthy and marginally beneficial convergence process.

In particular, without the $\mathcal{T}_r$ threshold, the algorithm would stop at the $\mathcal{T}_n$ threshold, that is – in our experiment – at iteration 18. As the recall metric of roughly 0.5 indicates, stopping the algorithm too early results in poor performance. Instead, with both thresholds, the algorithm stops at iteration 33, where the recall is greater than 0.9. Subsequent iterations only marginally improve the recall.

[8] `https://github.com/alitouka/spark_dbscan`

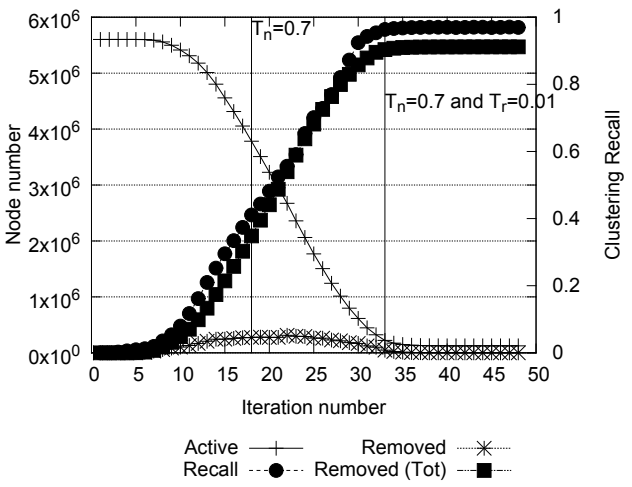


Figure 3: Analysis of the termination mechanism.

5.1.2 How to Set k

We now take in consideration the parameter k, which affects the number of neighbours in the neighbor graph, and perform clustering of the Twitter dataset (latitude and longitude values). We execute the algorithm 5 times for each configuration.

Figure 4a depicts the clustering recall we obtained with $k \in \{5, 10, 15\}$, as a function of the algorithm running time. Clearly $k = 5$ is not enough to obtain a good result, which confirms the findings of previous works on k-NN graphs [7, 15]. However, already with $k = 10$, the recall is considerably high, indicating that we retrieve approximately the same clusters as the exact DBSCAN algorithm. Increasing this parameter improves the quality of the result only marginally at the cost of a larger amount of algorithm runtimes. Due to the above considerations we think that $k = 10$ is an acceptable configuration value.

5.1.3 How to Set M_{max}

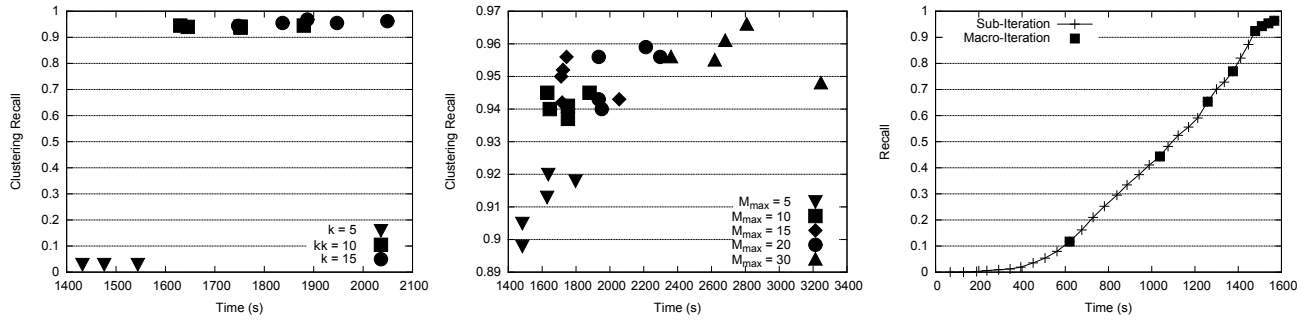
We analyse the impact of the M_{max} parameter using the Twitter dataset, and set $k = 10$. Results with different values of k lead to analogous observations. Figure 4b shows the clustering recall achieved for values of $M_{max} \in \{5, 10, 15, 20, 30\}$, as a function of the algorithm running time. We execute the algorithm 5 times for each configuration.

The recall value achieved by NG-DBSCAN is always larger than 0.9 when the algorithm stops. We also find that increasing M_{max} has positive effects on the recall: this confirms that a larger M_{max} improves the connectivity of the ε-graph in dense regions. However, there are "diminishing returns" when increasing M_{max}: a larger M_{max} value requires more time to meet the termination conditions because more edges must be collected in each node.

Overall, our empirical remarks indicate that M_{max} can be kept similar to k. In particular, values of $M_{max} \in [10, 20] = [k, 2k]$ give the better trade-offs between recall and completion time.

5.1.4 How to set ρ

The ρ parameter sets a limit to the number of nodes examined in the neighborhood of each node: if this were not done, in degenerate cases where nodes have a massive degree in the neighbor graph, the worst-case complexity of the first step of NG-DBSCAN could grow up to $\mathcal{O}\left(n^2\right)$. We have found, however, that our mechanism to remove nodes from the neighbor graph in practice already avoids the case in our experiments.



(a) How to set k.

(b) How to set M_{max}.

(c) Analysis of sub-iterations and parameter $\mathcal{S}$.

Figure 4: Analysis of the Parameter Space.

Table 2: How to set ρ.

ρ	1	2	3	6
Time (s)	3 985	2 303	2 233	2 241
Recall	0.089	0.95	0.944	0.951
Sub-Iterations	80	37	33	33

In Table 2 we show results for values of $\rho \in \{1, 2, 3, 6\}$. With a value of $\rho = 1$, the bound on the size of neighborhoods explored is too stringent, and NG-DBSCAN cannot explore new nodes quickly enough; as ρ grows, the algorithm performs better in terms of both recall and runtime.

We set a default value of $\rho = 3$ to ensure that algorithm terminates fast with good quality, while avoiding an increase in computational complexity for degenerate cases.

5.1.5 Sub-Iterations and $\mathcal{S}$

Figure 4c shows the amount of time to complete a sub-iteration and a macro-iteration as described in Section 3.1.3. As a reminder, the parameter $\mathcal{S}$ imposes a limit on the number of computing nodes in a given iteration. For instance, we set $\mathcal{S} = 500\,000$ for the Twitter dataset of $5\,602\,349$ tweets. This means that each sub-iteration involves approximately $500\,000$ nodes. The number of needed sub-iterations to complete the first macro-iteration should be $\lceil 5\,602\,349/500\,000 \rceil = 12$, but only 11 sub-iterations are actually necessary because some nodes already get deactivated in the first sub-iterations. As nodes get deactivated, macro-iterations become less and less expensive, requiring less sub-iterations and less time to complete. Since sub-iterations operate on approximately the same number of nodes, they keep a roughly constant size.

5.1.6 Parameters Discussion

We end this section discussing a set of parameters that we use as default, and give us a good trade-off between recall and completion time. $\mathcal{T}_n = 0.7$ and $\mathcal{T}_r = 0.01$ stop the algorithm when only very marginal benefit can be obtained by continuing processing; $k = 10$, $M_{max} = 2k$ and $\rho = 3$ yield a good trade-off between recall and run-time. In subsequent experiments we use the above configuration to evaluate NG-DBSCAN.

5.2 Performance in a 2D Space

We now move to a global evaluation of NG-DBSCAN, and compare the clustering quality we obtain to the baseline algorithm, namely the original DBSCAN, and to the SPARK-DBSCAN alternative. We use synthetically generated datasets and the latitude and longitude values of the Twitter dataset.

Table 3: Performance in a 2D space: Clustering Quality.

	NG-DBSCAN		SPARK-DBSCAN	
	Time (s)	Recall	Time (s)	Recall
Twitter	1 822	0.951	N/A	N/A
Circle	96	1	192	1
Moon	103	1	132	1
Blob	123	0.92	83	1

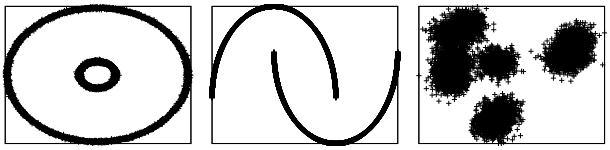
5.2.1 Clustering Quality

We begin with the synthetically generated datasets (described in Section 4.3) because they are commonly used to compare clustering algorithms. Figure 5 presents the shape of the three datasets called respectively *Circle*, *Moon* and *Blobs*. Each dataset has $100\,000$ items to cluster: such a small input size allows computing data clustering using the *exact* SciKit DBSCAN implementation and to make a preliminary validation of our approach. Results are presented in Table 3. NG-DBSCAN obtains nearly perfect clustering recall for all the datasets, when compared to the exact DBSCAN implementation. The completion time of NG-DBSCAN and SPARK-DBSCAN are comparable in such small datasets. It is interesting to note that while NG-DBSCAN is faster with moon and circle datasets characterized by two big clusters, it is slower in the blob dataset with respect to SPARK-DBSCAN. In the blob dataset, SPARK-DBSCAN partitions the space in a clever way to include each cluster in one partition. Instead, in the circle and moon dataset, each cluster covers multiple partitions, which slows down the algorithm.

In the Twitter dataset, NG-DBSCAN is able to achieve a good clustering recall, as described also in previous Sections. Instead, SPARK-DBSCAN is not able to complete the computation due to memory errors. To this end, we analyse in the next section the impact on the dataset size for both algorithms.

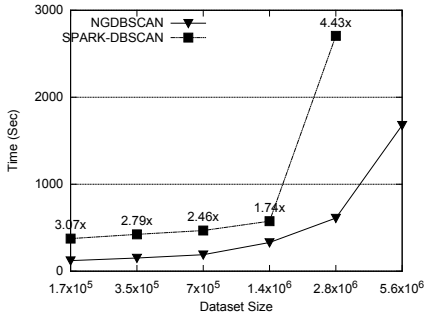
5.2.2 Scalability

We now study the scalability of NG-DBSCAN, and compare it to that of SPARK-DBSCAN. Figure 6a shows the algorithm runtime as a function of the dataset size, while using our entire compute cluster. We use 6 different samples of the Twitter dataset of size approximately $175\,000$, $350\,000$, $700\,000$, $1\,400\,000$, $2\,800\,000$ and $5\,600\,000$ (i.e., the entire dataset) tweets respectively. All values plotted are the average of 5 independent executions. Our results indicate that for smaller datasets, up to roughly $1\,400\,000$ samples, both SPARK-DBSCAN and NG-DBSCAN exhibit a linear scalability. For larger datasets, instead, the algorithm runtime increases considerably. In general, we note that SPARK-DBSCAN is al-

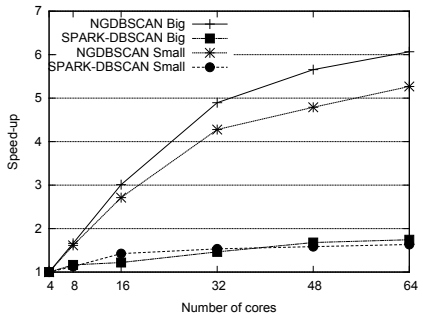


(a) Circle. (b) Moon. (c) Blobs.

Figure 5: Synthetic datasets plot.



(a) Scalability: Dataset Size.



(b) Scalability: Number of Cores.

Figure 6: Performance in a 2D space: Scalability.

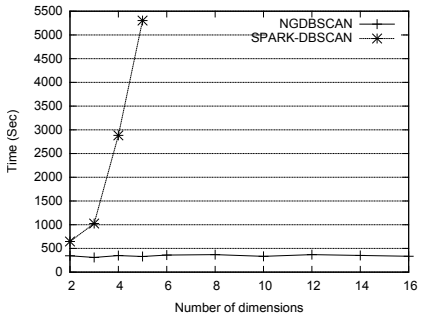
ways slower than NG-DBSCAN, by a factor of at least of $1.74\times$. In addition, SPARK-DBSCAN cannot complete the computation for the largest dataset, and with a size of $2\,800\,000$ it is already $4.43\times$ slower with respect to our NG-DBSCAN.

Figure 6b shows the algorithm speed-up of both NG-DBSCAN and SPARK-DBSCAN as the number of cores we devote to the computation varies between 4 and 64, considering a small dataset of $350\,000$ tweets and a larger dataset of $1\,400\,000$ tweets. Our results indicate that NG-DBSCAN always outperforms SPARK-DBSCAN, which cannot fully reap the benefits of more compute nodes. The speedup of NG-DBSCAN is sub-linear: past the cap of 32 cores, doubling the compute cores does not double the speedup. This is motivated by the fact that communication dominate computation costs.

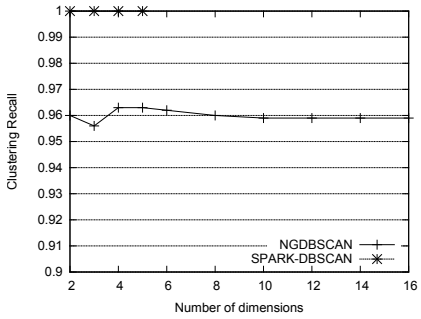
These results indicate that our approach is scalable – both as the dataset and cluster size grows. In our case, the time needed to compute our results with the configurations of Section 5.1.6 – which proved to be a desirable choice – is always in the order of minutes, demonstrating that our approach is viable in several concrete scenarios.

5.3 Performance in d-dimensional Spaces

Next, we evaluate the impact of d-dimensional datasets in terms of clustering quality and algorithm running time. For our experiments, we synthetically generate 10 different datasets, respectively of dimensionality $d \in \{2, 3, 4, 5, 6, 8, 10, 12, 14, 16\}$ of approximately $1\,500\,000$ elements each. The values in each dimension are a sample of the latitude and longitude values of the Twitter dataset.



(a) Time.



(b) Clustering Recall.

Figure 7: Performance in a d-dimensional space.

Figure 7a presents the algorithm runtime of both NG-DBSCAN and SPARK-DBSCAN as a function of the dimensionality d of the dataset. Results indicate that our approach is unaffected by the dimensionality of the dataset: algorithm runtime is roughly similar, independently of d. Instead, the running time of SPARK-DBSCAN significantly increases as the dimensionality grows: in particular, SPARK-DBSCAN does not complete for datasets in which $d \geq 6$. For small d, NG-DBSCAN significantly outperforms SPARK-DBSCAN.

Figure 7b shows the clustering recall of both NG-DBSCAN and SPARK-DBSCAN, as a function of d. In both cases, clustering quality is not affected by high dimensionality, albeit SPARK-DBSCAN does not complete for $d \geq 6$. The clustering recall of NG-DBSCAN settles at 0.96, which is slightly lower than that obtained by SPARK-DBSCAN: this is due to the approximate nature of our algorithm.

In conclusion, we NG-DBSCAN performs well irrespectively of the dimensionality of the datasets both in terms of runtime and clustering quality. This is a distinguishing feature of our approach, and is in stark contrast with respect to algorithms constructed to partition the data space, such as SPARK-DBSCAN and the majority of the state of the art approaches (see Table 1 in Section 2), for which the runtime worsens exponentially as the dataset dimensionality increases.

5.4 Performance with Text Data

We conclude our analysis of NG-DBSCAN by evaluating its effectiveness when using arbitrary similarity measures. In particular, we perform the evaluation using text data by means of two datasets:

Table 4: Spam and Tweets dataset: manual investigation.

Dataset	Cluster size	Sample
Spam	101 547	"[...]@[...].com Rolex For You -36%" "[...]@[...].com Rolex.com For You -53%" "[...]@[...].com Rolex.com For You -13%"
Spam	42 315	"Refill Your Xanax No PreScript Needed!" "We have MaleSex Medications No PreScript Needed!" "Refill Your MaleSex Medications No PreScript Needed!"
Spam	83 841	"[...]@[...].com VIAGRA Official -26%" "[...]@[...].com VIAGRA Official -83%" "[...]@[...].com VIAGRA Official Site 57% 0FF."
Twitter	7 017	"I just ousted @hugoquinones as the mayor of Preparatoria #2 on @foursquare! http://t.co/y5a24YMn" "I just ousted Lisa T. as the mayor of FedEx Office Print & Ship Center on @foursquare! http://t.co/cNUjL2L5" "I just ousted @sombrerogood as the mayor of Bus Stop #61013 on @foursquare! http://t.co/SwC3p33w"
Twitter	1 033	"#IGoToASchool where your smarter than the teachers !" "#IGoToASchool where guys don't shower . They just drown themselves in axe ." "#IGoToASchool where if u seen wit a female every other female think yall go together"
Twitter	23 884	"I'm at Walmart Supercenter (2501 Walton Blvd, Warsaw) http://t.co/4Mju6hCd" "I'm at The Spa At Griffin Gate (Lexington) http://t.co/Jb5JU8bT" "I'm at My Bed (Chicago, Illinois) http://t.co/n9UHV2UK"

the textual values of the Twitter dataset, and a collection of spam email subjects collected by Symantec. As distance metric, we use the Jaro-Winkler edit distance.

5.4.1 *Comparison with k-MEANS*

Since alternative DBSCAN implementations do not support Jaro-Winkler distance (or any other kind of edit distance), we compare our results with those obtained using k-MEANS on text data converted into vectors using `word2vec`, as described in Section 4.4. To proceed with a fair comparison, we first run NG-DBSCAN and use the number of clusters output by our approach to set the parameter K of k-MEANS.

We begin with a manual inspection of the clusters returned by NG-DBSCAN: results are shown in Table 4. We report 3 clusters for each dataset, along with a sample of the clustered data. Note that subjects or tweets are all related, albeit not identical. Clusters, in particular in case of the Spam dataset, are quite big. This is of paramount importance because specialists usually prefer to analyse large clusters with respect to small clusters. For instance, we obtain a cluster of 42 315 emails related to selling medicines without prescription, and a cluster of 23 884 tweets aggregating text data of people communicating where they are through Foursquare.

Next, we compare NG-DBSCAN with k-MEANS using the well-known internal clustering validation metrics we introduced in Section 4.2. Recall that compactness measures how closely related the items in a cluster are, whereas separation measures how well clusters are separated from each other. We perform several experiments with both Twitter and Spam datasets: Table 5 summarizes our results.

For what concerns compactness, higher values are better and both NG-DBSCAN and k-MEANS behave similarly. However, in the full Spam dataset, we are unable to complete the computation of k-MEANS: indeed, the k-MEANS running time is highly affected by its parameter K. In this scenario we have K$=$ 17 704 and the k-MEANS computation does not terminate after more than 10 hours. Hence, we down-sample the Spam dataset to 25% of its original size (we have the very same issues with a sample size of the 50%). With such reduced dataset, we obtain K$=$ 3 375 and k-MEANS manages to complete, although its running time is considerably longer than that of NG-DBSCAN. The quality of the clusters produced by the two algorithms are very similar.

For the separation metric, where lower values are better, NG-DBSCAN clearly outperforms k-MEANS. In particular in the Twitter dataset we achieve 0.2 instead of 0.42 suggesting that the clusters are more separated in NG-DBSCAN with respect to k-MEANS.

Table 6: Metrics comparison with NG-DBSCAN for Twitter.

metrics	#cluster	max size	C	S	Time
Jaro-Winkler	1 605	58 973	**0.65**	**0.2**	2 980
`word2vec` + euclidean	3 238	24 117	0.64	0.29	2 908

5.4.2 *Impact of Text Transformation Techniques*

One of the most common way to cluster text data is to convert it to a vector representation, for instance using text transformation techniques such `word2vec`. Next, we want to discover if preprocessing the data introduces a loss of clustering quality with respect to using a similarity metric that is tailored to the data. In this experiment we compare the quality of the clusters returned using the JaroWinkler similarity metric and the Euclidean distance working on the vectors generated by `word2vec`. Note that in both the scenarios we use the NG-DBSCAN algorithm since it is conceived to run with arbitrary similarity metrics.

Table 6 presents the results using the Twitter dataset. Our results indicate that indeed transforming text to a vector representation induces a clustering quality loss: the cluster separation is worse, and clusters are more fragmented (i.e., more clusters of smaller size) when NG-DBSCAN uses the traditional `word2vec` transformation. This result emphasizes another important feature of NG-DBSCAN: it allows working with arbitrary distance metrics, which can be tailored to the data under examination.

6. CONCLUSION

Data clustering and analysis is a fundamental task in data mining and exploration. However, the need to analyze unprecedented large amounts of data require novel approaches to algorithm design, often calling for parallel frameworks that support flexible programming models, while operating on large scale clusters.

We presented NG-DBSCAN, a novel distributed algorithm for density-based clustering that produces quality clusters with arbitrary distance measures. This is of paramount importance because it allows "separation of concerns": domain experts can chose the similarity function that is most appropriate for their data, given

Table 5: Evaluation using text data: Twitter and Spam datasets comparison with k-MEANS. "C" stands for compactness and "S" for separation.

Algorithm	Twitter			Spam			Spam 25%		
	C	S	Time	C	S	Time	C	S	Time
NG-DBSCAN	0.65	0.2	2 980	0.88	0.63	4 178	0.88	0.66	654
k-MEANS	0.64	0.42	4 477	N/A	N/A	N/A	0.84	0.67	27 557

their knowledge of the context; instead, the intricacies of parallelism can be addressed by designers who are more familiar with framework APIs than with the peculiar data at hand.

We showed, through a detailed experimental campaign, that our approximate algorithm is on-par with the original DBSCAN algorithm, in terms of clustering quality, for d-dimensional data. However, NG-DBSCAN scales to very large datasets, outperforming alternative designs. In addition, we showed that NG-DBSCAN correctly groups text data, using a carefully chosen similarity metric, outperforming a traditional approach based on the k-MEANS algorithm. We supported our claims using both synthetically generated data and real data: a collection of real emails classified as spam by Symantec security systems, and used to discover spam campaigns; and a large number of tweets collected in the USA, which we used to discover tweet similarity.

Our next steps include the analysis of the asymptotic behaviour of the first step of NG-DBSCAN and its convergence time. We also plan to devise an extension to NG-DBSCAN to adjust the parameter k in a dynamic manner, by "learning" appropriate values while processing data for clustering. In addition, we will consider the problem of working on "unbounded data", which requires the design of on-line, streaming algorithms, as well as the problem of answering ε-queries in real time.

7. REFERENCES

[1] Apache Spark. `https://spark.apache.org`.

[2] Apache Spark machine learning library. `https://spark.apache.org/mllib/`.

[3] Clustering the News with Spark and MLLib. `http://bigdatasciencebootcamp.com/posts/Part_3/clustering_news.html`.

[4] Word2vector package. `https://code.google.com/p/word2vec/`.

[5] B.-R. Dai, I. Lin, et al. Efficient map/reduce-based dbscan algorithm with optimized data partition. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 59–66. IEEE, 2012.

[6] B. Desgraupes. Clustering indices. *University of Paris Ouest*, 2013.

[7] W. Dong et al. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proc. of ACM WWW*, 2011.

[8] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, pages 226–231, 1996.

[9] J. Gan and Y. Tao. Dbscan revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 519–530. ACM, 2015.

[10] Y. He, H. Tan, W. Luo, H. Mao, D. Ma, S. Feng, and J. Fan. Mr-dbscan: An efficient parallel density-based clustering algorithm using mapreduce. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 473–480. IEEE, 2011.

[11] M. A. Jaro. Probabilistic linkage of large public health data files. *Statistics in medicine*, 14(5-7), 1995.

[12] Y. Kim, K. Shim, M.-S. Kim, and J. S. Lee. Dbcure-mr: an efficient density-based clustering algorithm for large data using mapreduce. *Information Systems*, 42:15–35, 2014.

[13] Y. Liu et al. Understanding of internal clustering validation measures. In *Proc. of IEEE ICDM*, 2010.

[14] A. Lulli et al. Cracker: Crumbling large graphs into connected components. In *Proc. of IEEE ISCC*, 2015.

[15] A. Lulli et al. Scalable k -nn based text clustering. *IEEE BigData*, 2015.

[16] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Comput. Surv.*, 48(2):25:1–25:39, Oct. 2015.

[17] M. M. A. Patwary, N. Satish, N. Sundaram, F. Manne, S. Habib, and P. Dubey. Pardicle: parallel approximate density-based clustering. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 560–571. IEEE Press, 2014.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.

[19] T. N. Tran, R. Wehrens, and L. M. Buydens. Knn-kernel density-based clustering for high-dimensional multivariate data. *Computational Statistics & Data Analysis*, 51(2):513–525, 2006.

[20] B. Welton, E. Samanas, and B. P. Miller. Mr. scan: Extreme scale density-based clustering using a tree-based network of gpgpu nodes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 84. ACM, 2013.

[21] X. Xu, J. Jäger, and H.-P. Kriegel. A fast parallel clustering algorithm for large spatial databases. In *High Performance Data Mining*, pages 263–290. Springer, 2002.

[22] S. Zhou, Y. Zhao, J. Guan, and J. Huang. A neighborhood-based clustering algorithm. In *Advances in Knowledge Discovery and Data Mining*, pages 361–371. Springer, 2005.